Composable Designs with UFA

Stephan Herrmann
Technical University Berlin,
10587 Berlin, Germany
stephan@cs.tu-berlin.de

ABSTRACT

While aspect–oriented approaches draw their strength from improving modularity at the source code level, their acceptance for wide spread application depends on a seamless embedding into the software life–cycle. When striving for an appropriate design notation for AOSD it is important to carefully balance the abstraction required by design with the precision required for implementing a given design.

We choose the Aspectual Collaborations model [7, 5] as a basis, which allows separate development of modules comprising sets of collaborating classes and a posteriori integration of such modules. We present UFA (UML for Aspects) as an extension of the UML, which maps these modules to packages in UML. We furthermore focus on multiple levels of bindings between modules and their elements. UFA closes the gap between collaboration based design and an implementation that allows unanticipated composition in an aspect–oriented style.

1. WHEN DO ASPECTS ARISE?

The notion of aspects as used in the AOP community seems to be closely related to the source code. It is defined by the crosscutting of concerns which is manifest in scattering and tangling of source code. In order to define a notation and a method for aspect–oriented design it is therefore important to examine whether the phenomenon of crosscutting also exists in earlier phases. There are two possible reasons, why crosscutting is less a problem in design, than it is in implementation:

Firstly, for some aspects there exists a level of abstraction at which they appear atomic, i.e., without internal structure. This is the kind of aspects for which it suffices to attach a tagged value or property to certain UML symbols in the design. Persistence is such a concern, which may well be scattered over large parts of a model, but at a given level of abstraction a property {`persistent`} is enough to say. There would be little advantage in separating out the definition of which classes should be persistent, because applying this aspect to an application is not more than creating a relation between a subset of classes and the persistence property. It is not a significant difference, how this relation is established: by inserting {`persistent`} at appropriate places in the class diagram (where it still can be hidden by a tool option), or by separately listing all classes that should have this property. This kind of aspect arises as an atomic property of certain design elements. Later on, such an aspect may well need to be refined, giving internal structure to what has been atomic before. At this point many CASE tools provide the capability of generating code fragments from stereotypes and tagged values. This generative technique corresponds to AOP techniques, as it introduces a new tool for multiply applying a structured concern to various model elements without requiring manual duplication of design or code.

Secondly, UML by itself is already multi–view capable. By definition, sequence diagrams[1], e.g., cut across the class structure of a system. Concepts can be kept apart in different diagrams, which contribute to the same graph of classes in the structural view. At a first glance this seems to imply that aspect–orientation has long been solved at the design level. At a closer look, much remains to be done, until it will be possible to produce really untangled designs which can smoothly be mapped to equally untangled implementation. Behavior modeling is more problematic in object–oriented design than structural modeling.

1. Sequence diagrams are only seemingly decoupled from class diagrams. In fact a sequence diagram may rely on almost every detail of its underlying class diagram: classes, associations, methods, signatures. Only inheritance is usually not visible in sequence diagrams. The effect is that class diagrams will be updated when creating a sequence diagram, and that modifying a class diagram may create arbitrary inconsistency to all dependent sequence diagrams.

2. The lack of compositionality of sequence diagrams has motivated a lot of research regarding their semantical foundations.

3. For some cases, sequence diagrams are too concrete,

Submitted for: Workshop on Aspect-Oriented Modeling with UML, April 22, 2002; Enschede, The Netherlands.

[1]In this discussion we focus on sequence diagrams for behavior modeling. Many observations will also apply to other behavioral diagrams.

and as [1] points out package parameters may greatly improve the reusability also of sequence diagrams.

4. Even if a design looks untangled it may not meet expectations as long as it does not give any hints on how to create an untangled implementation without major restructuring.

From our experience, the most significant problem with designs for aspect–oriented programs is in the independent stepwise definition of complex behaviors. This is although methods for role– and collaboration–based modeling have been around for many years now [9].

2. EXTENDING UML FOR PLUGGABLE COLLABORATIONS

We propose an extension to the UML, which is geared to modeling complex behaviors with the following properties and intentions:

- The behavior remains abstract at some points in order to remain reusable.
- Behaviors are designed for the purpose of being *applied* to one or more existing modules. This application of an aspect to a core module may have the effect of *adapting* the core.
- The behavior can be adapted to different contexts.
- Internal details of the behavior and of the core to which it is applied should be hidden to the other side. Independent refinement should be possible.
- The core module should in particular not be bloated with information from the applied behaviors.

The problem of crosscutting lies in non–trivial relationships between modules. Thus the `adapt` relationship between an aspect and a core module has structure, too. Accordingly, the emphasis of our UML extension lies on refining the binding by which a complex behavior is applied to a core module.

New design notations are often assessed by their independence from any programming language, i.e., by their abstraction over implementation. In contrast, we claim that the main goal of a design notation should be enabling the developers to smoothly move from analysis to implementation. Therefore developers need tools for abstraction *and* for refinement. Thus, the goal must be, to provide a minimal set of concepts, that allows an abstract conception of a large system as well as detailed instructions to the programmer. Since aspect–oriented programming languages (AOPL) are still quite young and no common structures have been agreed upon (despite the very similar intentions), a new design notation must choose a programming model towards which the refinement process is optimized. If seamlessness can be shown for one language, but transition to other languages remains difficult, it is still to be shown whether this is a deficiency of the AOD notation or the AOPL.

We chose for this work the model of Aspectual Collaborations [7, 5]. This model combines several composition techniques from existing AOPLs. It has been implemented in a language prototype called LAC [5]. A Java based compiler is under development.

2.1 From Decomposition to Composition

Separation of concerns and decomposition have been guiding principles in software engineering from the beginning. Central concepts in all aspect–oriented approaches add refined capability to speak about the composition of those disparate parts. Pointcuts in AspectJ [6] just like composition rules in Hyper/J [11] define the gluing of pieces of code from different modules.

Many different criteria have been proposed for a taxonomy of binding capabilities, including cardinality, granularity and time of binding [8, 3, 4]. Binding can furthermore be explicit or implicit, operationally or declaratively defined etc.pp. From this follows, that choosing the binding mechanisms significantly determines what designs can be expressed and whether these designs will be comprehensible and reusable.

Design should be regarded as a series of steps that add more and more details to an abstract model. It should be clear that designing the composition of modules into a system requires to capture the involved bindings at different levels of abstraction resp. detail.

2.2 Packages as first class citizens

Just like [1] we use UML packages for encapsulating parts of a system that contribute to a complex behavior. In order to give more semantics to packages than is provided by UML packages we allow defining top–level properties (attributes and methods) of packages, which encourages to use a package as a façade for controlled access to its contained classes. Graphically this is done by adding one or two corresponding compartments to the box representing a package.

When designing packages for reuse it is crucial to provide for extension points by which it can be configured for its different uses. Such packages can only be used in an application if all specific extensions are in place. Thus, the reusable part is in a way incomplete. We mark incomplete packages with an {`abstract`} property. Roughly spoken, this property should be applied if an abstract class exists, for which no concrete subclass is defined. Such a class cannot be used without additional measures like subclassing.

We have chosen the analogy to abstract classes rather than to template classes, because explicit parameters are only convenient when used in a small number and with little structure. However, package level incompleteness, as we will see in a moment, usually concerns several methods from several classes with their given signatures. At this level, developers are already familiar with incompleteness through abstract declarations. Technically, parameters and abstract methods are only different styles of defining open spots. What matters, is the provision for powerful binding mechanisms.

In accordance to the class–like status, which we give to packages, the specialization relation can also be applied to packages. This is, among other intentions, how concrete packages can be derived from abstract ones. A refinement (specialization) of a package contains all classes and package level

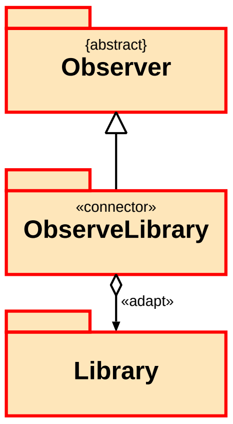


Figure 1: Applying the Observer pattern

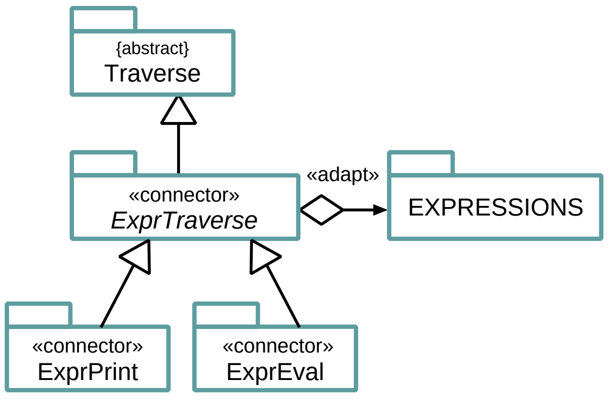


Figure 2: Two applications of a general traversal module

features of its super package. It may specialize an inherited class by simply defining a class of the same name, that implicitly inherits from the corresponding class of the super package.

2.3 Package composition — abstract level

The most common relationship between packages is classes of one package (explicitly) using classes or interfaces of the other package (dependency/uses relationship). The previous section has also introduced a refinement or specialization relationship for packages. A third relationship results from analyzing how AOPLs support independence between modules. Independence can either be two–way resulting in a symmetric binding as is the case with concern composition in Hyper/J [11]. Alternatively, binding can be performed within one of the involved packages. This is closer to the AspectJ [6] approach. We introduce a directed *adaptation* relationship which declares that an adapting package may use and influence another (adapted) package, without the latter knowing about the former. The adaptation relationship plus a refinement relation suffice to realize also a posteriori integration of two or more independent packages by extending one package and at the same time adapting one or more packages. For an example see figure 1. It reuses the example of [1] where the Observer pattern is applied to a model of a library in order to keep a book manager up-to-date regarding the status of book copies, which can be available or borrowed. So far, our figure adds little to the notation given in [1] other than giving a name to the binding relation that now is represented by the package `ObserveLibrary`[2]. This package is a refinement of the general Observer pattern which now adapts the library core model. The direction of the adapt relation signifies that the core model is unaware of this adaptation, but the adapting package knows the core. Also package `Observer` is unaware of the library package, only the connector has all needed information in order to weave the Observer behavior into the library model. This can roughly be compared to AspectJ's feature of abstract pointcuts: an aspect modifies a yet undetermined module.

[2]Our notation even seems to remove the essential part of composition patterns: package parameters, but these will re-appear in a different shape at the next level of detail.

Note, that refining `Library` is not an option, because it must be possible for other parts of the system to operate on the original package while still (unknowingly) taking benefit from the adaptation. The role of the intermediate package `ObserveLibrary` is additionally emphasized by a stereotype `«connector»`. The precise distinction between the annotations `«connector»`, `{abstract}`, and still other forms of incompleteness also requires considerations at the level of programming languages and thus is beyond the scope of this paper.

Another illustration is given in figure 2. The right hand package `EXPRESSIONS` contains the structural definition of an expression language. Let's assume this package is reused from a different project. This is illustrated by different naming conventions within this package — package and class names in all capitals — as compared to the mixed capitalization in all other packages.

Package `Traverse` defines traversal over arbitrary structures that follow the Composite design pattern. This package will only define a skeleton of traversal leaving a set of hook methods abstract. That's why the package is marked abstract.[3]

A direct sub–package (in terms of specialization) called `ExprTraverse` partially binds the abstract traversal to a given expression structure. This package connects packages `Traverse` and `EXPRESSIONS`, which is noted by the stereotype `«connector»`. From this still abstract package two specializations are derived that implement concrete traversals: evaluation and pretty printing. This design obsoletes the Visitor design pattern, as it allows separating structure and traversal. It is superior to the Visitor pattern, as it requires no pre–planning within the structure–defining classes.

Each connector stands for one application of the abstract collaboration (pattern) to a given core model. Reifying the connector serves two purposes[4]. First, it supports reuse of

[3]For explicitness, in our examples we mark package level abstractness textually using `{abstract}`. Normal abstract classes that are super classes of concrete classes have a name written in italics as suggested by the UML.

[4]This is at design level. Implementation draws even greater

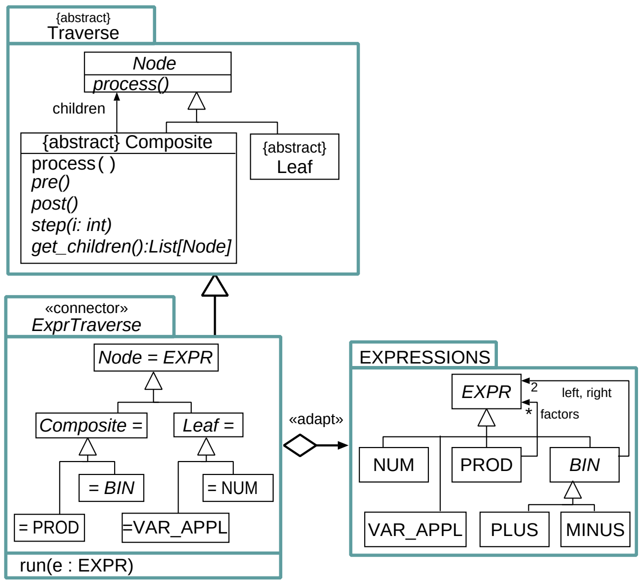


Figure 3: Mapping classes between packages

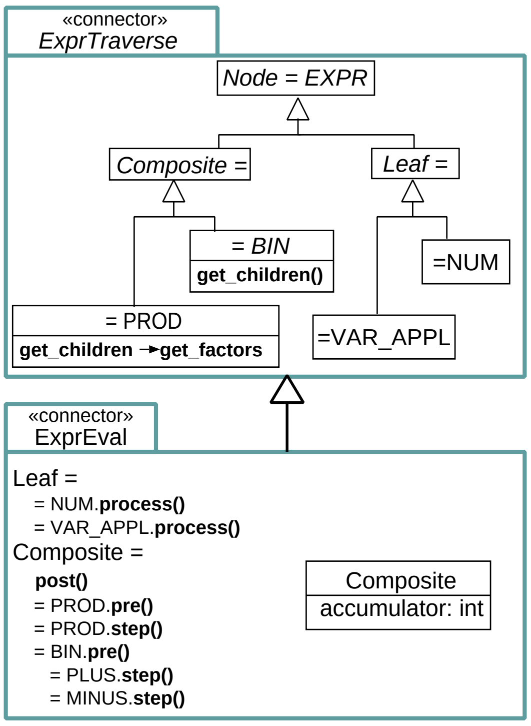


Figure 4: Method bindings

partial bindings. Secondly, this gives a natural place for adding details during the refinement process. These details can either be additional features or binding of existing features. We introduce notational means for binding details in two steps: at class–level and at feature–level. A programming language like LAC needs to also consider binding at the level of method parameters. We consider this last level — but only this last level — as beyond the scope of design.

2.4 Package composition — class level

The first step is shown in figure 3.[5] Here we specify which role in the abstract pattern is to be realized by which class from the core expression model. Note, that this mapping is not necessarily 1:1. In fact, it is central to AOP (as to any technique that supports reuse) to apply one concept at different places simultaneously. In this vein the role `Composite` from package `Traverse` is played by the set of classes `PROD` (n-ary products) and `BIN` (binary expressions) with its two subclasses `PLUS` and `MINUS`. Similarly, `Leaf` is played by classes `NUM` (number) and `VAR_APPL` (variable application). Our notation provides three options:

- A role class that is uniquely mapped to a core class appears printed as `ROLE=BASE` (cf. `Node=EXPR`).
- A role class that has no direct correspondence in the core package appears as `ROLE=` (cf. `Composite=`) signifying that bindings exist at a different level.
- A set of core classes that is bound to the same role class is shown as specializations of a (possibly unbound) role class using a `=BASE` notation (cf. `=PROD`).

Options two and three are a shorthand for specifying, e.g., `Composite=PROD`. While the differently bound versions of a role are in fact specializations of the abstract role, the role class can also be compared to a *supplementary super type*. In the traversal's view, expressions `PROD` and `BIN` have a common super type `Composite=` that did not exist in the original expression package. Subclasses `=PROD` and `=BIN` do the necessary adaptations in order to produce conformity to the common super type.

The reverse situation to the introduction of super types would be to map different subsets of one core class to different roles. On the object level, the first situation defines set union, while the second situation introduces subsets. The latter has been discussed in [4] but falls beyond the scope of this paper. The relevance of such mappings between mismatching inheritance structures lies in the quest for unanticipated composition of independently developed packages.

Aside from class mappings, figure 3 also shows one package level method, **`run`**. This method is the main entry to the functionality provided by the package. Clients of the package need not know any further details. The sub–package `ExprEval` (not detailed in this paper) also declares a package level attribute, by which the result of **`run`** is made available to clients. This demonstrates the similarity of this design to the Visitor pattern, i.e., the package itself acts as a visitor, that can, however, be applied without any pre–planning in the expression package.

2.5 Package composition — method level

When binding a role class to a base class, this binding has to be detailed at the method level. First of all, a role class may contain abstract methods, for which an implementation must be provided during binding. An implementation can

benefit from this concept.

[5]We skip this step for the observer example.

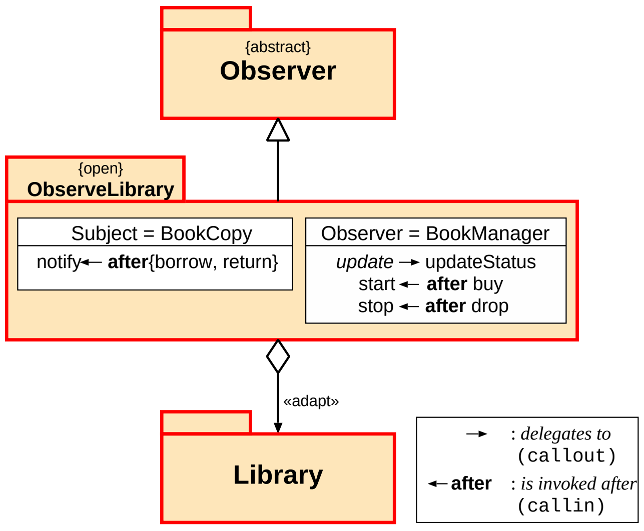


Figure 5: Different styles for method binding

be given by a method binding or in–place. For an example of in–place implementation, look at figure 4. Method `get_children()` in class `Composite=BIN` is implemented directly. It assembles a list from the current object's `left` and `right` sub–expression.

Using package refinement as in `ExprEval`, it may be convenient to just list all inherited methods that need to be defined. There is no benefit in drawing the same class structure again as in package `ExprTraverse`. The structure of such listings applies indentation to represent inheritance structure. This notational style is used when a package introduces no or little new structure, but only fills in the yet abstract methods from its super–package. It can be seen as a checklist for the programmer, which collects in one place all the methods he or she must implement within the given package.

Instead of in–place implementation, methods may also be bound in a declarative way as demonstrated by `get_children` in `Composite=PROD`. The method that is needed here already exists as `PROD.get_factors`. In this case delegation may be specified using a delegation arrow $\longrightarrow$. Such notation shows that nothing really remains to be implemented, but declarative binding in terms of delegation suffices.

A second way of binding methods is demonstrated in figure 5. Method `Subject.notify` needs to be *woven* into the core model. More precisely, two methods — `borrow` and `return` — need to be augmented by this additional behavior. This technique is inherent to many AOPLs. In order to illustrate similarity and difference to delegation–style bindings, we use the reverse arrow $\longleftarrow$, and speak of *callout* (delegation) versus *callin* (advice weaving) bindings. These names capture bindings from the perspective of a role class, that either uses external behavior (callout) or requests the base to call into the role method (callin). Just in the style of AspectJ, callin bindings can be specified as `before`, `after`, or `replace` (in AspectJ: around) advice.

If method definitions and method bindings are given in the same role class, these appear in different compartments. I.e., a role class may at most contain four compartments: name, attributes, methods, bindings.

3. RELATED WORK

Design notations for AOSD are a fairly young issue. The most advanced approach, to the best of our knowledge, is by Clarke and Walker. Their work has influenced the development of UFA. During the aspect workshop of ICSE 2001 it has been noticed that Clarke and Walker's composition patterns [1] fit very nicely to Aspectual Collaborations [5]. The examples for composition patterns have been implemented in LAC with great ease. Therefore with using Aspectual Collaborations as our conceptual foundation the motivation given in [1] applies also to UFA. The decisive difference lies in reifying the binding relationship. While composition patterns attach all details to a dependency relationship of stereotype `«bind»` we propose to use an additional package symbol, which can hold all details of this binding.

The UML requests such intermediate symbol even for instantiating a template class. We consider this an unnecessary burden, because no additional information is given by this symbol, which is thrown away for the implementation anyway. In contrast, we advocate to see composing different concerns as a concern in its own right. We have presented binding details regarding classes and methods that would without the connector symbol not have a natural place in diagrams.

Composition patterns introduce an additional convention for distinguishing two different methods of the same name: for the case of around advice, they prepend an underscore to the method name and use this as a package parameter. This gives a name to the base method to which an advice will be bound. We consider this superfluous, because this name may only be used within the corresponding advice. This would be like giving explicit names to all methods that are inherited but overridden. It is a matter of the programming language to provide means (like `super` for inherited methods) for accessing these methods within overriding or advising methods.

For a more in depth discussion of related work we refer to [1]. Much of what has been said there could only be repeated here. This concerns the discussion of SOP[2] and especially of OORAM[10].

4. SUMMARY

We have presented an extension to the UML called UFA (UML for Aspects), which treats aspect composition at the level of packages. Aspects as well as their bindings make use of a specialization relationship between packages. Details of binding can be given at the levels of classes and methods. These can bridge mismatching structures of an aspect (collaboration) and a core model to which it is being applied. Two styles of method binding enable a connector package to use functionality of the base model and also to modify the behavior of this base model. These two styles correspond to delegation and advice weaving.

We have presented the notation using the well–known design patterns Observer and Composite. The latter example replaces the Visitor pattern with better separation of a structure definition and traversals over this structure. These examples should, however, not imply that UFA only aims at modeling design patterns. We have first results of using this notation for modeling higher–level behaviors from business domains. It is this kind of application we had in mind when developing UFA for modeling complex behaviors. All our UFA models have also been implemented using the language prototype LAC. The results show, that UFA models can smoothly be implemented given a matching AOPL. Both a graphical editor for UFA and a Java based compiler for Aspectual Collaborations are being developed. Using both tools we plan real world case studies of UFA and Aspectual Collaborations for assessing both techniques and their combination. While we wouldn't preclude the use of UFA with different AOPLs, we hope that UFA plus Aspectual Collaborations will yield improved modularity of models and code, and smooth transitions between both representations.

5. REFERENCES

[1] S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *Proc. of the 23rd ICSE*, 2001.

[2] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proc. of OOPSLA'93*, pages 411–428. ACM, 1993.

[3] S. Herrmann and M. Mezini. Dynamic view connectors for separating concerns in software engineering environments. In *Proc. of MDSOC workshop at the 22nd ICSE*, 2000.

[4] S. Herrmann and M. Mezini. PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *Proc. of OOPSLA 2000.* ACM, 2000.

[5] S. Herrmann and M. Mezini. Combining composition styles in the evolvable language LAC. In *Proc. of ASoC workshop at the 23nd ICSE*, 2001.

[6] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, and J. Palm. An overview of aspectj. In *Proc. of 15th ECOOP,*, number 2072 in LNCS, pages 327–353. Springer–Verlag, 2001.

[7] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, April 1999.

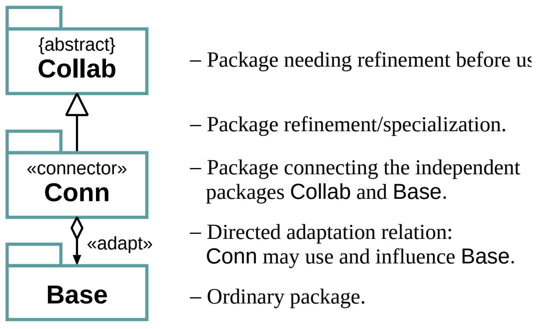
[8] K. Ostermann. Object-oriented composition: An analysis and a proposal. Master's thesis, University of Bonn, 2000.

[9] T. Reenskaug, E. P. Andersen, A. J. Berre, A. Hurlen, A. Landmark, O. A. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. L. Skaar, and P. Stenslet. OORASS: Seamless support for the creation and maintenance of object oriented systems. *Journal of Object-Oriented Programming*, Oct. 1992.

[10] Trygve Reenskaug. *Working with Objects – The OORAM Software Engineering Method.* Prentice Hall, 1996.

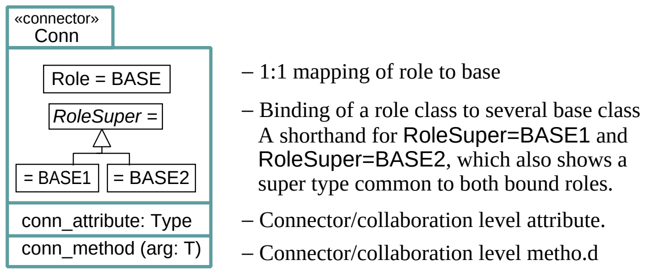
[11] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual.* IBM Corporation, 2000.

6. APPENDIX: UFA SUMMARY



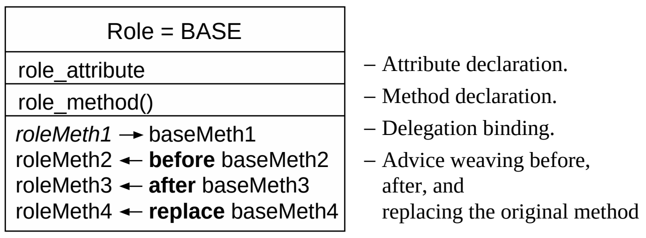
{abstract}
Collab
«connector»
Conn
«adapt»
Base
– Package needing refinement before us
– Package refinement/specialization.
– Package connecting the independent packages Collab and Base.
– Directed adaptation relation: Conn may use and influence Base.
– Ordinary package.

Figure 6: Package kinds and relationships



«connector»
Conn
Role = BASE
RoleSuper =
= BASE1
= BASE2
conn_attribute: Type
conn_method (arg: T)
– 1:1 mapping of role to base
– Binding of a role class to several base class A shorthand for RoleSuper=BASE1 and RoleSuper=BASE2, which also shows a super type common to both bound roles.
– Connector/collaboration level attribute.
– Connector/collaboration level metho.d

Fig. 7: Class mappings and collaboration level features



Role = BASE
role_attribute
role_method()
roleMeth1 → baseMeth1
roleMeth2 ← before baseMeth2
roleMeth3 ← after baseMeth3
roleMeth4 ← replace baseMeth4
– Attribute declaration.
– Method declaration.
– Delegation binding.
– Advice weaving before, after, and replacing the original method

Figure 8: Method bindings