# Towards a Composition Taxonomy

Klaus Ostermann

Siemens AG CT SE 2

osterman@cs.uni-bonn.de

**Abstract**

Different similarities and differences between AspectJ, the first general-purpose AOP-language, and Hyper/J, a recent implementation of multi-dimensional programming, have been identified. This paper explores how both proposals give different answers to the same question: How can we modify existing classes, that are used by existing clients?

In this chapter, different criteria are developed by which all reviewed proposals can be compared and classified on an abstract level. Our general model in this chapter is that we have *composition units* which have to be composed in order to build a *composite unit*. A composition unit can be almost anything, e. g. objects, classes, mixins, aspects, connectors.

In order to compare the new composition models to the existing ones, inheritance and object composition are included in our investigations. Other well-known language features such as structural subtyping or generic classes are mentioned in case they help to explain a term.

## 1 Coupling

Coupling describes the nature and extent of the connections between elements of a system. We define coupling as a criteria that describes the kind of dependence of one composition unit on another composition unit[1].

We define four different kinds of coupling which are explained below. The coupling structure of the composition models is summarized in figures 2 and 3.

### No Coupling

The composition unit is not dependent on any other composition unit with which it may be composed. For example, an object usually does not know anything about aggregators of the object.

[1]This definition is not symmetric: The coupling of the other composition unit to the first one may be completely different.

## Constraint Coupling

The composition unit can be composed with different other composition units, but the other composite unit has to obey specified constraints. These constraints can be defined by means of the subtype relationship, but many other constraint definitions are possible. Darwin, object composition, personalities and mixins use the classical subtype polymorphism. In Hyper/J, a hyperslice has to be *declaratively complete*, which means that it must declare everything to which it refers; locally undefined (abstract) declarations have to be mapped to some implementation in a hypermodule. Abstract classes can only be subclassed by classes providing implementations for the abstract methods[2] (as long as these classes are not abstract themselves). We can divide constraint coupling in two forms of compliance definitions

- **Explicit constraints**: Units that obey the constraints explicitly refer to the constraint definitions.
- **Implicit constraints**: Compliance to the constraints is inferred by the system.

and further divide them by the location of the constraints. Internal constraints are again divided into declared and hidden constraints.

- **External constraints**: The constraints are declared outside the unit.
- **Internal constraints**: The constraints are specified inside the unit.
  - **Declared Constraints**: The constraints are declared.
  - **Hidden Constraints**: The constraints are hidden in the implementation (see figure 1 for an example in AspectJ).

The usual subtype polymorphism is an example for external explicit constraints. The need to declare conformance to the constraints usually leads to *sharing of constraints*, e. g. a `Comparable` interface might be used by many classes. However, this may lead to *overconstraining*, e. g. an instance variable is typed to `LinkedList`, although the implementation of the list is not important, but a more general list type has not been reified. Thus, overconstraining limits reuse. If, on the other hand, every possibility for reuse is factored out, this may lead to an explosion of types. For example, Johnson and Rees proposed a fine-grained inheritance structure for a list class, which led to a list class that is made up of about 25 different classes organized in a complex multiple inheritance hierarchy [JR92].

These drawbacks are avoided by the use of implicit constraints. They allow fine-grained reuse while at the same time avoiding the type explosion. Besides the models found in this thesis, external implicit constraints can be found in languages featuring *structural subtyping* (or *implicit subtyping*) [Car88], e. g.

[2]Kiczales and Lamping named the interface between a class and its subclasses the *specialization interface* [KL92].

```
class A {
  void foo() { bar(); }
}
class B {
   introduction A {
     void bar() {...}
   }
}
```

Figure 1: Coupling of a class to its aspects in AspectJ may be implicit, internal and hidden

Emerald [RTL91]. If, however, constraint conformance is checked by the system, this may lead to wrong results. For example, `Cowboy` is not a subtype of `Drawable`, although it has a `draw()` method [Mag91].

Explicit internal constraints (e. g. abstract methods) limit the reusability of client composition units, because these clients are hard-coupled to the unit with the aforementioned constraints (clients have to explicitly refer to the constraints, but these are only defined inside the composition unit).

Examples for hidden constraints are Smalltalk-like polymorphism (an object can be passed to a method if it is able to respond to all methods that are send to the object in the method, otherwise a "message not understood" error is produced at runtime) and usage of global variables. Hidden constraints may also show up in AspectJ, see figure 1. We consider hidden constraints bad, because the detection of effects of changes becomes extremely difficult.

No coupling can sometimes be seen as a degenerated form of constraint coupling. For example, a non-abstract class is a degenerated form of an abstract class. If the constraints can be empty, we speak of *optional constraint coupling*.

## Selection Coupling

The unit is coupled to a set of other units. The elements of this set are all units, that pass a certain selection criteria, e. g. all units, whose name starts with `A` or all units, that contain a method returning an integer. The selection criteria may refer to unit invariants (i. e. properties that do not change at runtime) or to unit state. We call the former case *compile-time selection*, the latter case *runtime selection*. Selection coupling requires a *selection scope*, i. e. the set of all inspected units relative to the selection criteria. If the selection scope should include all available units, this requires a closed world where this information is requestable (see section 2).

Runtime selection poses many difficulties, see [Sch98] for a discussion on *constraint classes* (a special case of runtime selection). One major reason is, that the selection may change as a side-effect of an update.

### Hard Coupling

The unit explicitly refers to another unit and cannot be reused in another composition. For example, a class is hard-coupled to its superclass.

## 2 World Assumptions

The composition/compilation of units may happen under two different so-called *world assumptions* [PS94]:

- **Closed-world assumption**: All parts of the program are known at compile-time. The compiler has a *global view* of the program, which enables more advanced optimizations (dead code detection, flow analysis) and the detection of relations which are not visible locally, e. g. a unit A may influence the compilation of a unit B, although A is never referenced in B. Compilation under the closed-world assumption does not scale well.
- **Open-world assumption**: Different independent parts of the program can be compiled separately. The open-world assumption allows advanced features like dynamic class loading [LY96] and scales well.

## 3 Composition Binding

The composition process of a composite unit may take place at different times. We divide the proposals in three categories:

- **Static binding**: The composition is completely static (inheritance, mixins, Hyper/J, JPerspectives, static personalities, static aspects in AspectJ, aspectual components).
- **Activation binding**: The structure of the composition is static, but different units of it can be activated and deactivated at runtime (dynamic personalities).
- **Dynamic binding**: Units of the composite unit can be freely replaced at runtime by other units fitting predefined constraints, e. g. subtypes of a given type (object composition, Darwin).

## 4 Composition Scope

Sometimes a unit can be accessed in different ways and behaves differently depending on the way of access. We call each possible access way a *handle*.

We divide the different models in two groups:

- **Single handle**: Each composite unit has a unique handle by which all features of the unit can be accessed. For example, the features of all superclasses of a class are accessible by a reference to an instance of that class.

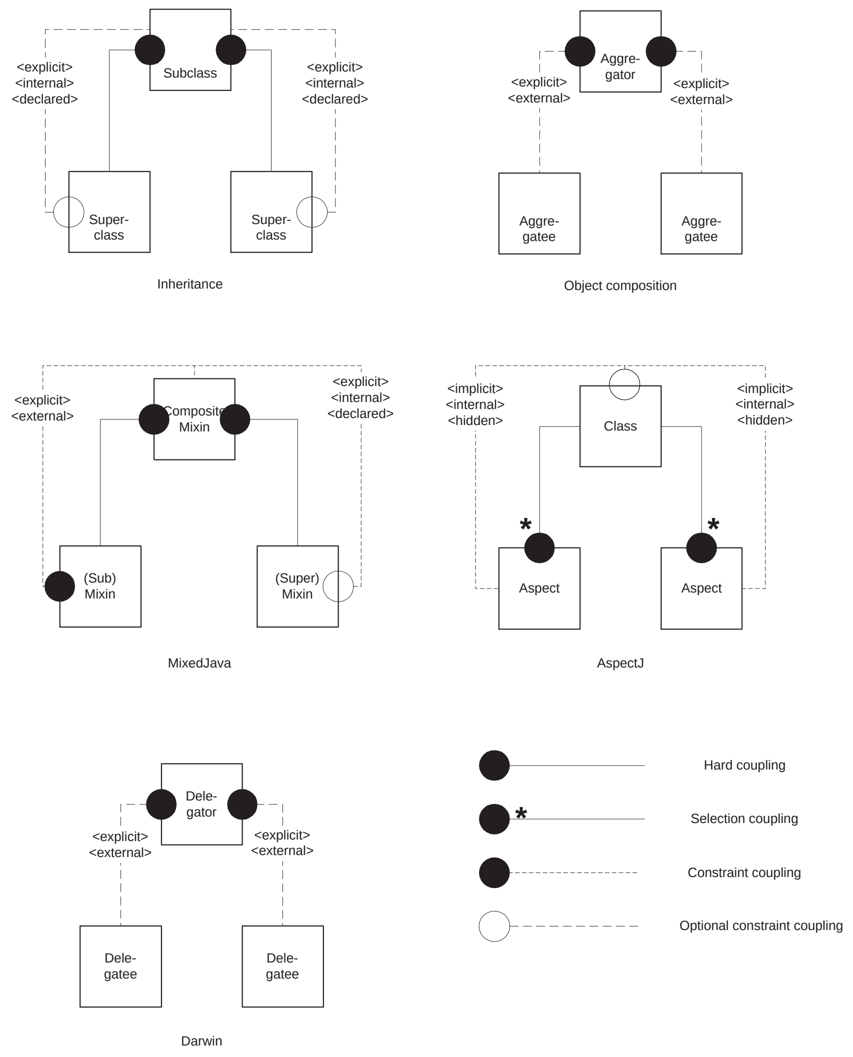

Figure 2: Coupling, part 1

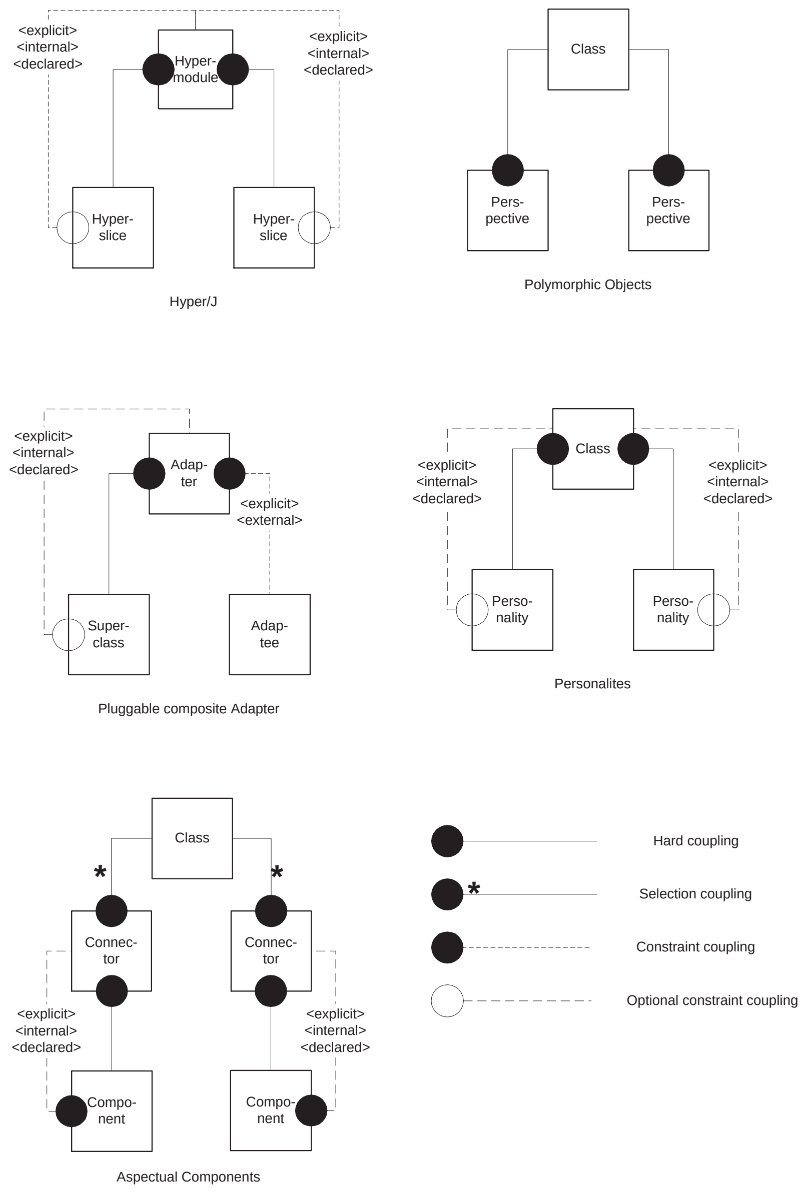

Figure 3: Coupling, part 2

- **Multiple handles**: The composite unit has more than one handle. The behavior, available features and internal interaction depend on the handle that is currently used. For example, an object can be viewed through different perspectives in JPerspectives. Every dynamic composition offers multiple handles (the different OIDs).

## 5 Composition Identification

If the structure of a composite unit is static (static binding and activation binding), the particular composition must have an identification that can be used to create an instance of the composite unit. Three identification schemes can be found in the composition models of chapter **??**:

- **Dedicated Unit (DU):** A specific unit is selected whose identification represents the composite unit. After the composition, it is not possible to refer solely to this unit, because it is identified with the composition. We call this unit the *dedicated unit*, the rest of the units *subordinate units*. In AspectJ, a class automatically consists of all aspects that are plugged into it. Other composition schemes in this category are inheritance, aspectual components and personalities. The disadvantage of this composition identification is that the dedicated unit cannot be reused separately without its subordinate units.

- **Identified Composition (IC):** The composite unit is identified with the composition itself, the subordinate units are left untouched. When using mixin-based inheritance, a specific composition of two units (mixins) has its own name. Hyper/J generates a completely new set of classes for each hypermodule. Generic classes like C++ templates also belong to this category. The obvious advantage is that all units can be reused in other configurations.

- **Aliasing:** A composite unit may have different identifications and each unit identification represents the whole composite unit. These different aliases may not be completely equal (e. g. they may correspond to different handles of the composite unit, although they refer to the same composite unit. In this case, no unit can be reused on its own in another composite unit. The only proposal in this category is polymorphic objects.

## 6 Composition Deployment

An important goal of component-oriented programming is that software should be extensible without modification of existing code. However, this goal cannot be satisfactorily achieved with current object-oriented technology (see discussion in problem **??**).

In this chapter, we present four different techniques to deal with this problem: *In-place addition*, *in-place modification*, *client migration* and *in-place client* *modification*. Three of them can be found in the state-of-the-art models; we think that the fourth one is an interesting middle road that complements the other ones.

- An *in-place addition* is the enrichment of the interface of a component after it has been constructed and delivered and without modification of its (eventually not available) source code. Such changes typically consist in adding data fields or methods that the original component did not support. JPerspectives (**??**) allows the addition of perspectives to a class, that encode additional state and behavior. Pluggable composite adapters are also related to in-place addition because an adapter can be associated with an object at runtime.

- An *in-place modification* is a modification of the behavior of a component without changing the component itself. An example is the definition of an action that should be executed instead of or in addition to a method body whenever that method is called. For example, an aspect in AspectJ (section **??**) can augment method bodies of its base class with aspect-specific code.

- A related strategy is *client migration*, but instead of changing the behavior of the existing component, a new extended component is created. The new behavior is incorporated into the application by duplicating clients[3] of the original component. In these duplicates, all references to the original component are replaced by references to the extended component. The Hyper/J example in section **??** shows that client migration may serve as an alternative for in-place modification (AspectJ example in section **??**).

- *In-place client modification* is a combination of the previous two techniques. In this case, an extended component with the new behavior is created. That new component is incorporated into the application by an in-place modification of clients. The in-place modification replaces all references to the original component with references to the new component.

**In-place addition** enables decentral addition of state and behavior to a component. It does, however, not alter any previously available behavior. For this reason, only clients that know about the augmentation can participate in it. With conventional technology, the effect of in-place addition can be partly achieved by one of the following techniques:

- If in-place addition is anticipated, a dictionary can be put into the classes. New state and behavior is encoded in separate classes and instances of these classes are stored in the dictionary. The problem in this case is the insertion of the objects into the dictionary. Objects could register itself in the dictionary in their constructor, but then the question is: Who

[3]In this context, clients are direct users of the component, e. g. subclasses or classes creating instances of a class.

creates these objects? In a language like C++, a global object could do that job, because these are initialized when the application is started. In a programming language like Java, classes are initialized by the class loader. However, the class loader will not load a class until it is used by some other class. For this reason, the global object approach will not work in Java. A possible solution would be to use class names as keys in the dictionary and objects of these classes are created as needed by usage of the Java reflection facilities.

- If in-place addition is not anticipated, essentially the same technique can be applied with a global dictionary that stores the dictionaries for the augmented objects (a dictionary is retrieved from the global dictionary with the object as the key). However, this technique is even less elegant than the first one.

**In-place modification** is destructive in the sense that the original behavior of the component is no longer available after composition. Thus, in-place modification is appropriate if *all* (known and unknown) clients of this class are intended to participate in the modification. If only selected clients should use the modified class, in-place modification fails.

**Client migration** is better suited for selective modification, because existing classes are not changed. On the other hand, this strategy imposes a maintenance problem, if the modified class should be used by all (existing and future) clients, because all clients have to be listed explicitly in the composition specification.

Additionally, client migration poses the question of how to handle clients of the migrated clients (*clients-of-clients problem*). Either these clients-of-clients are migrated, too, or they have to be invasively modified, or they continue to use the old version of the modified class. The first case can be iterated until the main entry into the system, e. g. the starting class, is migrated – in this case, the deployment has to be changed invasively (with of course less weighty consequences).

In-place client modification seems to be a promising middle road between the two other approaches. It enables selective modifications but it does not suffer from the clients-of-clients problem.

## Independent Extensibility

A useful property of an extensible component is *independent extensibility*: A system is independently extensible if it is extensible and if independently developed extensions can be combined [Szy96].

The difficulty in achieving independent extensibility when using in-place modifications is that different extensions might conflict and no central instance (like the inheriting class in a multiple inheritance hierarchy) is available to resolve this conflict.

For example, AspectJ suffers from this problem in case of conflicting introductions, which result in compiler errors. One could be tempted to limit the visibility of introductions by using `public`, `protected` and `private` modifiers. However, this will only reduce the number of conflicts but will not help to deal with the remaining ones, e. g. two conflicting `public` introductions.

Advice conflicts can be influenced by defining a partial order on the aspects, which introduces a low coupling among them. However, ordering aspects will often be too coarse grained, because an ordering that is appropriate for conflicting advices related to one particular method might be fully inappropriate for advices referring to another method. For instance, one might have an original class with methods `start()` and `end()` and two aspects: The one advices that a new method `openTheDoor()` should be invoked before `start()` and `closeTheDoor()` before `end()`; the other one advices that a new method `goInside()` should be invoked before `start()` and `goOutside()` before `end()`. Clearly, the only sensible sequence of events is: `openTheDoor()`, `goInside()` and `goOutside()`, `closeTheDoor()`, which corresponds to different orderings of the advices from different aspects.

A more subtle problem is related to updates: If an aspect updates state of its object, state hold or introduced by other aspects will in general become inconsistent. It is impossible to detect automatically, let alone to reconcile, different (implicit) invariants assumed by different aspects. Therefore, the only possible solution seems to be to let each aspect enforce its own invariants. This would be possible if aspects were informed about state changes in the shared object (e. g. as listeners of corresponding events). However, this is no general solution because the actions taken by one aspect to enforce its invariants might lead to further changes of shared state that might lead to other actions by other aspects, and so on *ad infinitum*.

Client migration, on the other hand, suffers from difficulties when it comes to the combination of extensions. Either a new component that combines the different extensions is created, which poses the clients-of-clients problem, or an existing component is modified in order to include the extensions. For example, in Hyper/J we can either create a new hypermodule containing all clients (clients-of-clients problem) or we can change an existing hypermodule specification.

## 7 Unit Interaction

Units may interact in a number of ways. We distinguish *explicit interaction* from *implicit interaction*. An interaction between A and B is *explicit*, if A explicitly uses or calls a feature of B, e. g. an object calls a method of another object. The interaction is *implicit*, if A does not “know” that it interacts with B, e. g. B overrides a method of A.

In a sense, we could have exchanged explicit with *anticipated* and implicit

| | **Inher.** | **Obj. comp.** | **Mixins** | **AspectJ** | **Darwin** |
|---|---|---|---|---|---|
| Composition identification | | | | | |
| dedicated unit | X | | | X | |
| identified composition | | | X | | |
| aliasing | | | | | |
| client migration | | | | | |
| in-place modification | | | | X | |
| in-place addition | | | | | |
| Composition binding | | | | | |
| static | X | | X | X | |
| activation | | | | | |
| dynamic | | X | | X | X |
| World assumption | | | | | |
| open | X | X | X | | X |
| closed | | | | X | |

Table 1: Properties of the composition models, part 1

| | **Hyper/J** | **PO** | **PCA** | **Person.** | **AC** |
|---|---|---|---|---|---|
| Composition identification | | | | | |
| dedicated unit | | | | X | X |
| identified composition | X | | | | |
| aliasing | | X | | | |
| client migration | X | | | | |
| in-place modification | | | | X | |
| in-place addition | | X | X | | |
| Composition binding | | | | | |
| static | X | X | | X | X |
| activation | | | | X | |
| dynamic | | | X | | |
| World assumption | | | | | |
| open | X | | X | X | |
| closed | | X | | | X |

Table 2: Properties of the composition models, part 2

with *unanticipated*, because explicit interaction is built in, while implicit interaction is patched in externally.

In the next subsection, we try to develop terminology to describe implicit interaction. A difficulty in analysing the interaction between the units was that the interaction patterns may depend on the handle that is used (cf section 4). For this reason, we analysed the interaction separately for each available handle in a composition model.

The application of these investigations to the composition models is shown in figures 4 and 5. The composition models in figure 5 exhibit handle-dependent interaction and the interactions for each possible handle are shown.

## Implicit Interaction Disassembled

The existence of *reference points* (or *join points* [KLM$^+$97]) for the composition process is essential to the coordination of implicit unit interaction. Although other reference points are possible (see [OT98] and [KLM$^+$97]), we restrict reference points to be methods or attributes. *Correspondence*[4] refers to the specification of reference points that are related, e. g. two methods in different units might correspond if they have the same signature. *Unification*, on the other hand, refers to the process by which corresponding constructs are combined. For example, two corresponding methods might be unified by selecting one of them to override the other.

Many correspondence and unification schemes are possible which can be combined in different ways. Only few of them will prove useful, however. We have identified three correspondence schemes in the composition models.

- **Correspondence by name**: This simple form of correspondence uses feature names to specify correspondence. Many proposals do not only use the name of a feature, but its full signature. Nevertheless, we do not want to focus on technical details like signatures, overloading or co-/contravariant parameter redefinition, so we refer to all these correspondence variants as name correspondence.

- **Correspondence by declaration**: Correspondence of different features is explicitly declared in a *correspondence expression.*

- **Correspondence by selection**: The correspondence expression specifies a rule by which features are selected to correspond, e. g. all features that return an integer or whose name starts with "get" correspond.

Inheritance, object composition, mixins, Darwin, JPerspectives and personalities are based on correspondence by name. Correspondence in aspectual components and AspectJ is specified by declaration or by selection. Hyper/J offers a combination of all three strategies: A general strategy, e. g. correspondence by name, is selected and exceptions to, or specializations of this strategy are defined for cases where the general strategy does not apply. Note that name

[4]Some terminology in this subsection has been adopted from [OKK$^+$96] and [SD99].

correspondence is the only strategy where no correspondence expression is necessary.

The most important unification strategies are:

- **Merge unification**: Two corresponding attributes are actually the same attribute in a composite unit and corresponding methods are executed sequentially in some order when any one of them is requested. Special attention has to be paid to the generation of unique return results. Hyper/J is the only model with special support for this strategy.

- **Event unification**: This is a restricted variant of merge unification for methods without return values. One of the methods is selected to be the *trigger method*. When this method is called, all other methods are executed sequentially in some order. When any other method is called, only this method is executed. Change propagation in JPerspectives (p. **??**) uses event unification.

- **Override unification**: One of the corresponding methods is selected to override all other methods, which means that the selected method is executed whenever any of the corresponding methods is requested. The selected method may or may not call the other methods. In the first case, the other methods are called either directly (e. g. C++ or Eiffel inheritance, Darwin, PO) or the methods are put in a linear order and the methods can refer to the next method by a special identifier (`call-next-method` in CLOS, `inner` in Beta, `super` in Java, MixedJava and JPerspectives). In some cases, the call to the next method is mandatory, either by mandatory explicit calls (`expected` in Aspectual Components) or by implicit calls (`before/after` advices in AspectJ). In other cases, the method writer is not even aware of the correspondence of that method with other method, e. g. in Hyper/J.

All strategies require some kind of order among the units. Merge and event unification require a linear order, override unification require at least a "maximal" unit. An order on the units can be created by the following order creation strategies:

- **Structural order**: The structure of the composite units induces a linear (mixins) or a partial (inheritance, Darwin) order. In the latter case, the partial order is converted to a linear order, ambiguities (units not ordered by the partial order) usually have to be resolved by sophisticated language features (e. g. feature renaming mandatory overriding in Eiffel inheritance), or their order is undetermined.

- **Explicit partial order**: The default order is undetermined, but special order hints can be specified (AC, AspectJ, Hyper/J). The generated linear order takes the partial order determined by the order hints into consideration.

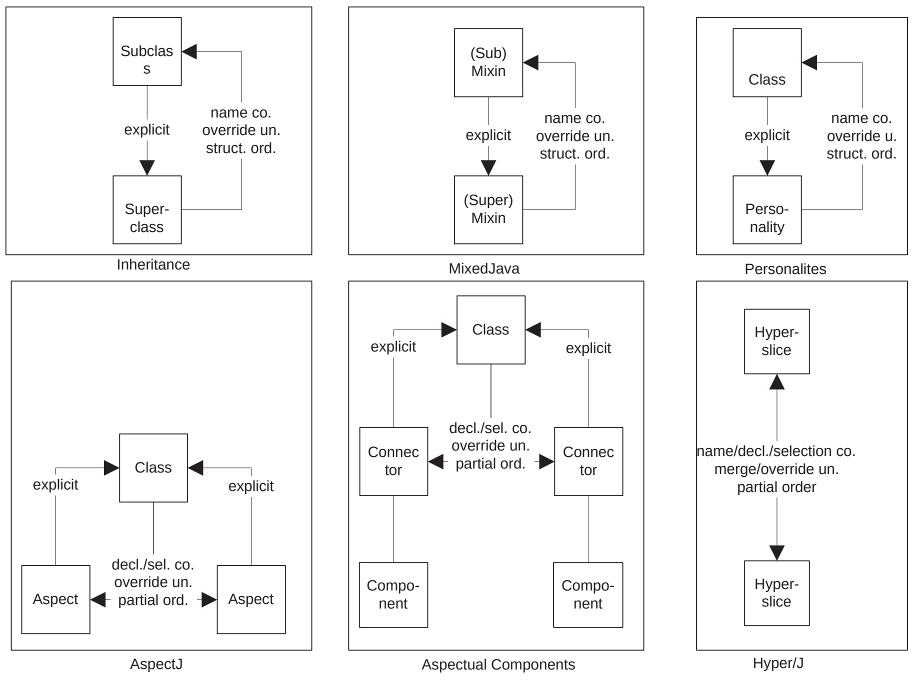

Figure 4: Interaction structure part 1

- **Undetermined order**: The order is completely undetermined, e. g. the order in which perspectives are notified about state changes in JPerspectives.

In the second and in the third case, we have the problem of *undeterministic behavior*. This is undesirable because the actual behavior may depend on the compiler, the platform or other circumstances. We can only advise the programmer not to write code that depends on the actual order, but this may be hard to achieve and is not verifiable by the compiler.

## References

[Car88] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the ACM conference on Principles of programming languages*, pages 70–79, 1988.

[JR92] Paul Johnson and Ceri Rees. Reusability through fine-grain inheritance. *Software – Practice and Experience*, 22(12):1049–1068, December 1992.

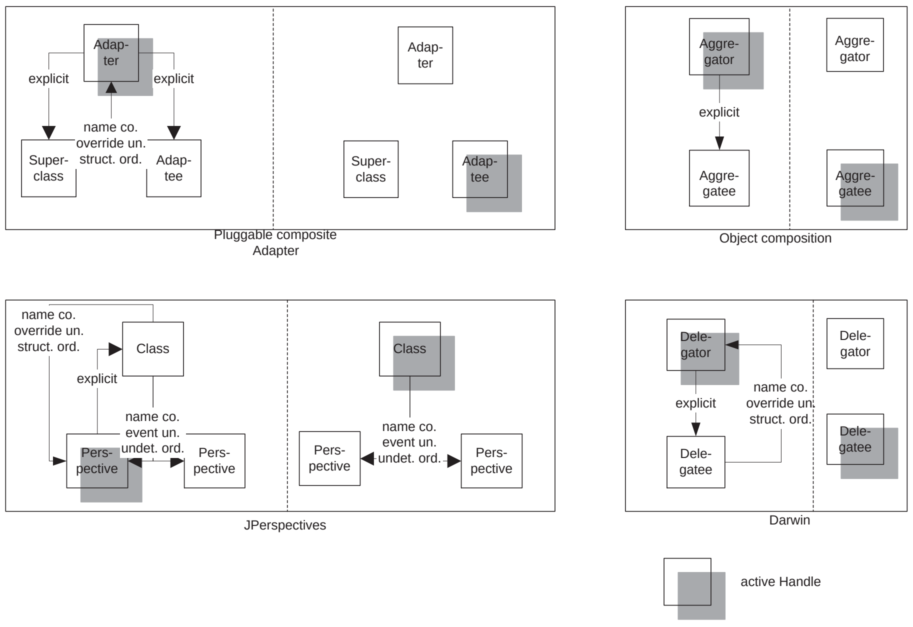

Figure 5: Interaction structure part 2

[KL92] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *Proceedings OOPSLA '92*, volume 27 of *ACM SIGPLAN Notices*, pages 435–451, 1992.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyvaskyla, Finland, 1997. Springer-Verlag.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Mag91] Boris Magnusson. Position statement during the ecoop '91 workshop on types. Geneva, Switzerland, July 1991.

[OKK$^+$96] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–292, 1996.

[OT98] Harold Ossher and Peri Tarr. Operation-level composition: A case in (join) point. In *ECOOP '98 Workshop on Aspect-Oriented Programming*, 1998.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[RTL91] Rajendra K. Raj, Ewan Tempero, and Henry K. Levy. Emerald: A general-purpose programming language. *Software – Practice and Experience*, 21(1):91–118, 1991.

[Sch98] Jürgen Schlegelmilch. Class and type hierarchies: Extension, constraining, and roles. Rostocker Informatik-Berichte 21, Universität Rostock Fachbereich Informatik, 1998.

[SD99] Mark Skipper and Sophia Drossopoulou. Formalising composition-oriented programming. In *Proceedings of the aspect-oriented programming workshop at ECOOP 99*, 1999.

[Szy96] Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings 19th Australian Computer Science Conference*. Australian Computer Science Communications, 1996.