Roles and Aspects: Similarities, Differences, and Synergetic Potential

Stefan Hanenberg and Rainer Unland

Institute for Computer Science, University of Essen, 45117 Essen, Germany
{shanenbe | unlandR}@cs.uni-essen.de

Abstract. Both, the role concept and aspect-oriented programming are techniques which permit a flexible adaptation of object-oriented constructs and therefore can be used to adjust existing software to new challenges. While the former one is already well known in the object-oriented world, the latter was only recently introduced. Currently, both techniques co-exist without affecting each other and therefore concrete software projects either use the one or the other approach. There are some situations where the result of utilizing the one or the other is approximately the same. Therefore, it is reasonable to analyze each approach in respect to its underlying philosophy and its impact on the implementation level and to compare them on the basis of those observations. This paper discusses the equivalences and differences between the role concept and aspect-oriented programming and reveals potential synergies between both approaches.

1 Introduction

In the traditional object-oriented literature real-world entities are represented by objects which interact with their environment. Entities in this environment interact with an object by using its intrinsic properties. Nevertheless, as pointed out by Harrison and Ossher in [11] users have different perspectives on an object. In that way the observable properties are not only objective, intrinsic properties, but also subjective, extrinsic properties which depend on a user's perspective.

The core assumption of the role concept is that there are (extrinsic) properties and behavior of an object which may change during its lifetime. In other words, it is assumed that the original classification of an object may change during life-time. A similar argumentation was done against class-based programming languages, for example Lieberman argues in [17] that people do not think in classes but in prototype object whose "essential properties" represent people's view on a class.

The concept of roles and its relation to object-oriented programming has been already widely discussed (cf. [20], [16], [15], [7]). Nevertheless, popular programming languages like C++ or Java do not support roles as first class entities on the language level. So developers who need to apply the role concept usually use a framework.

In the recent past, aspect-oriented programming (AOP) became more and more popular. One of the major observation in [14] is that there are concerns which cannot be cleanly encapsulated using traditional composition techniques and therefore are somehow tangled with other modules. Aspect-oriented programming is about modularizing such concerns, which are called *aspects*. [3] introduces AOP as "the idea that computer systems are better programmed by specifying the various concerns (…) of a

system and some description of their relationships". The underlying aspect-oriented environment is responsible for assembling those concerns together. The result of such a composition is that there are numerous (object-oriented) building blocks stemming from different concerns spread all over the object hierarchy. Hence, an object's member and the classification of objects are not only determined by the corresponding class definition, but also by all aspects which influence the class definition after the (aspect-oriented) composition.

The equivalences between the role concept and aspect-oriented programming are obvious. Both approaches soften the strict restrictions of static typed, class-based programming languages, since the association of class, members and behavior is not completely determined at class-definition time. If such a characteristic is needed in a class-based programming language, developers have to determine which approach serves better the needs at hand and what implementation techniques are the most adequate for the given problem. Currently, there is no known combination of both approaches so they can just be used mutually exclusive. For deciding what technique to use it is necessary to analyze the intention of both approaches, their equivalences, trade-offs and their impact on the resulting code.

In this paper we discuss the similarities and differences between the role concept and aspect-oriented programming. We introduce both concepts in section 2 and 3. Afterwards we discuss both approaches with respect to their similarities, differences and potential synergies. In section 5 we propose a software framework to support the role concept. In section 6 we discuss the result of applying this framework and aspect-oriented programming simultaneously. Finally, we summarize the paper.

2 The Role Concept

Roles are temporary views on an object. A role's properties can be regarded as subjective, extrinsic properties of the object the role is assigned to. During its lifetime an object is able to adopt and abandon roles. Thus, an object's environment can access not only the object's intrinsic, but also its extrinsic properties. In [15] and [16] Kristensen formulates some characteristics of roles:

- *Identity*: An object and its actual role can be manipulated and viewed as one entity
- *Dynamicity*: Roles can be replaced during an object's lifetime
- *Dependency*: Roles only exists together with its corresponding object
- *Extension only*: A role can only add further properties to the original object, but not remove any
- *Multiplicity*: An object can have more than one instance of the same role at the same time
- *Abstractivity*: Roles are classified and organized in hierarchies

In [7] Gottlob et al. emphasize another characteristic of roles:

- *Behavior: A role may change an object's behavior.*

The feature *abstractivity* emphasizes that roles are well-planed and organized in hierarchies similar to object-oriented ones. On the other hand, this characteristic insinuates that the role concept is highly connected to class-based programming languages and hence roles are classified by classes. Nevertheless, it should be emphasized that role concepts can also be used in class-less object-oriented programming languages.

An important characteristic of roles is that roles are dynamically added to objects whereas a role itself has properties (fields and methods). Hence, the accessible properties of a single object differ from perspective to perspective and from time to time. The *root object* describes the intrinsic object, i.e. the original object without any roles. A *role object* is the instance of a role which is added to a certain root object. A *role* is a generalization of its roles similar to classes. For reason of simplification we use the term role instead of role object except in situations where it is necessary to stress the difference. A *subject* is a special perspective on a root object including (some of) its roles. A root object has several subjects whereby every subject contains a different set of included roles. The interface of a subject is an aggregate consisting of the root object's interface plus every role object's interface.

One interesting property of roles (in comparison to aspect-oriented programming) is the behavior characteristic. A role when added to an object may change the object's behavior. While [7] describes this as an intrinsic role feature, [15] and [16] regard it as a special kind of role which they call a *method role*. A method role is a role's method which is bound to an intrinsic method of the root object. It is important to emphasize for later examinations that the cardinality between intrinsic method and method role is 1:n, i.e. every intrinsic method may have several method roles, but every method role has exactly one intrinsic method.

If and how a method role changes an object's behavior depends on what kind of method role it is. There are method roles, which alter a root object's behavior because the object's user is aware of the role, i.e. the role containing the method role is part of a subject used by the user. In that case we call the role method to be *subjective*. In the other case there are method roles which replace the root object's behavior independently of the user's perspective. Although such a behavior is not part of the root object's intrinsic behavior it is independent of a user's perspective. We call such method roles *non-subjective*.

It is obvious that there are several conflicting situations, since more then one role can be assigned to an object. Whenever an object's intrinsic methods are invoked the underlying environment has to determine if and how the corresponding roles influence the resulting behavior. The following conflict situations occur:

- *multiple subjective method roles*: there is more than one subjective method role assigned to the invoked method.
- *multiple non-subjective method roles*: there is more than one non-subjective method role assigned to the invoked method.
- *mixed method roles*: there are at least one subjective and one non-subjective method role assigned to the invoked method.

Furthermore, there is a conflict if there are at least two members from different roles with the same selector within the same subject. If an object uses such a selector to access a member it has to be determined what member to choose.

Figure 1 illustrates a person that has two jobs in parallel as a bartender. A job is a temporal role, because persons usually do not keep their job for the whole lifetime. The person has some properties like name and day of birth which are not influenced by any role. On the other hand there are the properties phone number and income. The phone number is on the one hand an intrinsic property, because it describes a person's private phone number. On the other hand it is an extrinsic property, because it de-

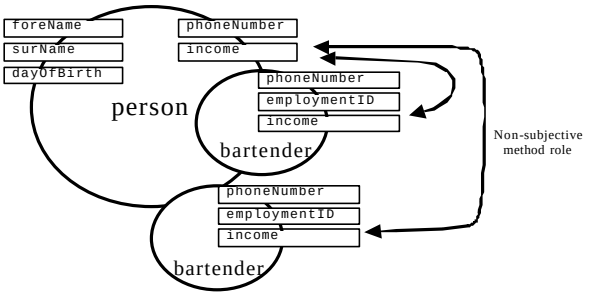

Fig. 1. Object person with two roles bartender

scribes a phone number specific to the bartender role (and contains a pub's phone number). If the phone number property is realized as a method, then the bartender's phone number methods are subjective, since if someone asks a person in private for his number he expects to get the private number, but if he asks a bartender for his number he expects to get the bar's number. On the other hand, a person's income directly depends on the income at his jobs. So the income methods of both bartender roles are non-subjective roles methods.

Although the benefit of role concepts has been accepted widely, popular object-oriented programming languages like C++ or Java do not support roles as first class entities on language level. The reason for it is quite simple: the underlying assumption for static typed, class-based programming languages is that an object's properties are entirely known at compile-time and can therefore be classified. Hence, class-based programming languages do not distinguish between intrinsic and extrinsic properties ([17] discusses this topic in detail). Therefore additional techniques are needed to support roles in class-based languages.

3 Aspect-Oriented Programming

In [3] aspect-oriented programming (AOP, [14]) is introduced as "the idea that computer systems are better programmed by specifying the various concerns (…) of a system and some description of their relationships". The underlying aspect-oriented environment is responsible for composing those concerns. The aspect-oriented term for such a composition is *weaving*. The composition consists of a transformation of all influenced building blocks at certain points specified by the developer which are called *join points*. They represent input parameters for the weaver.

The major observation in [14] is that there are concerns which cannot be cleanly encapsulated using traditional composition techniques and therefore the resulting code is tangled with other modules. So aspect-oriented programming is about modularizing such concerns, called *aspects* which cannot be cleanly separated by traditional composition techniques. A typical example of an aspect is *synchronization* that has no

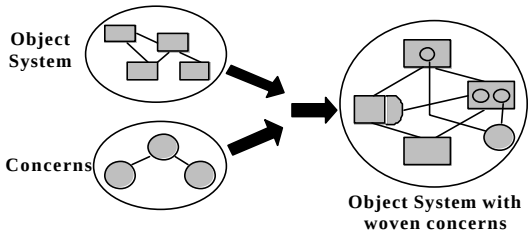

Fig. 2. Weaving Concerns into a software system

satisfactory pure object-oriented solution (cf. [18]).

Figure 2 illustrates the weaving process. There are different concerns and an object system defined in separated modules. The weaver is responsible for combining those concerns with the object system. How each concern is represented in the final woven system depends on the weaver.

Although there are already numerous works on AOP, there is until now no common agreement about what the core ingredients of aspect-oriented programming are, i.e. there is no agreement on what kind of composition mechanisms are necessary for a technique to be called aspect-oriented. In [4] Filman proposes *quantification* to be a major idea of AOP and describes quantification as "the idea that one can write unitary and separate statements that have effect in many, non-local places in a programming system". However, Filman does not propose how such a characteristic impacts the underlying programming languages. So the current situation is that different aspect-oriented techniques provide different mechanisms to achieve such a quantification and/or different kinds of quantification (cf. [10]). Nevertheless, there are already different techniques available which are generally accepted to be aspect-oriented.

The most popular ones are *AspectJ* [1] and *HyperJ* [8]. In the following we briefly discuss the communalities between both to work out the core ingredients of AOP. Afterwards we discuss the impact of different kinds of weaving.

3.1 AspectJ and HyperJ

AspectJ [1] is currently the most popular general purpose aspect language built on top of the programming language Java and offers additional composition mechanisms to modularize cross-cutting concerns. It supports *aspects* as first class entities that permit to define cross-cutting code. Aspects contain definitions of join points which are used by the weaver to change the behavior of objects or to change class definitions. Changing the behavior of objects is achieved by a method-like language construct called *advice* which specifies the new behavior. One advice can be connected to several different join points which may be spread all over the object structure. Moreover, several advices from different aspects can be woven to one join point. There are different kinds of advices like *before*, *after* or *around* advices. They specify when the new behavior is meant to take place in relation to the corresponding join points.

HyperJ [8] developed at IBM alphaworks is an offspring of subject-oriented programming (SOP, [11]) and is generally accepted to be an aspect-oriented technique. In contrast to AspectJ, HyperJ does not extend the programming language. So aspects are not supported as first class entities. Instead, HyperJ is a tool for weaving Java classes, whereas the weaving instructions are not defined in the building blocks which are about to be combined, but in separate configuration files. Like AspectJ it is possible to add fields or methods to classes and to change the behavior of objects.

Another equivalence between HyperJ and AspectJ is that both permit to change an object's behavior depending on its context: the behavior of an object may depend on the client who sends a message to the object (AspectJ even permits to define behavior for certain control flows). Furthermore, both approaches have in common they allow to group join points based on lexical similarities. A typical example is the grouping of all method calls where the method selector begins with the tokens "`set`".

3.2 Static and Dynamic Weaving

Above we introduced weaving as a mechanism for composing separate defined concerns into a software system. The underlying system is responsible for weaving the concerns. Nevertheless, the question is when concerns are to be woven to the system. Weaving may either occur *before* or *at* runtime. The first case is usually called *static weaving*, the latter one *dynamic weaving*. The point in time when static weaving occurs may correspond to the compile time of the concerns (as implemented in AspectJ), or it is after compile time and before runtime (HyperJ). If weaving occurs during runtime, concerns can be woven and unwoven depending on the systems state. Moreover, there are *load-time approaches*, like *Binary Component Adaption* (BCA, [12]) which utilize the Java-specific class-loading for transforming the concerns to be woven and the classes they affect. Load-time approaches are a special kind of dynamic weaving since the transformations are done during runtime.

The underlying weaving mechanism has a direct impact on the kind of quantification that can be supported. Static weaving permits to use all kinds of static information (type information, syntax tree, etc.) while dynamic approaches only use state information. An aspect that appears only at runtime cannot influence the whole system since parts of it are already executed without the aspect's influence. On the other hand dynamic weaving reduces the preplanning restrictions: instead of determining already at compile time what aspects appear in the system, this can also be achieved at runtime.

3.3 Characteristics of AOP

Based upon the observations above we can extract the following characteristics of aspect-oriented programming:

- *Aspect Proclamation*: Aspects arise by declaring them, i.e. the underlying environment is responsible at weaving time for identifying the objects influenced by the aspects and generating the new woven objects.
- *Context dependence*: Aspects allow to change objects' behavior depending on a certain context. E.g. HyperJ and AspectJ permit to define an object's behavior depending on the caller.
- *Split aspects*: A single aspect may influence several objects. An aspect may touch every part of an object structure at weaving time.
- *Cardinality between method and advice*: AspectJ and HyperJ permit a cardinality of `n:m` between the original methods and the added behavior. I.e. for every method there may be several advices and every advice may be added to several methods.

It is emphasized by numerous authors that aspect-oriented programming is not just restricted to object-oriented programming, but may also be applied to other paradigms. Nevertheless, almost all known approaches are built on object-oriented languages. Assuming an underlying object-oriented language, the characteristics above show that aspect-oriented programming represents an extension to object-oriented programming. In the traditional object-oriented literature it is accepted that "an object may learn from experience. Its reaction to an operation is determined by its invocation history" [22]. In aspect-oriented programming an object's behavior is additionally determined by its invocation context and the existence of other concerns.

4 Comparing Aspects and Roles

In the previous sections we have seen that both concepts permit to adapt the behavior and structure of objects. Here, we compare both approaches based on the above mentioned characteristics.

First of all we analyze in what way aspects match the characteristics of roles.

- *Identity:* Aspects do not have to be instantiated for each object they are woven to. This is done by the underlying environment. Furthermore, a single aspect may influence numerous objects (split aspect) which means that an aspect and the objects it is woven to do not form one single entity/unit.
- *Dynamicity:* The question whether aspects can be added dynamically depends on the underlying aspect-oriented system. Dynamic weavers permit it while static weavers do not. Therefore dynamicity is not a mandatory characteristic of aspects.
- *Dependency*: Aspects do not exist on their own. Instead they depend on the object-oriented structure they are woven to. Hence, aspects have this characteristic.
- *Extension only:* Based on the above introduction of AOP the answer needs to be: yes, like roles aspects are extension only. On the other hand, systems like AspectJ permit to declare restrictions on the object-oriented structure. It is possible to e.g. declare that "a class `A` must not have a method `B`. Otherwise class `A` will not be compiled". This means that aspects are not extension only. However, up to now there is no common agreement on whether this is an essential aspect-oriented feature or not. Hence, it cannot be finally decided whether aspects meet this characteristic.
- *Multiplicity*: From the technical point of view there is no reason why an aspect may not be applied to the same object twice. Nevertheless, the major focus of AOP is to weave different concerns at the same time into a system. Usually a single concern is not applied to an object or class for more than one time. This implies that multiplicity is not a characteristic of AOP.
- *Abstractivity*: Like roles, aspects can be organized in hierarchies. In AspectJ aspects are treated like classes. In HyperJ it is possible to define dependencies between the configuration files. So, aspects meet this characteristic.
- *Behavior*: Aspects and roles can change the behavior of the structure they are woven to. While a role may change the behavior of single objects using method roles, aspects may change the behavior of larger units (collection of objects). Usually aspects are woven to classes and not to single objects.

On the other hand we have to check if roles share some properties of aspects:

- *Aspect Proclamation*: Roles are always assigned to objects. Therefore, aspect proclamation is not supported by roles.
- *Context dependence*: Roles do not permit to change an object's behavior depending on the context. Either a method role is added to a root object or not. E.g. a method role does not vary its behavior in dependence of the clients sending a message to the root object.
- *Cardinality between method and advice*: As already mentioned, the cardinality between method and its roles is (usually) 1:n. AspectJ and HyperJ permit a cardinality of `n:m` between the original methods and the added behavior. However, it is

possible to implement the role concept in a way that is supports this n:m relationship.

- *Split aspect*: A role instance can only influence the behavior of the object it is assigned to. A role cannot be split so that each subpart influences a different object.

The overall conclusion is that aspects (especially when based on dynamic weaving) match almost every characteristic of roles while roles do not match the characteristics of aspects. Nevertheless, it should be emphasized that current aspect-oriented approaches (like AspectJ or HyperJ) provide only static weaving and, therefore, do not support dynamicity. But dynamicity is a very important characteristic of roles.

The above discussion clearly indicates that developers currently have to decide whether they need dynamicity. In case they do they cannot use current aspect-oriented techniques. If developers need to exploit context dependent object behavior they cannot use roles. Moreover, if developers want to declare concerns in their system, i.e. want to adapt numerous classes and objects without the additional effort of identifying and modifying the sources to be changed, they need the characteristic of aspect proclamation which is not supported by the role concept.

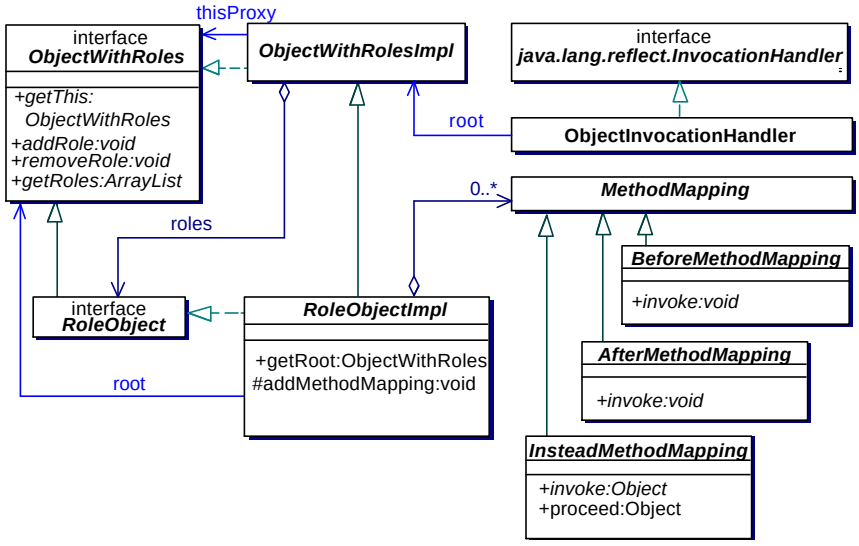

Fig. 3. Framework to Support Roles in Java

5 Implementing Roles in Java

There are mainly two different directions for realizing roles. On the one hand there are approaches on the implementation level which directly depend on language specific features. E.g. Gottlob et al propose in [7] a Smalltalk implementation, which is based on the Smalltalk specific feature of handling incoming messages on the meta level. [15] discusses implementations in BETA by extending the compiler. In [21] VanHilst and Notkin use C++ class templates to implement roles and Neumann and Zdun propose in [19] *per-object-mixins* which use message interception techniques. [13] pro-

poses an aspect-oriented implementation of the role concept but neglects the impact of static weaving for the characteristic of dynamicity.

On the other hand, there are approaches for supporting roles at design time. E.g. Fowler discusses in [5] different ways for designing roles based on some design patterns [6]. The usual argument against the latter approach is the *preplanning problem*, i.e. "the use of the patterns for extension must be done in advance of an actual need to exploit its flexibility for a particular piece of software" [9]. For realizing roles this means that the designer has to decide what type of object may realize what kind of roles. On the other hand this is a limitation to the dynamicity characteristic, since at design time is must be determined which objects may have what roles (if any) at runtime.

The approaches above cannot be applied to Java, since Java does not provide the necessary features: It does neither permit to alter the implementation of meta classes nor to extend the compiler. Likewise, Java does not support class templates. Although extensions like *generic java* (see [2]) permit the usage of generic types until now this is not part of standard java. Furthermore, Java does not provide any mechanisms for mixins.

Since version 1.3 Java contains a mechanism called *dynamic proxies*. A dynamic proxy is an object, whose interfaces are specified during runtime. An invocation handler permits to reflect and redirect incoming messages. Although the main contribution of dynamic proxies is that objects can be created whose interfaces are specified during runtime, for realizing roles the ability to intercept message is much more important. In the following we propose a framework to address the role concept and discuss its structural elements. Implementation details are out of the scope of this paper.

Figure 3 illustrates a framework to support roles in Java based on dynamic proxies. Each class is divided into interface and implementation. Each interface extends the root interface `ObjectWithRoles` which contains methods for attaching and detaching role objects. Each implementation extends the root class `ObjectWithRolesImpl`. The constructor of `ObjectWith-RolesImpl` creates an invocation handler (instance of `ObjectInvocationHandler`) and registers the new created instance at the handler (attribute `root`). Furthermore, the constructor creates a new proxy object initialized with the invocation handler (`thisProxy`). Clients creating a new instance only use the proxy instance for further work.

Each class, which represents a role must also be split into interface and implementation and extends the interface `RoleObject` respectively `RoleObjectImpl`. The constructor of `RoleObjectImpl` has an `ObjectWithRoles` parameter which represents the root object. Furthermore, the framework contains *method mappings*. A method mapping is an object which determines which methods of the root object are influenced by which method roles. The framework supports three kinds of role methods: *before*, *after* and *instead* mappings. A before method mapping allows method roles to be executed before the original invocation takes place, after and instead mapping behave correspondingly. Instead method mappings differ slightly from the other ones: they have a method `proceed()` which allows the root method to be invoked.

The invocation handler receives messages sent to the root object and analyzes if there are attached roles that contain method mappings matching the called method. Afterwards the invocation handler invokes the method role with the highest priority

(in our implementation the role added at last has the highest priority). Details about how the handler works are out of scope of this paper.

Figure 4 shows how an income method role of the type mentioned in section 2 is implemented: the income of a person is calculated by adding the income of all roles. The method `proceed()` returns the value of the next method role (either the next method role registered to the same method or the target method itself). The parameter `ic` contains some context information necessary for method roles. The constructor of the method mapping contains parameters to determine to what entity the method role is registered. The parameters in figure 4 determine that the method role is registered to a method `getIncome()` without any parameters.

```
public class BartenderImpl extends
       RoleObjectImpl implements Bartender {
  ...
  {
    addMethodMapping(
    new InsteadMethodMapping ("getIncome", new Class[0]) {
      public Object invoke(InvocationContext ic)
             throws Throwable {
        return getIncome()+ proceed(ic);
      }
    });
  }

  float income;
  public float getIncome() { return income; }
  ...
}
```

Fig. 4. Implementation of an income method role

6 Collaboration with AspectJ

In the previous section we introduced a framework to support the role concept. Here we discuss how the proposed framework collaborates with aspect-oriented programming by using AspectJ[1] as the most popular general purpose aspect language. The proposed framework fulfills the characteristics of the role concept. Especially the characteristic of dynamicity which is not provided by AspectJ is supported.

Since AspectJ is an extension of Java it seems to be reasonable to use it in addition to the framework. Nevertheless, for the following reasons there are some difficulties: AspectJ uses the compilation unit's syntax trees for weaving. The framework (or more precisely: the invocation handlers) on the other hand uses reflection to redirect incoming messages which are not transformed in AspectJ. The consequences of this are not that obvious:

- *Double advice invocation*: Each context independent advice, i.e. each advice that adapts an object's behavior independently of the calling object, is invoked twice. The reason for this is that AspectJ tests the type of the target object for each redirected call in the woven code. If the class does not match the place where the advice is woven to the advice (more precisely: the advice's Java representation) is

[1] The observations in this paper are based on AspectJ, v 1.0.3

invoked directly. However, in the proposed framework the target object is always a dynamic proxy. So AspectJ invokes the advice twice.

- *Inelegant weaving in method roles*: For mainly two reasons there is no elegant solution for weaving advices to method roles: in the framework the method mapping classes are abstract and it turned out to be a good idiom to use anonymous classes for registering method roles (see figure 4). Since AspectJ uses lexical similarities to identify join points, these classes can hardly be identified.
- *No context dependent behavior in method roles*: The static weaver in AspectJ cannot accomplish call dependent weavings in method roles, since the root object's invocation handler is responsible for invoking method roles and reflective calls are not transformed by the weaver.
- *Non-natural parameter passing*: parameters which are related to the calling object or the target object are always instances of implementations. Nevertheless, the framework assumes clients to work on the proxy instances. Hence, advices always need to execute the `thisProxy` instance for each passed parameter. Therefore, parameter passing is in a way not natural.

It should be mentioned that technical solutions exist for all above mentioned problems. Nevertheless, the developer has to be aware of these problems, because they influence the usage of both, the underlying framework for roles and AspectJ.

7 Conclusion

In this paper we discussed the similarities and differences between the role concept, introduced in section 2, and aspect-oriented programming. In section 3 we elaborated some aspect-oriented characteristics based on AspectJ and HyperJ. Moreover, we discussed the impact of different weaving techniques. Afterwards we compared both approaches. In section 5 we proposed a software framework for the support of the role concept. Section 6 discussed the consequences of applying the proposed framework and AspectJ simultaneously.

The paper provides two important contributions. First, we showed that there is a difference between aspects and roles. This conclusion is quite interesting, since both approaches are about object adaptation and there are numerous identical coding examples which claim to be typical applications for only one approach (e.g. an implementation of the observer pattern). Moreover our comparison showed that there is a difference between dynamic weaving and the role concept. Second, we discussed the consequences of using roles and aspects at the same time by introducing a framework based on AspectJ that supports roles. We showed that this leads to undesired results and restricts the usage of both approaches.

Both, roles and aspects offer valuable mechanisms for adapting software systems which have some characteristics in common. Nevertheless, it has to be considered that an integration of both techniques requires more effort than a first glimpse may pretend.

References

[1] AspectJ Team, The AspectJ Programming Guide, http://aspectj.org/doc/dist/progguide/

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler, Making the future safe for the past: Adding Genericity to the Java Programming Language, OOPSLA 98, Vancouver, October 1998.
[3] Tzilla Elrad, Robert E. Filman, Atef Bader, Aspect-oriented programming: Introduction, Communications of the ACM, Volume 44 , Issue 10, October 2001, pp. 29-32
[4] Robert E. Filman, What is Aspect-Oriented Programming, Revised, Workshop on Advanced Separation of Concerns, ECOOP 2001
[5] Fowler, M., Dealing with Roles, Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, September 2-5, Washington University Technical Report 97-34, 1997.
[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
[7] Georg Gottlob, Michael Schrefl, Brigitte Röck, Extending Object-Oriented Systems with Roles, ACM Transactions on Information Systems, Vol. 14, No. 3, July 1996.
[8] IBM alphaworks, HyperJ Homepage, http://www.alphaworks.ibm.com/tech/hyperj, last access: February 2001
[9] IBM Research, Subject-Oriented Programming and Design Patterns, http://www.research.ibm.com/sop/sopcpats.htm, last access: January 2001
[10] Hanenberg, S., Unland, R.: A Proposal For Classifying Tangled Code, Workshop Aspekt-Orientierung der GI-Fachgruppe 2.1.9, Bonn, February, 2002
[11] William Harrison, Harold Ossher, Subject-Oriented Programming (A Critique of Pure Objects), Andreas Paepcke (Ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), SIGPLAN Notices 28(10), October 1993.
[12] Keller, R., Hölzle, U., Binary Component Adaption, ECOOP 1998, LNCS, 1445, 1998, pp. 307-329.
[13] Elizabeth A. Kendall, Role Model Designs and Implementations with Aspect-Oriented Programming, Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99), SIGPLAN Notices 34 (10), 353-369.
[14] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J., Aspect-Oriented Programming. Proceedings of ECOOP '97, LNCS 1241, Springer-Verlag, pp. 220-242.
[15] Bent Bruun Kristensen, Object-Oriented Modeling with Roles, Proceedings of the 2nd International Conference on Object-Oriented Information Systems (OOIS'95), Dublin, Ireland, 1995.
[16] Bent Bruun Kristensen, Kasper Østerbye, Roles: Conceptual Abstraction Theory & Practical Language Issues". Theory and Practice of Object Systems, Vol. 2, No. 3, pp. 143-160, 1996.
[17] Lieberman, Henry, Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, Proceedings of OOPSLA 1986, SIGPLAN Notices 21(11) pp. 214-223
[18] Matsuoka, S., Yonezawa A., Analysis of Inheritance Anomalies In Object-Oriented Concurrent Programming Languages, In: Agha, G., Wegner, P., Yonezawa, A. (eds.), Research Directions in Concurrent Object-Oriented Programming Languages, MIT-Press, 1993, pp. 107-150´
[19] Gustav Neumann and Uwe Zdun. Enhancing object-based system composition through per-object mixins. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), Takamatsu, Japan, December 1999.
[20] Pernici, Objects with Roles, Proceedings of OOIS, 1990
[21] VanHilst, M., D. Notkin, "Using Role Components to Implement Collaboration- Based Designs," Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'96), ACM Press, 1996, pp. 359 - 369.
[22] Peter Wegner. The Object-Oriented Classification Paradigm. In: Bruce Shriver and Peter Wegner (eds.), Research Directions in Object-Oriented Programming, MIT Press, 1987, pp 479-560