

Role collaborations*

Kasper B. Graversen
IT-University of Copenhagen
kbilsted@itu.dk

19.01.04

Abstract

We present our latest preliminary work on collaborating entities, not sequentially, but by taking outset in the discussions of the topics comprehensibility, predictability, evolvability, and semantic interactions. Essentially, collaborations are just classes containing inner roles. This combined with around-methods (advices) and slightly modified delegation scheme provides a flexible and interesting language construct. Our implementation is capable of supporting specialization of advices, making advices for advices, aspectual polymorphism along with all the advantages of having roles as a first class construct. Comparisons with other collaboration-based models are further carried out.

1 Solution name

A descriptive name: “a role model augmented with AOP capabilities, inside scopes with virtual inner classes”, however “role collaborations” is probably a better name.

2 Problem addressed

The problem addressed here will only be sketched. Our motivating example is a patient in a hospital: How to represent a hospitalization, the wards round, the administration of drugs etc. Should we have a Doctor class with a heal method taking a person as an argument?

3 Description

This papers topic is the representation of collaborating entities. The backbone of our collaborations is our role implementation from our past research. We thus give a recap of what roles are about.

3.1 Role recap

Basically, a role object is equivalent to an ordinary wrapper object, with the three exceptions, (1) it is a subtype of what it is wrapping (henceforth know as its *intrinsic*). (2) The role cannot live independently, it always has to be attached some

intrinsic. (3) messages are delegated from the role to its intrinsic. The last point is equivalent to super calls in standard OO languages; hence method invocations are late bound (under the assumption that methods are virtual). Generally, roles can be perceived as run-time inheritance, not necessarily applying all objects of a given type. Different from static inheritance is *multiple view*: One can choose to view an object through any one of its roles or not. Several roles can be applied an object or a roles, but each role has only one intrinsic. Roles can be removed or moved around (doing this safely, however, is a difficult and so far expensive operations). Roles typically require manual attachment, since they take place on the object level. However mechanisms such as the *class role* [7, p. 28] has been invented to automate role attachment—this at the price that roles are attached all objects of the given type. From a research point of view, many issues arisen within the field of inheritance are issues in role languages as well.

The notion of classes focuses on the *capabilities* of objects, while the notion of roles focuses on the *position and responsibility* of an object within the overall structure of objects [20, p. 65]. Our case with the hospitalizations is shown on figure 1.

Roles are advantageous in that role playing is a natural part of our understanding of the real world; hence the gap between design and implementation is reduced. Roles allows entities, in an easy manner, to participate in several places of the system and still maintain a nice separation of concern on the code level. Finally, since roles encapsulate fewer decisions than classes, they are more stable with respect to evolution [23, p. 1].

For information on roles modelling perspectives are e.g. [13, 14, 20]. [1, 2, 6, 7, 10] are also easily read and takes discussions relating to the actual code.

The AOP augmentation of our role model is not essential to this paper, hence only a short description is given. Around methods (also called advices here) can be defined along with pointcuts. Pointcuts are evaluated at call time (before the call message is sent), and only pointcuts on roles for the receiver of the message (or roles for those roles, etc.) are evaluated. Hence, the around methods are only active, when the role in which they reside is attached some intrinsic (object or role instance). For reasons of simplicity, the current implementation poll the pointcut constantly, rather than inferring the places they could evaluate to true.

*This is version 1 for the SPLAT workshop at AOSD 2004.

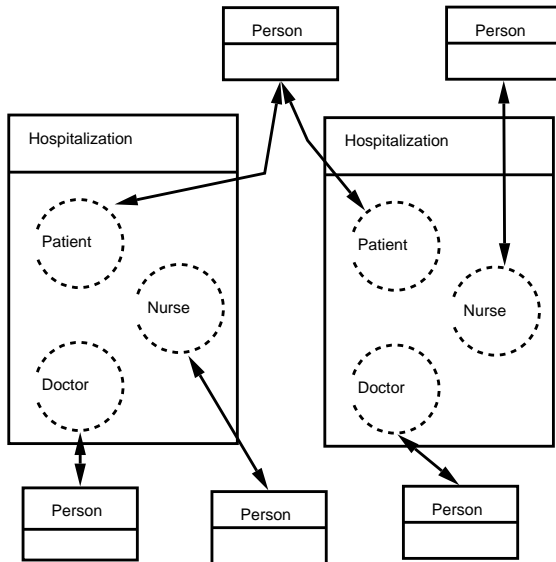


Figure 1: A picture of two collaboration instances (*Hospitalization*). The dotted semi-circles are role instances. The small boxes are objects. Notice there is no difference between objects and collaborations.

3.2 Description of ‘role collaborations’

We present a “traditional role model”, which has been augmented with simple pointcuts and advices known from the AOP domain. This model is an evolution of the Chameleon role model presented in [7], which to the our knowledge is the only role model which has such an augmentation. The focus in this paper is not on the AOP augmentation, but on the particular use setting of roles:

1. Placing roles inside a scope naming this a collaboration¹.
2. Methods and inner classes (and roles) are virtual².
3. Utilizing multiple inheritance to compose units of separated methods, pointcuts and advices³.

Others has attacked the issue from the opposite angle, in that the collaborating participants are not enclosed in a scope, see e.g. Lasagne/J [9], C++ templates Roles [23], “mediator” design pattern [5, p. 273], and Transverse activities [12].

Our motivation is to be able to construct entities that are larger than objects. We achieve this by taking outset in collaborations—that is multiple entities connected closely together with the focus of solving a given problem. Similar to [11] we have observed that between inner classes and their outer class, an implicit association is established. And secondly, that the outer class scope serve as a repository for state and functionality shared by the inner classes.

The first problem that strikes the mind where collaborations are useful, is the situation that state/functionality has to be

¹This is not a new idea, it has been proposed in many variants, e.g. complex associations [11], Aspectual collaborations, Object Teams [8], Caesar [18], mix-in inheritance [21], UML collaborations [19].

²The inspiration comes from the language Beta[15].

³It is smart but not crucial.

placed in one of n equally appropriate entities. Here a collaboration is a natural habitat. But we would like the status of collaborations to go further than being a sparingly used exotic feature. Rather, we want it to be a central element in future program development.

The usage of inner classes is spiced up with the utilization of roles, enabling a role of an object to be a part of a collaboration, rather than the object itself. This enables the object to participate in several collaborations simultaneously.

Focusing on the outermost class as a collaboration rather than a class changes the status of its methods and fields. Public methods become the interface of a set of objects, where private methods are secret helper methods internally for the collaboration.

4 Comprehensibility

Distinguishing between comprehensibility and predictability is difficult. For the purpose of this paper we define *comprehensibility* as a measurement for the immediately intelligible impression/understanding of a system. It is on the level of a UML diagram or similar high level understanding of structure, main entities and their relation and communication paths.

The parameters for comprehensibility are thus, being able to retrieve as much information possible from the static structure of the code. Navigational aid from the development environment significantly impacts comprehensibility to such a degree, that it can remedy many deficiencies in the language itself. This is one of the places, where the borders between development environment, language and documentation become blurred. The reason to draw documentation into the picture is based on the fact that programmers today usually spend more time in the documentation (e.g. the Java API), than on their compiler.

Yet there are many things taking place in code which cannot be understood or be presented by the development environment. Here the degree of interplay between documentation, language features and support from the development environment impacts the comprehensibility of the code.

The focus of OOP has been on the encapsulation of state and functionality, elevating the conjunction of these to the first class entity “class”. Along with this followed inheritance and aggregation, and later contracts for the usage, here exemplified by the design by contracts paradigm (DBC henceforth) first seen in Eiffel [16]. DBC enhanced the language in that it could be specified that a method not only returned a list of names, but that the names be sorted.

4.1 Support for comprehensibility

The rationale behind “role collaborations” is, that although classes and objects conceptually smoothly goes along with our understanding of concepts, phenomena and state, there are issues which are far from handled well. The focus on isolation and encapsulation of information (as objects) makes it difficult to capture processes, activities and relations. This makes it harder to express a functional decomposition of a problem,

which depending on the case of course, is just as conceptually pleasing as the focus traditional OO modelling prescribe. When entities are no larger than objects, one must rely on other facets of the language to establish links between entities. Access modifiers is a first candidate for this. Alas, access modifiers are too general, in that they apply to all other entities in the system—with a few exceptions such as the `friend` mechanism in C++ [22]. In dynamically typed languages access modifiers are typically not available at all. And in all cases, access modifiers merely represent compile-time restrictions for the type system.

As presented in section 3.2, the meaning of outer methods and fields is changed. Similarly are the inner roles, defining context specific collaboration with other entities. Further, the operations inheritance and aggregation can now be used to aggregate or specialize a set of entities. A classic examples of this is the graph traversal collaboration upon which a weighted-graph traversal collaboration can be specified as a specialization. This is possible due to the virtual inner classes (and roles), since roles can be overwritten in the subclass collaboration.

Finally, since the role concept is implemented transparently participants of a collaboration can view other roles in the collaboration differently, by applying roles to them (figure 2). The last possibility is to apply roles to the collaboration itself. This enables to view the public interface of the collaboration differently. But just as importantly, due to the virtual classes (and roles), the attaching role can overwrite classes and roles in the collaboration (when calls come through the role).

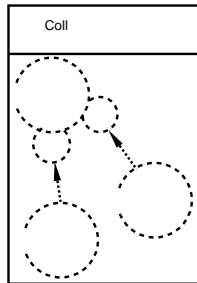


Figure 2: Role are free to apply roles on roles to suit their needs for interacting with the role. The dashed arrow shows a reference exist from one role to another.

4.2 Reduction of comprehensibility

Practical usage with the model may show that in order to assign objects to correct collaborations, require not only objects but potential collaborations to be passed around. A solution to this problem may in many cases be, that since an intrinsic knows the roles it plays, enough information is provided to be able to “dot” a way from an instance to the collaboration in question—hence passing around the collaborations can be avoided or greatly reduced.

5 Predictability

We define *predictability* as a parameter for how easy it is to understand the code when reading it. This definition is closely related to the comprehensibility parameter. Focus, however, is different: We are now closer to the source code. Often flexibility and predictability are two opposite parameters. Dynamically typed languages are more flexible, but harder to read since there is no type information available (although proper naming of variables can somewhat remedy this). The same story can be told when talking about late bound self references and polymorphic references, where only an upper bound of what kind of object is referenced and called on. This uncertainty is somewhat reduced in the language Beta, where the least specific version of the method is invoked first, which then calls down the inheritance chain rather than up. The code is more predictable, but less flexible. Using DBC does not help, since contracts may be weakened in subclasses. No OO programmers will give up this flexibility to get higher predictability; seeking low predictability cannot sensibly be a goal in itself. But as we have shown, mechanisms need not necessarily either be present or not present—the Beta approach is a good example of a middle way.

5.1 Support for predictability

In this section we address three issues: activation of collaborations, reference recycle, and scoping rules for name lookup. The first two issues are motivated by being issues in Object Teams and Caesar, the two latest collaboration-centric models in the AOSD community⁴. The last point is tied to the above discussion.

5.1.1 Collaboration activation

As outset collaborations are explicitly handled, hence there are no shortcuts for instantiating or activating collaborations. This gives a high predictability score, but requires more code potentially blurring focus. Object teams and Caesar has taken a more automated approach: They specify pointcuts which ensures instantiation of the specified collaboration, instantiating all roles with a mapping from the call to the collaboration. Take the call `a.foo(b,c)`. It can enable an instantiation of a collaboration with e.g. three collaborator roles mapped to the instances `a`, `b`, `c`. This mapping goes most smoothly if the instances are of different types. A more dynamic version is also available where the activation only takes place if the call happens within an activation block, hence the activation can be restricted to certain objects or certain execution time.

Static binding between collaborations and “the rest of the program” are in many cases practical, but reduces predictability. Predictability is further reduced when the collaboration is only activated in certain parts of the program. Except for the type-inference mapping shortcut, the automatic activation of a collaboration can be handled in our system by a combination

⁴Aspectual collaborations should not be left out, but we have limited knowledge about the model, except it has great similarities with the other systems.

of the class role concept and an around method for the triggering method. However, we have deliberately chosen not to walk down such a path. The biggest problem with the solution is that the actual collaborations are given a status of implicitness.

5.1.2 Reference recycling

“Reference recycling” is used both in Object Teams and in Caesar which ensures that a role for an intrinsic is only instantiated if the intrinsic have not previously visited the role. This ensures unique roles and preservation of identity and state of the roles when the same intrinsic enters a collaboration multiple times [17, p. 58] [8, p. 7]. From a predictability point of view, things are not looking good. First, object instantiation has different semantics, but has retained its original syntax. Secondly, the memory-usage is unpredictable, and could easily lead to memory-leaking situations as each role has to keep track of who has visited it.

Wrapper recycling fits well with the idea of “implicit collaborations”, but not well with explicit collaborations as presented here. For example, it is unclear, how a shift at the hospital is going to take place. A shift entails, that a new person takes over the doctor role of the patient-hospitalized collaboration. Wrapper recycling forces an allocation of a new role instance for the new intrinsic entering the collaboration. This is the opposite of what we want to model.

5.1.3 Name lookup rules

Examining various languages no common semantics for inner classes can be established. In languages such as C++, Python, C#, the inner class is merely lexically hidden from the outside, whereas e.g. in Java or Beta, the inner class resides in the scope of the outer class (block structure). We have chosen an implementation which has both flavours. The inner classes (and roles) reside in a scope, but access to the scope requires explicit use of a pseudo variable `outer` (initialized and maintained by the system). This makes the system less flexible, but far more predictable. Figure 3 shows how many places the name `i` can be looked up from role `A` had `outer` not been mandatory.

5.2 Reduction of predictability

We now extend the discussion of the previous section. We deal with the fundamental nature of roles in collaborations. Throughout the paper, we have uniformly termed the entities in a collaboration “roles”. This is in accordance to the terminology of our language as well as the example languages Object Teams and Caesar. However, although the two other systems refer to the collaborating entities as roles, in fact they are mere wrappers. We distinguish between wrappers and roles in that there is at least a delegation link between intrinsic and role (hence methods are late bound at run-time), whereas in the other systems, a consultation link is established. As discussed in section 5, late binding of methods reduce predictability, but enhances flexibility. For instance, it allows us to attach roles to

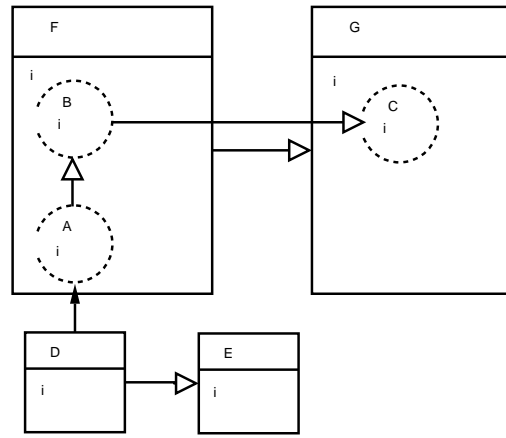


Figure 3: Figure illustrates the many places `i` can be found from within `A` and the names denote the lookup order. Making `had outer` mandatory reduces the number of possibilities and increases the predictability of the code.

collaborations and have their inner roles overwrite the roles of the collaboration when calls are made on the roles (see figure 4).

6 Evolvability

Evolvability is a parameter for how easy it is to adapt a system when changes happen in the requirement specification. These changes can either be “incremental”, or “destructive”. Incremental changes mean certain features needs elaboration, whereas destructive is removing features entirely or replacing them with something new. “adaptable” can be in the form of specifying changes as a subclass (e.g. a new collaboration). Evolvability and efficiency are typically contravening forces—evolvability typically comes at the cost of performance.

6.1 Support for evolvability

Our implementation is fairly versatile. We shall here briefly look upon the ways a system can evolve and how using multiple inheritance can separate pointcuts, advices and base code, making them easier to re-use independently of each other.

A prerequisite for understanding this section is to know a few implementational details. Our implementation is very straight forward. The class concept has remained unchanged. Inner classes has been augmented with an `outer` reference. Collaborations are just classes with inner classes and inner roles. Roles are implemented as classes with a delegation link to its intrinsic. Pointcuts are special methods which are invoked automatically by the system. They return true/false based on information on the executing environment (e.g. which method is being called). Advices are similarly also just specially named methods, which are automatically invoked by the system when their pointcut evaluates to true. Since everything is classes and methods most operations are given for free by the implementation language (in this case Python). The

current implementation has no emphasis on static typing issues.

6.1.1 Evolution of collaborations

Since collaborations are regular classes, they can be specialized. However, unlike a class specialization, specializing a collaboration means refining the interaction, i.e. the participating roles, or refining pointcuts and or advices—the binding of the collaboration. As collaborations are mere classes references are still polymorphic when referencing collaborations. This has been termed *aspectual polymorphism* by [18, p. 96] however, the term has not been ratified yet i.e. [4, p. 155] uses it in a broader sense for “polymorphism in connection with aspects”.

Since classes can play roles, a collaboration can participate in other participations.

6.1.2 Evolution of aspects

Since collaborations and roles are just classes, they can be subject to static specialization (including advices and pointcuts). As methods are virtual in Python, the language ensures the most specific version of pointcuts and advices are executed. As advices are implemented as methods, and since pointcuts are evaluated each time a method is to be invoked, it is unproblematic to specify around methods for around methods. This can either be used to re-define the existing around method, or describe new aspects of the system simply by just adding behaviour.

Both kinds of evolutions stand in contrast to Aspect/J, which allows only abstract aspects to be subclassed and has no way of specifying advices for advices or specialize pointcuts.

6.1.3 Composable pointcuts and around methods

We utilize multiple inheritance as a way of separating the distinct entities: roles, pointcuts and advices. This is done by putting the content of each entity in separate classes, and then merging these using inheritance. Multiple inheritance has a “bad reputation” partly due to hard to predict name lookup when dispatching and problems with name clashes. Our use situation differs in that classes containing only pointcuts and around methods contain nothing more. By separating the entities, each can be reused by letting several unrelated classes inherit them. It also allows for creation of separate hierarchies of pointcuts, advices and roles.

6.2 Reduction of Evolvability

A consequence of the model is that roles are confined in a scope. This confinement impedes specialization of a single role in a collaboration. Illustratively speaking, the outer scope (which forms the collaboration), is like a delimiting wall blocking the escape of participants. One can define a role **B** in collaboration **D** for the role **A** in collaboration **C**, but **B** will

always have as legacy **A** and **C**. That is, **B** can only be instantiated when **A** is instantiated, and **A** can only exist when **C** is instantiated. A possible solution is to refactor a common independent superclass **S**, which then is subclassed by **A** and **B**.

Another problem with evolvability is that in order for a role to be type safe, it can only be applied to objects that it has been written for or subclasses thereof. This may impede a general usage of a role, since the only common ancestor for the entities to apply the role to is **Object** (the superclass of all classes in the language). Creating roles for **Object** has the downside that nothing interesting is inherited or can be called (as nothing is known about the intrinsic). One solution is to specify a generic collaboration (with roles for **Object**) and then specialize the collaboration to more concrete ones. As an alternative we propose what we call *typed delegation*, which basically means that a reference’ type is a deciding factor in method selection during dispatch. In other words we allow role **RO** (written for **Object**) to be applied as follows: `Person p = new RO(new Person())` and further `p.getname()`. We can permit this since although **RO** is not of type **Person**, its intrinsic object is of such a type—hence by travelling the chain of intrinsics eventually the right method will be met. One must just ensure that the **RO** instance does not change intrinsic which is not of type **Person**. The technique has similarities with Eiffel, but was invented for different reasons. In Eiffel the type of the reference matters due to renaming and undefinition of properties in subclasses, we introduce the feature to gain evolvability.

7 Semantic interactions

We define *semantic interactions* as the phenomenon that by introducing a certain set of semantics for a language construct, this interacts with other parts of the languages semantics. Naturally, we must keep the semantic interactions at a minimum. When introducing run-time inheritance pr. object to a statically typed language, you are bound to run into trouble. Alas space prevents us from digging into these details here, but many excellent discussions are found in [10]. Instead we draw our attention to possible ill’ities the collaboration concept may introduce.

7.1 Roles on collaborations

Overwriting a collaboration **Coll**’s role **FOO** by Putting a role **R** on the collaboration (fig. 4), requires **R.FOO** to be a subtype of **Coll.FOO**. This requirement cannot be uphold statically since roles can be applied to subtypes of what it has been written for. If a subclass **CollSub** redefines inner role **FOO**, role **R** cannot be put on **CollSub** since its **FOO** will not adhere to the subclass restriction. We do not have a solution to this problem.

7.2 Exiting a collaboration

It is not clear what happens if a role overwrites an inner role in a collaboration, and from within the collaboration an instance

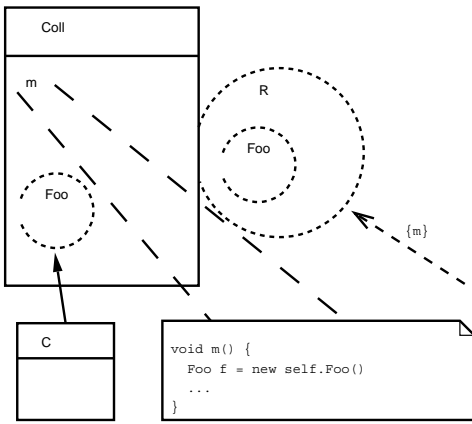


Figure 4: The figure shows that when calling on a role for a collaboration, its inner role **Foo** can be instantiated instead of the collaborations' inner role. This happens as when calling through **R** binds self to **R**.

of that role is made, then removing the role on the collaboration and then calling on the inner role instance. The instance still exists, but its scope doesn't. There may be a need for roles to some how "clean up" when being removed, or maybe inner classes should be disallowed. Certainly the problems related to removing a role stand out here. A bold proposal could be, that collaborations and roles simply do not fit together! It seems that there may be unsolvable problems in the wake of late binding and virtual inner classes.

8 Impact on other software...

Empty due to space limitations...

9 Discussion

Our implementation strategy has been a dynamically typed language with a mind for reusing as much of the implementation language as possible. This means that i.e. pointcuts and advices are implemented as methods and roles and collaborations as classes. Not only did this significantly reduce development time, interestingly it also provided for free, advanced features which are regarded valuable in the AOP community—and which are missing in popular AOP languages today. These features are aspect polymorphism, specialization of aspects, aspects for aspects, dynamic aspects (can be applied/removed run time and on the instance level). There has been put no emphasis on static typing issues in the implementation, since we find it more important to focus on use and use-situations first. The implementation is done in Python and is approximately 300 lines of code. This low number of lines is due two things, the lack of static typing, and the lack of certain things which normally are automated, which must be explicated by the programmer; an example is that different syntax must be used when calling on the self reference and passing the reference in a method call.

References

- [1] M. Büchi and W. Weck. Generic wrappers. In E. Bertino, editor, *ECOOP, LNCS 1850*, pages 201–225, 2000.
- [2] W. W. Chu and G. Zhang. Associations and roles in object-oriented modeling. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 257–270, 1997.
- [3] M. Corporation. *The C# Language Specification*. 1 edition, 2001. Available at <http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpdownload.asp>.
- [4] E. Ernst and D. H. Lorenz. Aspects and polymorphism in AspectJ. In *International Conference on Aspect-Oriented Software Development*. ACM, 2003.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1994.
- [6] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.
- [7] K. B. Graversen and J. Beyer. Chameleon, August 2002. Masters thesis. IT-University of Copenhagen.
- [8] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. *Proceedings of Net.ObjectDays*, 2002.
- [9] B. N. Jørgensen and E. Truyen. Evolution of collective object behavior in presence of simultaneous client-specific views. In *Proceedings of the 9th international Conference on Object-Oriented Information OOIS*, 2003. Lecture Notes in Computer Science, Vol. 2817. Springer Verlag, (2003) 18-32.
- [10] G. Kniessel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, Computer Science Department III, University of Bonn, July 2000.
- [11] B. Kristensen. Complex Associations: Abstractions in Object-Oriented Modeling. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 29:10, pages 272–283, 1994.
- [12] B. B. Kristensen. Transverse activities: Abstractions in object-oriented programming,. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 279–296. Springer-Verlag, 1993.
- [13] B. B. Kristensen. Associative modeling and programming. In *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'2002)*. Springer, 2002.

- [14] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.
- [15] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
- [16] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
- [17] M. Mezini and K. Ostermann. Integrating independent components with on-demand modularization. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 52–67, 2002.
- [18] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2003.
- [19] T. Reenskaug. Uml collaboration semantics, November 1999. <http://heim.ifi.uio.no/~trygver/1999/UMLcollab/uml-collaboration.pdf>.
- [20] T. Reenskaug, P. Wold, and O. A. Lehne. Working with objects the OOram software engineering method, February 2001. heim.ifi.uio.no/~trygver/documents/book11d.pdf.
- [21] Y. Smaragdakis and D. S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *Software Engineering and Methodology*, 11(2):215–255, 2002.
- [22] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.
- [23] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–369, 1996.