Roles From an Aspect-Oriented Perspective

Stefan Hanenberg
Dominik Stein
Rainer Unland

University of Duisburg-Essen
Essen, Germany
{shanenbe, dstein, unlandR}@cs.uni-essen.de

Abstract. Roles and aspects have become more and more popular in the recent past. Although similarities as well as differences have been often discussed in literature, there is still no clear separation between both approaches, nor a convincing statement whether there is a difference at all. One reason for this confusion is caused by the fact that there is a variety of role systems as well as aspect-oriented systems which provide quite different sets of mechanisms for similar concepts. This paper provides a unified view on aspect-oriented systems based on design dimensions for aspect-oriented systems, and maps the notion of roles to such design dimensions. Based on this mapping, this paper argues that role systems can be seen as a special kind of aspect-oriented systems having particular characteristics.

1 Introduction

Roles have been introduced in many different variants to support modern software development [15]. Their ability to adapt the structure and behavior of objects depending on a particular context gave rise to multiple studies (e.g. [1], [9], [7], [16]) investigating how and in what extent roles compare to aspects in Aspect-Oriented Software Development (AOSD) [2], which – after all – are able to do the same thing: Aspects in AOSD may enhance a given set of objects with additional features and may influence their dynamic behavior. The comparisons are difficult to conduct, though, since different implementations of both role systems and aspect-oriented systems differed widely in their realization of the envisioned role or aspect concepts, respectively. For example, in the aspect-oriented world, there exists a number of different interpretations of the join point concept and corresponding implementations [7]. Likewise, there are several role systems that realize roles in different ways [15]. Hence, it is difficult to detect constitutional characteristics of either system type, and to compare them to each other so that to come to a meaningful conclusion.

In this paper, we present a comprehensive set of design dimensions that we have developed for the assessment of aspect-oriented systems. Its goal is to permit the formulation of qualified statements on the commonalities, differences, and individual capabilities of different aspect-oriented systems. We introduce the different design dimension step by step and, afterwards, apply them to role systems (as introduced in [11] and [12]). In taking an aspect-oriented look at role systems, we are able to identify in what respect role systems compare to aspect-oriented systems, and why and

where there are differences.

The remainder of the paper is structured as follows: In the next section 2, a short overview to roles is given with help of an example. After that (section 3), the various design dimensions are introduced. In section 4, we map role systems to those design dimensions. And finally, in section 5, we conclude the outcomes of our investigation.

2 Roles

Roles (cf. [11] and [12]) are (possibly temporary) views on objects. They define extra properties which are added to objects. These role properties can be regarded as subjective *extrinsic* properties of an object, which can be accessed from the object's environment just like the object's very own *intrinsic* properties. During its lifetime, an object may adopt and abandon roles. In consequence, different role and role properties may be accessible at different points in time. A the same time, an object may play different roles simultaneously.

One interesting property of roles (especially when compared to aspect-oriented programming) is their ability to change the behavior of an object. While [3] refers to this as an intrinsic feature of roles, [11] and [12] deem this to be a special kind of role, which they call *method role*: A method role is a method (of a role) which is bound to an intrinsic method of an object.

For example, Fig. 1 illustrates a person who has two jobs as a bartender. A job is represented as a (temporal) role in this example because persons usually do not keep it for their whole lifetime. The person has a couple of intrinsic properties, such as the name and the date of birth, which are not influenced by any role. Furthermore, though, the person has other properties, such as the telephone number and the income, whose meaning depend on the role the person is playing: In spare time, the telephone number should refer to the person's private (intrinsic) telephone number. During working time, the telephone number should refer to the person's (extrinsic) duty telephone number (i.e. the phone number of the bar where the person is working at). To cope with that situation, *method roles* are used: Depending on the context in which the telephone number is requested, either the private number or the phone number of the bar is returned.

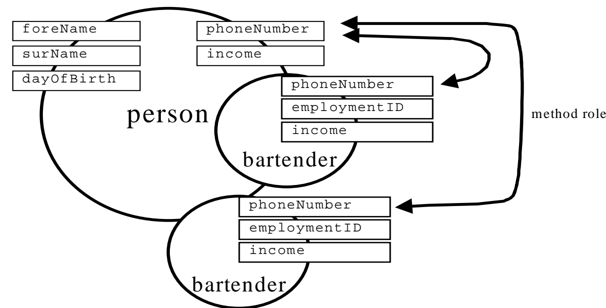

Fig. 1. Object person with two bartender roles

3 Design Dimensions of Aspect-Oriented Systems

In an aspect-oriented system, developers must be able to specify *where* an aspect

tackles the base system and *how* the aspect adapts the base system. To specify the "where", developers require features that allow the designation of (a set of) join points to which the aspect should be woven to. To specify the "how", developers require features which permit to describe how exactly each of the chosen join points is to be adapted. In brief, an aspect-oriented system needs to provide a join point representation of the base system, some language constructs that permit developers to select join points, language constructs that permit developers to specify how selected join points are to be adapted, and a weaver that composes the resulting system.

A **join point representation** is a specific view on the base system which is extracted directly from the base application. It is the task of an aspect-oriented system to decompose the base application into a number of join points (see circles in Fig. 2). To do so, an aspect-oriented system needs to determine what elements of the base application actually represent join points. This decision is based on the aspect-oriented system's **join point model**.

To select join points, aspect-oriented systems provide a **selection language** that permits developers to address some of the join points the base system is decomposed into (see triangles in Fig. 2). This implies that the system must equip join points with a number of characteristic marks or properties that developers can refer to, and upon which the aspect-oriented system can decide whether or not a particular join point is to be selected. These characteristic marks are extracted from the base application, and are part of the join point representation. We call the extraction of such join point properties **join point encoding**.

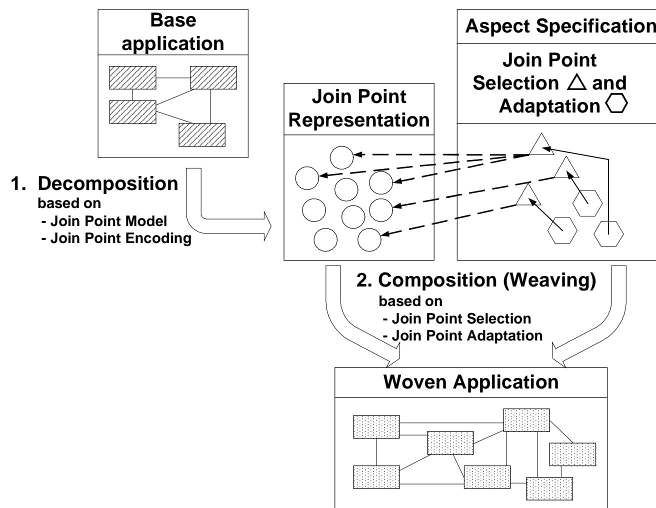

Fig. 2. Schematic illustration of aspect-oriented systems [5]

Once join point selections are specified, developers need language constructs that refer to a join point selection and that specify how the selected join points are to be adapted. Such **join point adaptations** are illustrated by means of hexagons in Fig. 2.

Finally, the aspect-oriented system needs to weave the resulting system. To do so, the **weaver** takes the base system, its join point representation, as well as the user-defined selections and adaptations, and composes the woven system[1].

[1] Note, the design dimensions for weavers (i.e. the techniques being used to compose aspects with the base system) are beyond of the scope of this paper and are not explained in the following sections.

3.1 Design Dimensions of Join Point Models

The notion of join point differs widely in different aspect-oriented systems. In this section we illustrate two design dimensions, reflecting on the different interpretations of the join point concept.

Level of Dynamicity: Static and Dynamic Join Points

Different aspect-oriented systems have different join point notions with respect to their dynamicity. Systems like Hyper/J [14] or Sally [6] permit to select and adapt elements which have a direct correspondence in the program code; systems like AspectJ [10] or AspectS [8] permit to select and adapt join points due to runtime-specific information. Therefore, we distinguish between static join points and dynamic join points:

- **Static join point**: A static join point is a selectable and adaptable item that represents an element in the base program's code.
- **Dynamic join point:** A dynamic join point is a selectable and adaptable runtime-element from the application's execution context.

An example for a static join point is a method definition in Hyper/J or Sally; an example for a dynamic join point is a method execution in AspectJ (when referred to by means of dynamic pointcuts).

Level of Abstraction: Structural and Behavioral Join Points

Furthermore, the join point notions in different aspect-oriented systems differ with respect to their level of abstraction. Systems like AspectJ or Sally permit to select and adapt method calls, while systems like AspectS permit to select and adapt method definitions. Accordingly, we distinguish between structural and behavioral join points:

- **Structural join point**: A structural join point is a selectable and adaptable element that represents a structural abstraction within the based application, based on an abstraction provided by the underlying programming language.
- **Behavioral join point**: A behavioral join point is a selectable and adaptable element in the application that represents a part of the application's behavior, which is encapsulated by corresponding structural building blocks.

Orthogonality of Level of Dynamicity and Level of Abstraction

The dimensions "level of abstraction" and "level of dynamicity" can be applied independent of each other, i.e. they are orthogonal. In consequence, join point models can be divided along both dimensions at the same time.

Both orthogonal dimensions are illustrated in Fig. 3. Aspect-oriented systems do not necessarily need to provide only one single point in the matrix. For example, introductions in AspectJ are applied to static, structural join points (type definitions), while call-pointcuts in AspectJ refer to dynamic behavioral join points.

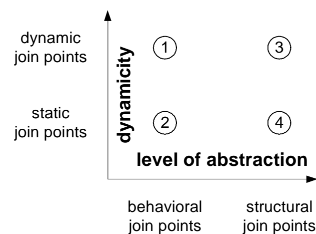

Fig. 3. Orthogonal dimensions of join point models

3.2 Design Dimensions of Join Point Properties

Different aspect-oriented systems provide different properties for join points. In AspectJ, for example, method call join points can be selected based on the signature of the called method, consisting of the method name, return type, and parameter types. Hence, *signature* represents a property of a method call join point in AspectJ, while a specific signature like `void m()` represents the value of that signature property for a certain join point.

Level of Dynamicity: Static and Dynamic Properties

A fundamental difference between join point properties is their relationship to runtime data in the base system. A property like `this` in AspectJ, for example, is based on dynamic system information (or *some portion of the program state* [17]). A property referring to some parameter of `declaredMethod` in Sally, on the other hand, is based on static system information (i.e. information that is available at a static analysis of the base application).

- **Static join point property**: A join point property is static if its values can be directly derived from the base system's code.
- **Dynamic join point property**: A join point property is dynamic if runtime-specific information is needed in order to provide the corresponding values.

Directness of Property Correspondence: Direct and Abstract Property Correspondence

The "directness of property correspondence" dimension distinguishes between properties that can be directly derived from the data available at the corresponding join point, and those that require further computations. For example, the declared return type of a method declaration join point represents a direct correspondence property because the return type (or the name of the return type) directly appears in the source code[2]. Other properties, such as "number of parameters", however, do not (literally) appear in the source code; it must be computed from the method parameter list at the corresponding method declaration join point. Accordingly, we distinguish:

- **Direct property correspondence:** A property has a direct correspondence if the property can be directly derived from the data available at the corresponding join point.
- **Abstract property correspondence:** A property has an abstract correspondence if the property does not directly represent some data available at the corresponding join point. Additional computations by the aspect-oriented system are needed in order to provide the corresponding property.

The dimensions "level of dynamicity" of join point properties and "directness of property correspondence" are orthogonal to each other (see Fig. 4).

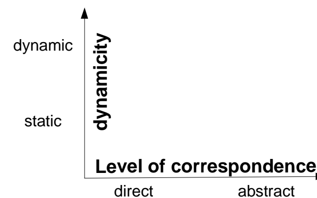

Fig. 4. Orthogonal dimensions of join point properties

[2] Such properties are prerequisite for the term **lexical crosscutting** as introduced in [13] which says that *the join points […] consist of names that appear in the implementation […].*

3.3 Design Dimensions of Properties Addressing

Having extracted join points and their properties from a base application, aspect-oriented systems need to provide means to constrain the join point properties for actual selection. In the following, two (orthogonal) design dimensions are described that categorize possible ways to do so (see Fig. 5).

Level of Dynamicity: Lexical and indirect addressing

In the most trivial case, the aspect-oriented system provides language constructs that permit developers to specify the value of a certain property. This corresponds, for example, to the specification of a class name within a type pattern in AspectJ. This class name is lexically compared against all class names in the application, and all matching classes are selected. This kind of join point selection is the most straight forward way and all known aspect-oriented system provide it. *Lexical crosscutting* [13] as well as *enumeration-based crosscutting* [4] are based on such an join point property addressing.

In contrast to this, AspectJ allows to specify join point selections without referring to the join point's properties in a lexical way by using the + operator. By doing so, it is possible to select a join point based on the type relationship between a join point property and another type. That is, there is no lexical correspondence between the specified selection criteria and the matching property value.

Recognizing this substantial difference, we identify the "directness of value addressing" as a design dimension of join point selection language constructs and distinguish between lexical value addressing and indirect value addressing:

- **Lexical value addressing**: A language constructs for the selection of join points provides a lexical value addressing if the developer needs to specify the lexical characteristics of the property's value.
- **Indirect value addressing**: A language construct for the selection of join points provides an indirect value addressing if the developer may specify characteristics of the property's value in a non-lexical way.

Closed and open value addressing

In AspectJ and Hyper/J, it is possible to use wildcards for addressing join point properties. Such wildcards permit to specify a value of a certain join point property not (necessarily) in its entirety, but (also) only in parts. This is particularly interesting when base applications are based on a number of naming conventions.

The characteristic of such an addressing in comparison to a pure enumeration of properties is that it refers to an infinite set of possibly matching join point properties: Join points that are added to the system during system evolution can be taken care of. With that in mind, we distinguish between closed value addressing and open value addressing and call the design dimension the "openness of value addressing":

- **Closed value addressing**: An aspect-oriented system provides a closed value addressing if it allows to specify a finite set of values for a given property, i.e. it is known how many values match the selection at specification time.
- **Open value addressing**: A join point selection language permits an open value specification if it permits to specify an infinite set of values for a given property.

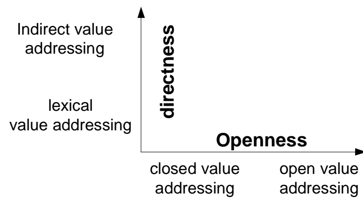

Fig. 5. Orthogonal dimensions of join point property addressing

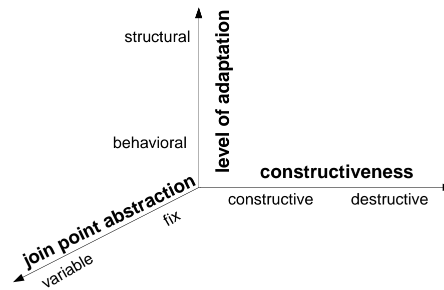

Fig. 6. Orthogonal design dimensions of join point adaptation

3.4 Design Dimensions of Join Point Adaptation

The design dimensions for join point models and join point selections determine the (characteristics of) elements that can be selected and (eventually) adapted. However, they do not describe how the adaptation can be actually accomplished. This section describes three (orthogonal) design dimensions along which join point adaptation techniques can be classified (see Fig. 6).

Level of Adaptation: Structural and Behavioral Adaptation
The first dimension is based on the observation that introductions and advice in AspectJ, for example, differ substantially: Introductions are structural adaptations while an advice is a behavior adaptation. Consequently, we identify the design dimension "level of adaptation" and distinguish between structural join point adaptation and behavioral join point adaptation.

- **Structural join point adaptation:** An aspect-oriented system provides structural join point adaptations if there are constructs that permit to change the structure of the join points such constructs refers to.
- **Behavioral join point adaptation:** An aspect-oriented system provides behavioral join point adaptations if there are constructs that permit to change the behavior of the join point the construct refers to (without changing its structural appearance).

Level of Constructiveness: Constructive and Destructive Adaptation
Advice in AspectJ and bracket relationships in Hyper/J permit to change the behavior at behavioral join points. This can be accomplished such that the original join point remains (potential) part of the system, and only additional behavior is executed before or after the join point (e.g. by means of before and after advice in AspectJ, or an around advice with a call to proceed). On the other hand, though, the adaptation at a given join point may also lead to the definite exemption of that join point from the system (e.g. by means of an around advice in AspectJ, without a call to proceed).

Based on this observation, we identify the "constructiveness of an adaptation" as a separate design dimension for join point adaptations, and distinguish between destructive join point adaptation and constructive join point adaptation:

- **Destructive join point adaptation:** An aspect-oriented system provides destructive join point adaptations if the adaptation necessarily replaces its join point,i.e. the original join point is no longer (potential) part of the application.
- **Constructive join point adaptation:** An aspect-oriented system provides constructive join point adaptations if the adaptation does not necessarily change the original join point such that the join point has no longer the same properties after weaving as it had before weaving.

Level of Join Point Abstraction: Variable and Fix Abstraction

Systems like AspectJ provide the ability to expose context information (from the base application) to the advice by means of pointcut parameters. That is, the context information to be used within the join point adaptation is defined in the corresponding join point selection (among others). Consequently, the data available for adaptation at a certain join point possibly varies from adaptation to adaptation.

This is unlike to systems like AspectS. Here, each (method execution) join point is provided with a list representing the list of parameters being passed to the method. Imagine now that a single adaptation needs to refer to two method execution join points with two different shadow join points, and that the adaptation needs to refer to the first method's first parameter and the second method's second parameter. Then, it is up to the developer to select the appropriate parameter within the adaptation (rather than within the selection).

Reflecting on this difference, we identify the "level of join point abstraction" as another design dimension of aspect-oriented systems, and distinguish between fix and variable join point abstraction:

- **Fix join point abstraction:** An aspect-oriented system provides a fix join point abstraction if the join point abstraction's context is fix for all join points
- **Variable join point adaptation:** An aspect-oriented system provides a variable join point abstraction if the developer can individually specify what context information the adaptation operates on.

4 Mapping Role Models to Design Dimensions

In order to regard role models from the aspect-oriented perspective, we now map their characteristics to the previously described design dimensions.

4.1 Join Point Model

In order to understand the join point model in role systems, it is necessary to discuss the different selectable and adaptable elements in role systems:

First, roles can be applied (or "bound") to objects. The role enhances the interface of the object which it is applied to. Consequently, an object represents a join point for roles. In class-based languages, objects represent *dynamic* and *structural* join points.

Second, method roles are defined for particular methods of the objects which the role is applied to. Such methods also represent *structural* join points. Since method roles adapt only methods of objects (which they are applied to), method-roles also refer to *dynamic* structural join points.

4.2 Join Point Properties and Property Addressing

Join point properties in role systems are, first of all, the target objects to which a particular role is applied. Furthermore, the target method is characterized by its method signature.

The target object is obviously a *dynamic* property which has a *direct correspondence*. The property is being addressed *lexically* as the value of the property has to be specified directly, i.e. the very object has to be delivered. Beyond that, the property is being addressed in an *open* way as roles can be applied to arbitrary objects.

The way of how signatures are represented depends strongly on the underlying object-oriented language: On the one hand, all known object-oriented languages identify methods by name. Such a name can be directly derived from the application's source code. Consequently, the method name is a *static* property with a *direct correspondence*. On the other hand, typed languages might provide additional properties for signatures that refer to the parameter types and the return type, for example. In case, method roles require a static matching of parameter and return types defined in the target and role method, there is a (structured) property referring to the declared parameter types, and a property referring to the declared return type. Again, both such properties are *static* properties with a *direct correspondence*.

In the role model as introduced in [11] and [12], all elements of the signature need to be addressed in a *lexical* way. Furthermore, each property needs to be specified in a *closed* way.

4.3 Join Point Adaptation

The two ways of adapting join points are roles themselves and method roles defined within roles.

Roles define a number of new members for objects they are applied to. Consequently, roles (containing at least one member) represent *structural* adaptations of objects. From the role model introduced in section 2, it is not clear whether roles can override methods, i.e. whether a method defined within a role (no method role!) could override a method already existent in the bound object. In case this is not possible, each method defined in a role is a pure *constructive* join point adaptation. In respect to its join point abstraction the context of a role is determined by the complete join point – the way of how a target object is selected does not change the context in which the adaptation occurs. Consequently, roles have a *fix* join point abstraction.

Method roles represent another kind of adaptation unit: Their underlying join points are method definitions (of the bound object). In contrast to ordinary members defined within roles, method roles do not change the object's structure – they rather adapt the behavior of a certain method. Consequently, a method role is a *behavioral* adaptation of the dynamic structural method definition join point.

The adaptation of the original method could be constructive as well as destructive: In case the method role refers to the original method, the adaptation is *constructive*; in case the method role does not refer to the original method, the adaptation is *destructive*. In respect to the join point abstraction, method roles have the same context as their join point – they have the same parameters, they refer to the same object, and they have an additional element in the context which refers to the role itself. Consequently, a method role does not have a context defined by the corresponding selection. Hence, a method role has a *fix* join point abstraction (too).

5 Related Work

[9] investigates different ways how roles can be implemented with aspects. Several options are discussed: The first option is to define role members in an aspect, which are then statically introduced to a class (this leads to a violation of role dynamicity, though). Another possibility is to use aspects to modify the behavior of already existent members of objects. In its ultimate, this means that all role members are statically implemented in an object, and dynamic aspects intercept any access to "invalid" role members (and/or adapt any access to "valid" role members in a role-specific way). Apart from that, all role members can be encapsulated in an aspect (i.e. an aspect instance), which maintains the role relationships to objects as well as the role context (this enables role multiplicity, in particular). Finally, aspects can be used to "glue" objects to "role objects" (i.e. regular objects that implement the role members).

Although the work presented in [9] unveils some of the core characteristics of aspect-oriented systems, it deems aspect-orientation to be a promising yet complementary technique to implement role models rather than to be a software development approach on its own. Consequently, it does not contemplate on the similarities and differences between the conceptual ideas of aspect-oriented systems and role systems, and focuses on implementation concerns only.

One of the first comparisons of the conceptual ideas of aspect-oriented systems and role systems has been conducted in [7]. The approach derives a couple of conceptual characteristics of aspects by investigating prominent aspect-oriented programming languages (i.e. AspectJ and HyperJ): *Distributed Effects* (aspects may affect multiple objects at the same time), *Cardinality* (aspect methods, e.g. advice, may be applied to multiple methods), *Context Dependency* (aspects may affect objects and actions depending on a certain context, e.g. depending on the caller of a method invocation), *Deployment* (aspects are applied to objects indirectly in terms of join point selections).

These characteristics are compared to the conceptual characteristics of roles as mentioned in [11] and [12]: *Visibility* (roles can restrict the visibility and access to objects), *Dependency* (roles cannot exist without their affiliated objects), *Identity* (an object and its role can be manipulated and viewed as one entity), *Dynamicity* (roles can be added and removed during an object's lifetime), *Multiplicity* (an object can have more than one instance of the same role at the same time), *Abstractivity* (roles can be classified and organized in generalization and aggregation hierarchies), *Extension only* (a role can only add further properties to the original object and cannot remove any).

Fundamental differences have been identified with respect to the *Identity* and *Multiplicity* of roles: Aspects do not have to be instantiated for every object they are applied to. Furthermore, the same aspect may influence multiple objects at the same time. Hence, aspects and objects usually do not form a single entity. Apart from that, the same aspect is usually not applied to the same object multiple times (even tough this would be technically possible). On the other hand, roles differ from aspects with respect to their *Distributed Effects* (unlike aspects, roles affect a single object only), *Cardinality* (a role method is usually assigned to a single method (of the affiliated object)), *Context Dependency* (roles usually do not affect objects depending on a varying context[3]), *Deployment* (roles are affiliated to objects directly by enumerating

[3] Although it should be mentioned that Kristensen introduces the (conceptual) notion of *Local-*

their names).

The outcomes of [7] are elucidating yet somewhat ad-hoc. The main problem is that only implementation-specific details of particular aspect-oriented systems (i.e., of AspectJ and Hyper/J) are considered. Therefore, the outcomes cannot be considered to represent conceptual foundations for aspects and aspect-oriented systems in general. The dimensions presented in this paper, on the contrary, can be used as a general gauge to assess the conceptual differences of arbitrary aspect-oriented systems. In [5], we demonstrate how different aspect-oriented systems map to these design dimensions. Furthermore, we elucidated how the dimensions can help in identifying aspect-oriented systems that could be candidates for the implementation of a particular problem.

6 Conclusion

In this paper, we have presented a number of (orthogonal) design dimensions that we have derived for the assessment of different aspect-oriented systems. These dimensions have been identified by investigating the differences and commonalities of different existing aspect-oriented systems. The dimensions include characteristics of the join point model, the join point properties, the join point property addressing, as well as the join point adaptation. Then, we have taken a role systems (the one introduced in [11] and [12]) and mapped it against the design dimensions. In doing so, the difference of this role system compared to other (particular) aspect-oriented systems become manifest:

The role system described in [11] and [12] compares to (other) aspect-oriented systems in that it provides join points which can be selected and adapted, and in that it provides the language constructs to actually do so (i.e. it allows the selection of join points by addressing particular join point properties, and it allows the adaptation of the selected join points). Differences become manifest in the precise capabilities of the role system to select and adapt join points. In particular, the join point selection capabilities are rather limited: For example, it is not possible to apply method roles to an arbitrary set of target methods in an *open* way. In consequence, the role system is rather inflexible against system evolution. Likewise, the role system does not provide capabilities to *abstract* over the join points' context, such as method parameters – which, for example, is possible in AspectJ using the `args` pointcut designator. In consequence, developers have no means to narrow the number of parameters at different method definition join points to those which are of actual interest.

The observations made in this paper are not necessarily generalizable to all role systems. Other role system implementations may fill into other quadrants of the described design dimensions. Consequently, they compare more or less to other particular aspect-oriented systems. Thanks to the design dimensions presented in this paper, we have now a gauge at hand that allows a closer estimation of the similarities and the differences between two arbitrary systems (no matter if they are called "role-oriented" or "aspect-oriented").

---

ity, according to which a role is available only in a certain context. Within this context, though, roles act context-insensitive, e.g. do not distinguish between different callers.

References

[1] Bardou, D., *Roles, Subjects and Aspects: How do they relate?*, Aspect Oriented Programming Workshop (AOP) at ECOOP '98, Brussels, Belgium, July 21, 1998

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

[3] Gottlob, G., Schrefl, M., Röck, B., *Extending Object-Oriented Systems with Roles*, ACM Transactions on Information Systems, Vol. 14, No. 3, July 1996, pp. 268-296

[4] Gybels, K., Brichau, J., *Arranging Language Features for More Robust Pattern-based Crosscuts*, in: Proc. of AOSD 2003, Boston, MA, ACM, pp. 60-69

[5] Hanenberg, S., Stein, D., Unland, R., *Eine Taxonomie für aspektorientierte Systeme*, in: Proc. of SE'05, Essen, Germany, LNI 64 GI 2005, pp. 167-178

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

[7] Hanenberg, S., Unland, R., *Roles and Aspects: Similarities, Differences, and Synergetic Potential*, in: Proc. of OOIS'02, Montpellier, France, LNCS 2425, pp. 507-520

[8] Hirschfeld, R., *AspectS - Aspect-Oriented Programming with Squeak*, in: Proc. of NODE'02, Erfurt, Germany, LNCS 2591, pp. 216-232

[9] Kendall, E.A., *Role Model Designs and Implementations with Aspect-oriented Programming*, in: Proc. of OOPSLA'99, Denver, CO, SIGPLAN Notices 34(10), pp. 353-369

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

[11] Kristensen, B.B., *Object-Oriented Modeling with Roles*, in: Proc. of OOIS'95, Dublin, Ireland, Springer, 1995, pp. 57-71

[12] Kristensen, B.B., Østerbye, K., *Roles: Conceptual Abstraction Theory & Practical Language Issues*, in: Proc. of TAPOS, Vol. 2, No. 3, 1996, pp. 143-160

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

[14] Ossher, H., Tarr, P., *Using multidimensional separation of concerns to (re)shape evolving software*, Communication of the ACM, 44 (10), 2001, pp. 43-50

[15] Steimann, F., *On the representation of roles in object-oriented and conceptual modelling*, Data & Knowledge Engineering, 35 (1), 2000, pp. 83-106

[16] Steimann, F., *Why most domain models are aspect free*, Aspect-Oriented Modeling Workshop (AOM) at UML'04, Lisbon, Portugal, October 11, 2004

[17] Wand, M., Kiczales, G., Dutchyn, C., *A Semantics for Advice and Dynamic Join Points in AspectOriented Programming*, Workshop on Foundations Of Aspect-Oriented Languages (FOAL) at AOSD'02, Enschede, The Netherlands, April 22, 2002