

Aspect modelling as Role modelling

Kasper B. Graversen
IT University of Copenhagen
kbilsted@it-c.dk
(corresponding author)

Kasper Østerbye
IT University of Copenhagen
kasper@it-c.dk

Abstract

AOP languages tend to be focused on the solution domain. Focusing on the solution domain rather than the problem domain inhibits the strength of AOP in early phases of design and modelling. By using roles to model aspects, modelling is conceptually supported by the programming language, and focus is kept in the problem domain.

The paper explains modelling using roles, then it shows that modelling aspects with roles is just a subset of role modelling. Finally, new properties of aspects are shown, when aspects are modelled with roles, e.g. that an aspect can be applied a perspective of an object rather than the object itself. Along the way a pictorial representation is presented.

Keywords: Aspect-oriented design & modelling.

1 Introduction

AOP is based on a fundamental paradox: *The greatest advantage of AOP is that at any point in the source code it is impossible to see all the code, which is to be executed. At the same time the greatest disadvantage of AOP is, that at any point in the source code it is impossible to see all the code, which is to be executed.* To encounter this fact, strong emphasis must be put on having a clear model which corresponds well with the world to be modelled, and secondly and of equal importance, there must be a strong correspondence between the model and the actual implementation.

In this paper we propose a modelling and implementation concept called ‘roles’. Roles is an old concept but has not yet gained the attention it deserves. We extend the role concept and show that roles can model aspects, and that when aspects are modelled with roles, new properties of aspects arise.

2 Modelling with roles

Having roles as first class entities yields new possibilities when modelling compared to using ordinary object-oriented modelling. The motivation for roles is to be able to model and apply perspectives on to objects which models phenomenon. Such perspectives are used by other objects to access the object in a specialized form—as a special way of knowing and using the object. Four important traits of the perspectives are

- They can change dynamically (removed and attached).

- One perspective does not change the behaviour of other (unrelated) perspectives applied the same object.
- Roles are instantiated, referenced and used in the same manner as objects are.
- A role has the same types as the object it is a role for, hence it can be used in the same contexts the object can be used.

We call such perspectives ‘roles’. The role can introduce new properties (fields and methods) to an object, as well as overriding the behaviour of existing methods. The behaviour of roles corresponds to how we as humans think, express and organize our world—“we think in roles” [7]. We can, however, not easily obtain the behaviour of roles using normal class-based languages. Parts of the dynamic behaviour can be simulated with the “Decorator” design pattern, but its usage blurs the code, makes the code rigid (changes are more difficult as they must be made several places). And making further abstractions on top of it is difficult and error-prone (further discussed in [5, p. 17–23]). In short we define roles as *object-based inheritance with multiple views*.

Take a person Bert. He is modelled by a name and a birth date. Let Bert’s main interest be playing golf, hence he is associated with a golf club and has several golf course records of ‘personal best scores’. The golf club has a lot of members, of whom a limited number is on the board and one is the chairman. The golf club owns several lanes. Members must pay both an annual fee as well as for each time they want to use the lanes (which again vary in size, price and quality). Another side of Bert’s life is his financial situation, although limited, he has one nevertheless—Bert is associated with a bank, on which he can perform operations.

When modelling Bert’s world using roles, each property is evaluated as either *intrinsic* or *extrinsic*. Intrinsic properties are properties which the object cannot exist without, whereas extrinsic properties are properties only needed for certain situations. Notice that the intrinsic/extrinsic distinction is dependant on the overall purpose of the system—nothing is intrinsically intrinsic so to say. In our case we model Bert as a person containing a name and birth date. The rest of the properties we have described, we define extrinsic. The extrinsic properties are placed in roles rather than classes. Since the banking business and golfing are unrelated in our model, they are decomposed into two roles, as depicted on figure 1. If needed,

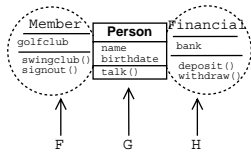


Figure 1: A person object with two roles, a golfer and a financial role. F, G, H are 3 distinct views of the object.

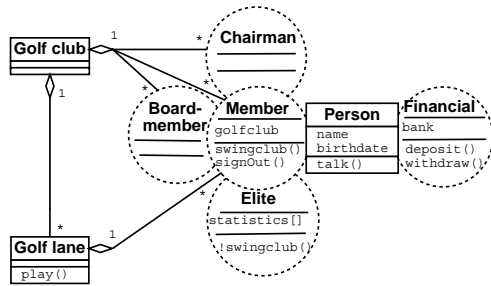


Figure 2: Persons of a system associated with either a golf club, a bank or both. Members of a golf club can play a variety of roles.

further decomposition can be accomplished by creating roles for the golfer and financial role. In fact all abstraction mechanisms available for objects are equally available for roles (such as aggregation and inheritance). On the figure the entities F, G, H represent multiple (distinct) views on the person object. And since the properties are placed in roles, they can dynamically be attached and detached person objects. This means, that not all persons in the system needs not be golfers, nor necessarily have an association with a bank or know how to use it.

We now extend the example to also include a golf club. As the golf club uses its members for a variety of things, there are many roles applied the ‘member role’ of the person and the multiple view of objects are naturally used (see figure 2). Members can either be just members, or be on the board or be the chairman. Finally for the elite members there are statistics associated with them. The golf club owns several golf lanes, these are independent entities which are modelled by a class.

Using roles, complex objects such as Person, can be split up in different complementary views. Further, objects which are needed for different purposes can similarly benefit from such separations.

3 Extending the role concept

In many role models and delegation systems [1, 2, 4, 6], roles are used only to dynamically extend objects at runtime. Fundamentally there are two problems with such role concepts which inhibits it from being used for aspect-oriented modelling: Roles must be attached the object at every object creation in the program, and more importantly, an object is only extended by the role when the object is being referenced through the role rather than being referenced directly (due to

the multiple view property).

We shall address both problems and show that, not only can we model aspects but the nice properties of modelling with roles are preserved. The two role extensions we present are: “constituent methods” and “life roles” (introduced under another name and shape in [8] and [5] respectively).

Constituent methods: Are special methods residing in roles, which hooks onto methods of the roles’ intrinsic. The constituent methods cannot be invoked—they are executed only when the method they hook on to is invoked. The execution takes place before, after, or instead of the original method—with the additional possibility of changing input arguments or return values.

Constituent methods radically change how roles work. Prior to constituent methods, a role’s functionality was only executed, when methods were invoked on the role. With constituent methods it is easy to functionally decompose a single object by the use of roles.

Constituent methods are graphically depicted as methods with a preceding ! as on figure 2 where the swingClub() is a constituent method in the Elite role.

Life roles: Life role is a role modifier (in a similar fashion as “private” is a modifier to a class). Specifying a role to be a life role has the effect that whenever an object is created which is the roles intrinsic, the role is automatically instantiated and attached the object. The second property of life roles is, that the role is attached its intrinsic the whole life of the intrinsic—hence it cannot be removed when first applied.

Life roles are graphically depicted as a solid circle whereas normal roles are depicted with a dashed circle.

Using the two extensions, one can functionally decompose an entity using roles, and have the roles be automatically attached. Prior to the introduction to constituent methods, roles did not really represent separations of concerns, as each concern (the roles) were explicitly used, instantiated, referenced etc., hence roles was no more than an “advanced form of aggregation”.

4 Roles as aspects

A role cannot define an aspect, as an aspect is defined as “a cross-cutting concern”, and hence involves multiple entities. A set of roles in unity, on the other hand, can represent an aspect. If an aspect, requires the code chunks x, y, z to be inserted in the entities A, B, C in their method $f_{OO}()$, then this is done by creating a life role for each entity, each containing a constituent method with the code x, y or z which hooks onto the method $f_{OO}()$ of its intrinsic. Much in a similar fashion to AspectJ, where the three constituent methods would be called “advices” and be defined within an explicit block called an “aspect”. This is illustrated on figure 3.

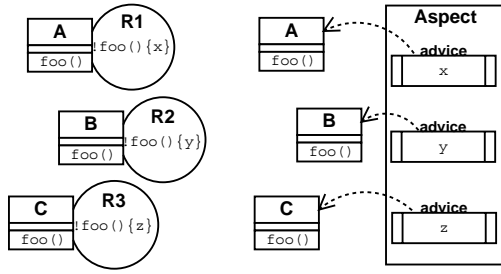


Figure 3: On the left an aspect is made up by three roles R_1 , R_2 , R_3 . The equivalent situation is depicted on the right using AspectJ.

We have here a modelling mechanism, which allows us to clearly separate concerns by the use of roles. More interestingly, when we let aspects be modelled by roles, we can apply an aspect to *a perspective of an object* rather than the object. Naturally such behaviour can be simulated in other languages, e.g. AspectJ, which gives the programmer access to the calling stack. However, such solutions are rather technical compared to our conceptual role model and which when being technical, typically are rooted in the solution domain.

Let us extend our golf club example from figure 2 with aspects. Our first aspect is an economy aspect. Members must pay an annual fee to be a member of the club, and additionally must pay each time they use one of the lanes. We model this aspect by applying an “economic” role to the entities “golf club”, “golf lane” and “member”. The members each have an account balance which when zero results in notifying the member e.g. through email. The golf lane’s economy role ensures that the member’s accounts are adjusted according to the current prices. The golf club’s economy role takes care of managing the fees etc. The second aspect we apply the system is a security aspect, which is applied all board members and the chairman roles of the system. These special members have access to certain administrative functionality of the system which ordinary members are prohibited. The security aspects ensures that a logon is required to access this functionality along with logging facilities which logs all actions so e.g. changing the fee to zero will not go unnoticed. The two aspects are depicted on figure 4.

5 Dynamic aspects

We are able to set up aspects for certain objects in the system (not all members are members of the board), and we are able to specify that they apply only to certain perspectives of these objects. Such functionality is to our knowledge not supported by existing AOP languages, which typically operate on a static basis—that is, a ‘member aspect’ can only be applied all persons, not individuals.

Since perspectives (roles) are dynamically attachable and detachable, so becomes the “aspect roles” applied them (removing the board member role removes the security aspect role as well). But the dynamics can be taken one step further. As

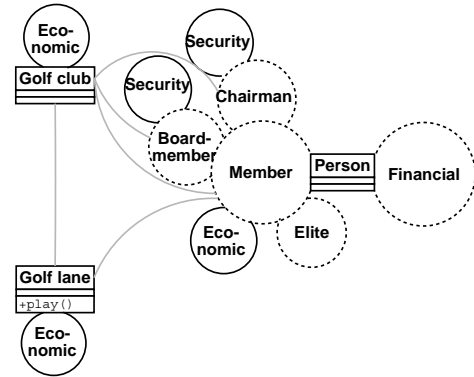


Figure 4: The golf club model from figure 2 extended with an economy and a security aspect. Unimportant entities has been removed or faded.

we model aspects using a set of roles, it is natural to let the properties of roles affect the modelling—in other words, an aspect could very well be dynamic in the sense that it can be removed (and re-attached when needed). An aspect can be removed from certain entities merely by removing the “aspect-roles” from them.

This requires either a change of our role extensions or a new concept similar to life roles, but which allows the role to be detached at run time. Such an extension however, exposes a vulnerability of the model: Letting “autonomous” entities represent an aspect, allows a subset of the roles to be removed, thereby leaving the aspect in a inconsistent state—e.g. when removing role R_1 from the situation illustrated on figure 3 page.

We remedy this problem by gathering roles in groups called “aspect groups”—each containing a set of roles, which together form an aspect for a set of objects. Deactivating a role in the system has the effect, that all roles within the aspect groups the deactivating role resides, are deactivated as well.

Unfortunately such groups must be established and roles be subscribed, which entail some inconvenience (in some ways similar to the problems of modelling AOP without life roles). However, we see the grouping mechanism as a nice concept usable to group “ordinary” roles as well as “aspect roles”, e.g. the grouping can mimic grouping of entities of the world to be modelled, and grouping of roles allows actions to be performed on a group of roles rather than single roles.

6 Discussion

One of the important contributions that roles have to offer to aspect-oriented programming is its foundation in modelling. The traditional distinction between problem domain and solution domain is excellently described in the book on Generative Programming [3]. The problem domain has to do with the real world, e.g. students, courses, grade average, and how to model that world in a manner that is useful for developing applications. The solution domain is the domain of software artefacts,

e.g. source programs, classes, and objects.

Role modelling takes outset in the problem domain, addressing a need for separation of concerns in that domain. The work in [5] builds a bridge from the problem domain oriented role models, to the solution domain, by providing linguistic support for roles. Aspect-oriented programming has its natural outset in structuring problems in the solution domain, separating e.g. logging, transactions, and security concerns (which are the standard examples).

There are two important perspectives to examining aspects for the early phases in software development: One is to find out how to handle solution domain issues. Another perspective is to investigate how aspect-oriented thinking can be applied to the problem domain, rather than the solution domain. The previous sections has revealed that linguistic support for roles, with some extensions, is remarkable close to aspects. The rest of this section will investigate what contributions role modelling can offer to early aspects, aspects in the problem domain.

One of the fundamental properties of roles is that they encapsulate perspectives of an object, which is relevant from the perspective of some other (set of) objects. This allows us to break down large objects into an intrinsic part, and a set of roles. The contents in a role is additional instance variables, methods that manipulate those and/or integrate the role into the intrinsic. This corresponds somewhat to aspects that only introduces instance variables and methods onto objects, though an aspect cannot override a method from one perspective only. In the golfing example, it is worth noticing that roles are not something defined in isolation, the golf member roles are all working in unison with the golf club. The logical package that deals with golfing consists of a number of classes representing the golf club, and some roles for the persons involved in the golf club. As mentioned in section 5, we have felt a need for linguistic and conceptual support for this “package” level of modelling which we called “aspect groups”.

Returning to aspect-oriented programming, it becomes clear that the same problem exist. In AspectJ, an aspect can contain aspects of other classes, but not classes by itself. If we for an instance assume that persons were born golfers, then an aspect cannot describe our example, as it does not allow us to describe the classes of the golf club *and* the advices and introductions necessary to establish the member hierarchy. Thus, our modelling perspective points to a natural extension of aspects.

Another facet of roles is that the roles are first class abstraction mechanisms. It has in the literature been described how one make specializations of roles, how one establishes roles of roles, how roles can be aggregated etc. Such maturity of abstraction does not seem to be present for aspects, e.g. what is an aspect of an aspect? There are some mechanisms for specialization in AspectJ through the use of abstract pointcuts. However, we believe that it is possible with outset in roles to further develop the aspect paradigm to handle aspects of aspects and support a richer notion of composition and specialization.

With regard to the early phases of software development, it will be fruitful to attempt a merge between the modelling capabilities of roles and the grouping capabilities of aspects. However, our own modelling practice show that the aspect-oriented thinking, at least as represented by AspectJ, has so high emphasis on the crosscutting portion, that it misses the obvious, that full classes can also be part of an aspect—the logger class itself is part of the logging aspect. This would lift aspects to a interesting proposal on an abstraction mechanism that goes beyond classes, being not only a scooping mechanism like packages in Java, but a first class abstraction mechanism.

7 Conclusion

On the basis of the paper we conclude, that roles can be used to model aspects. Further the role concept provides new properties to aspects both in terms of dynamics as well as in terms of decomposing the problem domain into manageable units.

References

- [1] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs.*, 1993.
- [2] Wesley W. Chu and Guogen Zhang. Associations and roles in object-oriented modeling. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 257–270, 1997.
- [3] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming, Methods, Tools, and Applications*. Addison Wesley, 2000.
- [4] Georg Gottlob, Michael Schrefl, and Brigitte Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.
- [5] Kasper B. Graversen and Johannes Beyer. Chameleon, August 2002. Masters thesis.
- [6] G. Kniessel. Objects don’t migrate!—perspectives on objects with roles, 1996.
- [7] Bent B. Kristensen. Object-oriented modeling with roles. In John Murphy and Brian Stone, editors, *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer-Verlag, 1996.
- [8] Bent Bruun Kristensen and Kasper Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.