A Generalization and Solution to the Common Ancestor Dilemma Problem in Delegation-Based Object Systems

Eddy Truyen[1] Wouter Joosen[1] Bo Nørregaard Jørgensen[2] Pierre Verbaeten[1]

[1]Department of Computer Science
K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven Belgium
email: [eddy,wouter,pv]@cs.kuleuven.ac.be

[2]Maersk Institute of Production Technology
Southern University of Denmark
DK-5230 Odense M, Denmark
email: bnj@mip.sdu.dk

Abstract

This paper studies the diamond problem in the context of delegation-based object systems. The diamond problem occurs when the same ancestor is inherited multiple times via different inheritance paths. The challenge is that replication and sharing of distinct attributes of the common ancestor must be simultaneously supported. We illustrate the relevance of the diamond problem by showing that it arises not only in multiple inheritance but also in other inheritance techniques, hence the more general term 'the common ancestor dilemma'. More specifically the hybrid approach that integrates object-based inheritance in a class-based model is also affected by the problem. We show that the hybrid approach provides an elegant solution for orthogonal expression of replication and sharing. Attributes that should be shared are modeled as part of the delegating objects, whereas attributes that should not be shared are modeled as part of subobjects of the delegating objects.

1 Introduction

The diamond problem[20], also known as "fork-join" inheritance[18], is a troublesome situation with multiple inheritance which occurs when the same ancestor is inherited multiple times via different inheritance paths, i.e. when two or more ancestors of a class D have a common ancestor A. The question arises whether the attributes (from the common ancestor A) should be inherited in as many versions as there are components deriving from it, or in a single version shared by all components. As argued by [7], both *replication* (i.e. inheriting multiple times) of the meaning of certain attributes and *sharing* (i.e. inheriting once) of the meaning of some other attributes should be simultaneously supported.

We study the diamond problem in the context of delegation-based aspects. Since delegation supports composition at the level of objects, it provides a simple technology for dynamic aspects. Dynamic aspect-orientation is looked upon in this paper as adding or removing aspects to an already running application. A running application consists of a group of collaborating objects that interact with each other by sending messages. An aspect injects components into one or more of these objects and within each object the corresponding component adds new behavior for one or more operations of that object. Examples of delegation-based aspect technology are JAC[16], Delegation Layers[15], Object Teams[5] and Lasagne [23].

This paper focuses on intra-object composition: we look at the composition of components in one object. Each component stems from a different aspect and the different components are composed by placing them in an incremental modification hierarchy, very similar to a linear mixin-based inheritance hierarchy. This paper addresses the following issues:

1. We discuss the scope of the diamond problem. Mira Mezini referred to the diamond problem in her dissertation[11] as the *common ancestor dilemma* problem. We propose to use this name for the diamond problem because we argue it is a more general problem that does not only arise with multiple inheritance but in every inheritance technique that supports *composition of independently developed components*. Here the sharing versus replication issue will arise in any composition of independently developed components that inherit from a common ancestor, hence the more general name the 'common ancestor dilemma'. Specifically, in aspect-oriented programming when two aspects extend (by means of any available incremental modification relationship) a common aspect, their composition will obviously face the same problem.

2. We highlight the limitations of existing solutions to the common ancestor dilemma problem in the context of *delegation*.

3. We illustrate the strength of hybrid models that integrate *delegation* in a class based programming model. Recent work[6, 2] has elaborated on such an approach. We show that the hybrid approach naturally provides an elegant solution for expressing replication and sharing. As such this solution applies to any delegation based aspect-oriented technology.

4. We document the challenge of separating replicated methods. As argued by [11] replicated attributes must be kept separated from each other in different visibility scopes. Although the hybrid approach effectively resolves ordinary name collisions, it fails to separate replicated methods. Different solutions to this problem exist. We discuss the strengths and weaknesses of each of these solutions.

As will be discussed in Section 2 all existing solutions to the original diamond problem incur problems. It is well known that these problems arise because inheritance and visibility control (for the sake of encapsulation) are not orthogonally realized from each other in the design of contemporary programming languages. This lack of orthogonality appears both at the language run-time level and at the programming level. At the language run-time level, the execution environment does not maintain sufficient information to keep the mechanisms apart from each other. At the programming level, wrong programming abstractions are provided so that inheritance and visibility control cannot be expressed in isolation from each other. The existence of this lack of separation of concerns is already well-documented, see for example [13] and [12]. In the line of these thoughts, this paper looks upon the issue to which extent the hybrid approach succeeds in orthogonalizing inheritance and visibility control.

The paper is structured as follows. Section 2 overviews the original diamond problem in the context of multiple inheritance. Section 3 discusses the more general form of the problem and, therefore, proves the relevance of the problem in the context of broad software composition technologies. Section 4 studies the common ancestor problem in the light of delegation and points out why existing solutions needs to be restudied. Section 5 shows how the hybrid approach provides an elegant solution for orthogonal expression of replication and sharing. Section 6 discusses the problem of keeping replicated attributes separated in distinct visibility scopes and discusses the existing solutions to the problem. Section 7 summarizes the paper.

2 The common ancestor dilemma

As shown by [20] the common ancestor dilemma occurs in multiple inheritance when the same ancestor is inherited multiple times via different inheritance paths, i.e. when two or more ancestors of a class D have a common ancestor A. As demonstrated in [7] it is desirable to be able to choose the alternative (replication versus sharing) individually for each attribute. Figure 1 (due to [7, 21]) illustrates this point. When looking at the attributes `name`, `address` and `seniority` in the common ancestor `UniversityEmployee`, there is no doubt that the attributes `name` and `address` should be shared by the `Lecturer` and `AdministrativeStaff` classes. What about `seniority` then? The employee in question has two seniorities, one for each sort of employment. Therefore, the attribute `seniority` should be duplicated in the `Lecturer` and `AdministrativeStaff` classes.

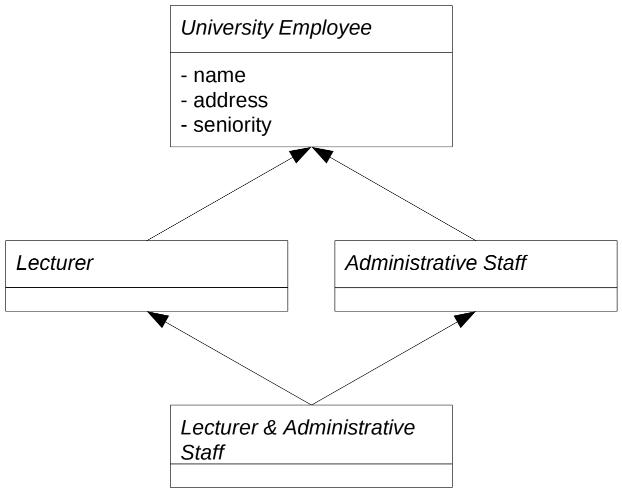

Figure 1. Common ancestor dilemma

Different solutions to the common ancestor problem in multiple inheritance hierarchies have been proposed. None of the approaches is fully satisfactory however (due to [21, 11]):

- graph multiple inheritance: suffers from encapsulation problems and from the undesired duplicate parent operation [20]
- linear multiple inheritance: all replication is simply not allowed to occur.
- tree multiple inheritance: only supports replication [7]

A more general technique is to use renaming[10]. Those attributes that must be replicated are renamed so that there are no name conflicts: those inherited attributes shall be shared that have not been renamed along any of the inheritance paths[18].

Sakkinen[18] refutes all of the above approaches because the common ancestor `UniversityEmployee` gets *effectively split into two*. The problem with this, according to Sakkinen, is that the *integrity* of the independently developed components `Lecturer` and `AdministrativeStaff` *is violated*. Sakkinen defines ’integrity’ as the requirement that no property of an object must be changed except by operations that intend and have the right to modify that object.

Sakkinen notes that the "mathematical difference" (i.e. the incremental modification) of a subclass object and a corresponding superclass object is not defined in the conventional inheritance view (i.e. it is embedded directly in the subclass), or in any case it is not an object. To remedy this situation, Sakkinen proposes an inheritance model that is reduced to aggregation, yielding an inheritance model in which the difference will always be an object; but these objects cannot exist alone, only as part of *complex objects*. He shows that this leads to an inheritance model with much less ambiguous concepts[18].

If we translate the example of Figure 1 into Sakkinen's inheritance model, the `Lecturer&AdministrativeStaff` class is represented by a complex object that aggregates three *subobjects* which are respectively instances of `Lecturer`, `AdministrativeStaff` and `UniversityEmployee`. Returning to the common ancestor problem: in order to accommodate the desired application semantics the `UniversityEmployee` subobject must be split into two: an *S* part that corresponds with the shared attributes and an *R* part that corresponds with the replicated attributes. Figure 2 illustrates this. The integrity of subobjects is thus violated according to Sakinnen. Indeed, if an operation of any of the classes involved ([UniversityEmployee], [Lecturer] or [AdministrativeStaff]) updates shared attributes, based on the value of a replicated attribute, then invariants or assumptions that have been made about the children ([Lecturer] and [AdministrativeStaff]) may break. In other words all operations must be checked in order to identify and resolve such problems; thus the advantage of inheritance is lost.

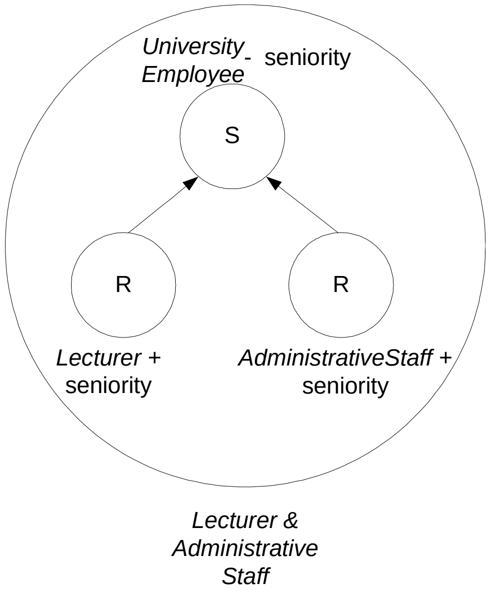
University Employee - seniority
S
R
R
Lecturer + seniority
AdministrativeStaff + seniority
Lecturer & Administrative Staff

Figure 2. Splitting the common ancestor

In order to circumvent integrity violation Sakkinen argues that the application designer must explicitly divide the common ancestor class into two classes in the first place: `Person` (containing `name` and `age`), and `Only-UniversityEmployee`(=`UniversityEmployee-Person`, containing the attribute `seniority`). `Person` is then a shared parent of `Only-UniversityEmployee`. `Lecturer` and `AdministrativeStaff` would inherit without sharing from `Only-UniversityEmployee` [18]. Figure 3 illustrates this. Although this is a clean approach, it puts a burden on the application designer because it implies that the common ancestor dilemma must be anticipated at class design time.

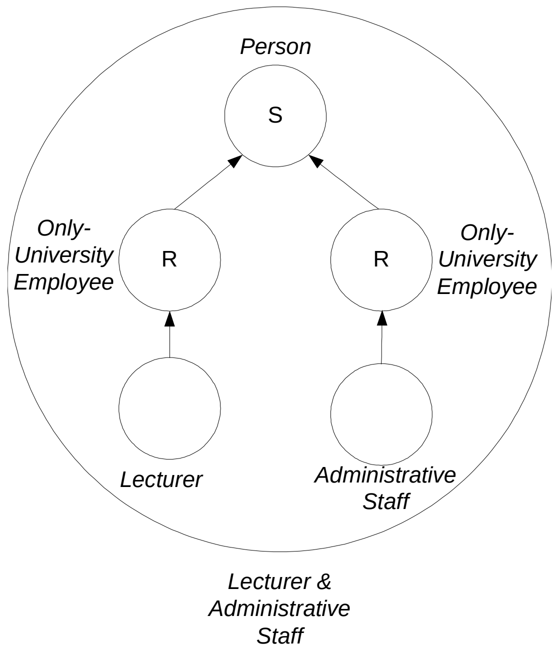

Figure 3. Splitting the common ancestor during class design

3 Generalization of the problem

The existing literature we have studied [7, 18, 21, 11, 19] indicates that the diamond problem is very difficult to solve for state attributes. As such, it is tempting to ignore multiple inheritance: given the fact that the common ancestor problem seemingly only appears with multiple inheritance, the problem could simply be side-stepped by discarding multiple inheritance as a useful software composition tool because it inherently suffers from implementation problems[3, 22] and conceptual problems[19] anyway. This would however be the wrong thing to do as argued by the following points.

We observe that the common ancestor dilemma appears with *any inheritance approach that supports composition of independently developed components*. Indeed, if two independently developed components, *that inherit from a common ancestor*, are composed (by either multiple inheritance, mixin-based inheritance[1], or any other eligible inheritance technique), the problem arises. For example, suppose it was possible to specify explicit inheritance relationships between mixin-classes, then mixin-based inheritance would also have to deal with the problem. Figure 4 illustrates this. Consequently the original diamond problem, identified in multiple inheritance, is an instance of the more general ’common ancestor dilemma’ problem.

Another, but less important point is the usefulness of *repeated inheritance* as a compositional tool. As indicated by [10], inheritance of the same ancestor via different paths is a generalization of repeated inheritance, where the same parent class is directly inherited multiple times. To better distinguish the two cases, the former case is also called *indirect repeated inheritance*, while the latter *direct repeated inheritance*. Direct repeated inheritance is clearly useful as a compositional tool. For example, one class can be explicitly inherited twice to implement two similar, but distinct features at the object level (for example a student that is also an employee at our university has two `MemberID` attributes; Both of the attributes may be instantiated from the same class, but their respective values are necessarily different[21]). Consequently we cannot ignore the common ancestor dilemma problem because otherwise the usefulness of direct repeated inheritance would be wasted.

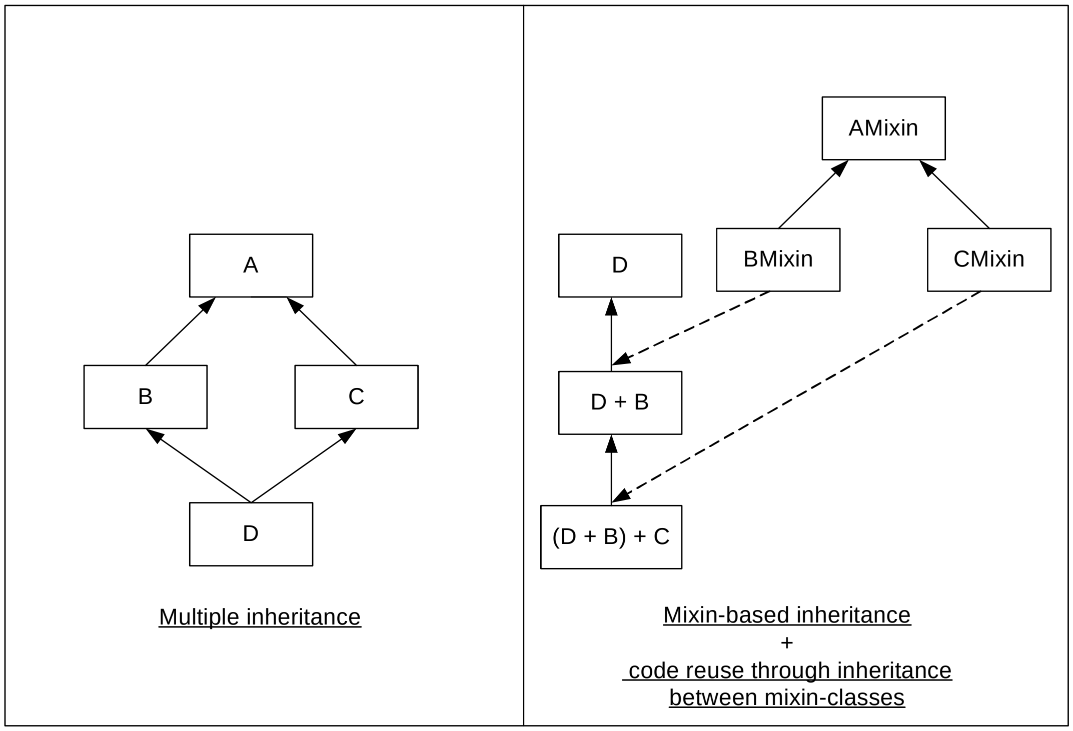

Figure 4. Diamond problem and common ancestor dilemma problem

The above two points therefore prove the relevance of the common ancestor dilemma problem in the context of a broad range of software composition technologies. In fact, the problem also arises in delegation-based object systems. The next section discusses this in further detail and explains why solutions to the common ancestor dilemma problem must be revisited in the light of delegation and, therefore, dynamic aspects.

4 The common ancestor dilemma in the context of delegation

This section studies how the common ancestor dilemma arises in programming languages that support delegation. We will show that the problem of integrity violation, as pointed out by Sakkinen, is irrelevant in the light of delegation, because it is superseded by another problem. Hence, we investigate what is a good solution to the common ancestor dilemma problem in the light of delegation. Before we proceed with the discussion, we first introduce delegation and explain the kinds of delegation that are relevant in the context of aspect-orientation.

4.1 Delegation

Delegation was originally introduced by Lieberman[8] in the framework of a classless prototype-based language. Delegation allows the behavior of an object to be defined in terms of the behavior of another object. An object, called the *child*, may have modifiable references to other objects, called its *parents*. A message for which the receiving object has no matching method are automatically forwarded to one of its parents, that responds on behalf of the receiver. When a suitable method is found in the parent object (the *method holder*) it is executed after binding its implicit *self* parameter. This parameter refers to the message receiver on whose behalf the method is executed. Automatic forwarding with binding of self to the message receiver is called *delegation*. Automatic forwarding with binding of self to the method holder is called *consultation*[6].

There are two forms of delegation. Static and dynamic delegation. In dynamic delegation the parent of an object can dynamically change. Here the parent of an object is typically stored in some specially identified instance variable. This instance variable can be consulted and also modified, thus changing an object's parent. With static delegation, the parent object must be assigned when the object is created and cannot be reassigned during the object's lifetime. Furthermore, with multiple delegation a child object can have multiple parent objects, whereas with single delegation a child object can only have one parent object.

Delegation can also be interpreted as an incremental modification mechanisms and, therefore, delegation has also been called *object-based inheritance*[21]. Child and parent respectively correspond with inheriting client and ancestor.

4.1.1 Hybrid approaches

Delegation has recently regained a lot of interest as part of a *hybrid approach* that integrates delegation in a class-based model. The hybrid approach has regained interest because of its powerful, yet type safe use in the context of class-based programming languages, demonstrated by Lava[6], and support for type transparency, demonstrated by the Generic Wrappers approach[2].

To illustrate the concepts in the remainder of this paper as concretely as possible, we will use a concrete programming model that integrates delegation with the class-based programming model. With this end in view we take the programming model of the Generic Wrappers approach [2] which we will shortly overview here.

Parent and child objects are declared as normal classes. The parent object and each of its child objects may be associated to a separate (class-based) inheritance hierarchy. For example:

```
public class DeclaredParent {
   public void b();
}

public class Parent extends DeclaredParent {
   public void b() {
      ...
      super.b();
      ...
   }

  public void foo() { ... }
}
```

Child objects are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. The wrapped instance is called the *wrappee*. Like an `extends` clause to specify a superclass, a `wraps` clause is used to state the static wrappee type. This also declares the wrapper class to be a subtype of the static wrappee type. For example the declaration

```
public class Child wraps DeclaredParent {}
```

states that each instance of the class `Child` wraps an instance of a class DeclaredParent or of any subtype thereof. The declaration makes Child a subtype of DeclaredParent. Thus, instances of Child can be assigned to variables of type DeclaredParent and Child has all public members of DeclaredParent.

To assure that this subtyping relationship always holds (and thereby that forwarding of calls never fails) instances of Child must always wrap an instance of DeclaredParent or a subtype thereof - already during the execution of constructors. Hence the wrappee must be passed as a special argument (in the syntax of [2] by `< >`) to class instance creation expressions

```
Parent p = new Parent(...);
DeclaredParent c = new Child(...)<p>;
```

Figure 5 shows a class diagram[1] that graphically represents the program listed above where delegation links correspond with the `wraps` clauses and inheritance links correspond with the `extends` clause.

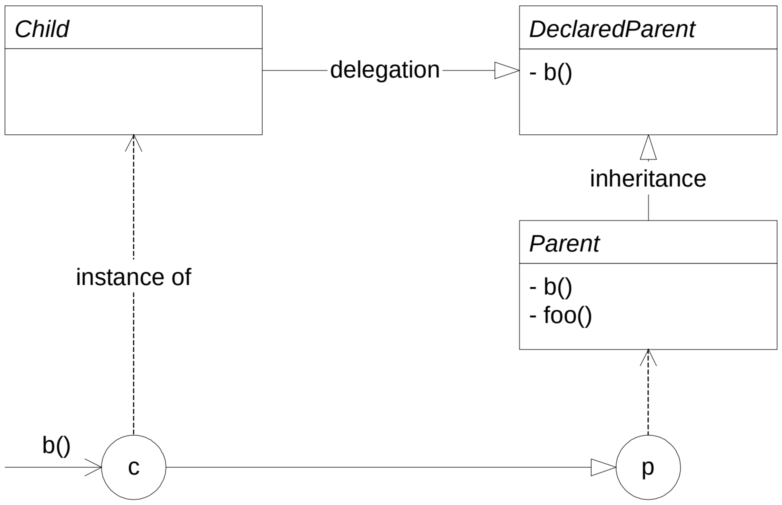

Figure 5. Delegation in a class-based programming model

Delegation is illustrated by the fact that method `b()`, declared in `DeclaredParent`, can be called on the `Child` object. This is illustrated in the following program fragment, which is based on the program listed above. Furthermore, since delegation enables late binding of self, the `b()` method of `Parent` is actually executed:

```
c.b();
```

A particularity of the hybrid approach is its support for type transparency. This means that child objects are not only of the static, but also of the actual wrappee type. For example, a `Child` object wrapping a `Parent` object is also of the latter type and not just of type `DeclaredParent`. Hence, such an aggregate can be assigned to a variable of type `Parent` and the latter's methods can be called on it. In the following program fragment, which is based on the definition of `Child` above, the type test returns true and the cast succeeds:

```
if (c instanceof Parent) {
  ((Parent)c).foo();
}
```

4.1.2 Delegation and dynamic aspects

The hybrid approach has especially gained a lot of interest as a simple technique for dynamic composition of aspects. An aspect could be modeled as a set of child classes. An aspect can then

[1] The graphical notation of the figure is due to [6].

be injected in an already running application by placing instances of these child classes around different application objects that play the role of parent object. Furthermore, there already exist various approaches that lift delegation to real dynamic object modification. For instance, approaches such as JAC[16], Delegation Layers[15], Object Teams[5] and Lasagne [23] free the programmer from having to manually interpose a child object around a parent object through explicit object reference switching.

Given the subject of this paper, we only focus on intra-object composition: the composition of multiple child objects around a single parent object. Each child object stems from a different aspect and the different child objects are composed by placing them in a linear incremental modification hierarchy (also known as conjunctive wrapping), very similar to mixin-based inheritance hierarchies [1]. Of course this statement indicates that the topic of this paper is a general language design issue that has only marginally to do with aspects. However, when looking through an aspect-oriented lens, the relevant design space of delegation-based systems becomes considerably smaller. First, we do not regard delegation in the arena of conceptual modeling but as a tool for composing independently developed components. In other words we study delegation as a composition operator, not as a specialization or an "is-a" relationship. Second, this paper takes single delegation as the basic intra-object composition operator because every child object is meant to extend only one parent object.

4.2 Revisiting the common ancestor dilemma problem

Single delegation also has to deal with the common ancestor dilemma. When a child and a parent object are composed by means of delegation, the common ancestor dilemma arises in one of the following two cases as illustrated in Figure 6:

- **(a)** inheritance from a common declared superclass
- **(b)** delegation to a common declared super type

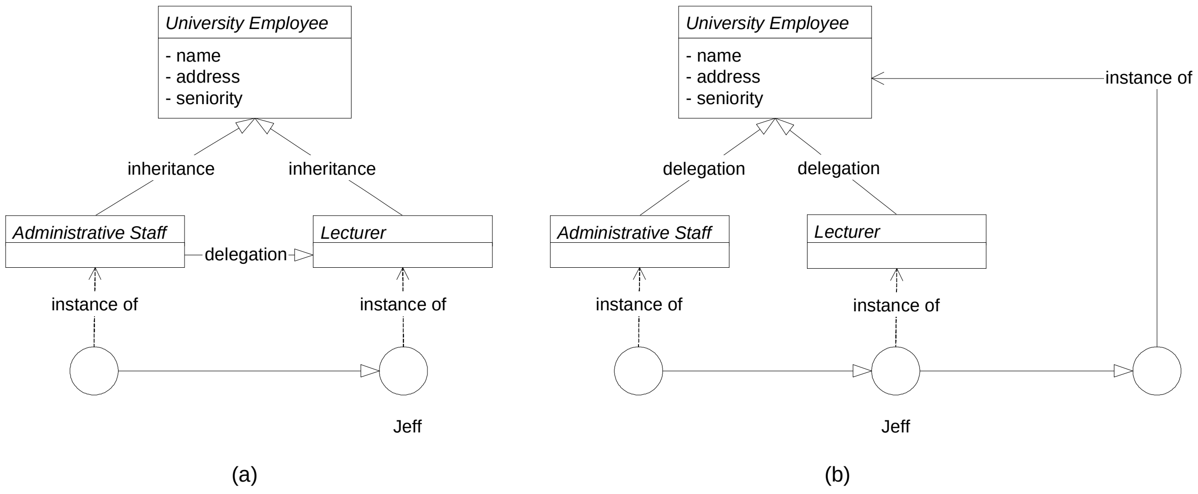

Figure 6. Delegation and the common ancestor dilemma

Since we studied the common ancestor dilemma in the context of Sakkinen's inheritance model[18] (see Section 1), we first have to map Sakkinen's model to delegation. This is quite easy to do because Sakkinen reduces inheritance to aggregation: the notion of complex object corresponds with the dynamically bound common self across the web of parent and its child objects. A subobject in that complex object corresponds with the parent or one of its child objects.

We will now explain how the problem focus is shifted in the light of delegation. Suppose Jeff who is a `Lecturer` is asked to handle some administrative task as well. To accommodate this situation in real-time, a reconfiguration must take place at run-time: the complex object representing Jeff must be dynamically extended with an `AdministrativeStaff` subobject. As argued in Section 1 it is desirable to be able to choose between replication and sharing individually for each attribute of the common ancestor `UniversityEmployee`. In case (a), however, replication is the obligatory default for all the common ancestor's attributes, while sharing is in case (b). In case (a) Jeff would have duplicate name and address attributes which is undesirable from a conceptual modelling standpoint (the name and home address of a person are conceptually unique) and from a state consistency standpoint (clients accessing different subobjects must observe and modify (through getters and setters methods) the accidentally duplicated attributes in a mutually consistent fashion). In case (b) Jeff would have the same seniority for both sorts of employment which is obviously undesired from a conceptual modelling standpoint. So also here, the `UniversityEmployee` subobject of Jeff needs to be split into two.

However, splitting of the common ancestor simply *cannot* be performed because the composition operator provided by delegation operates at run-time, at the level of *operational* subobjects. Technically speaking, splitting a subobject after creation (i.e. at run time) is not feasible.

As such, when already running application objects (complex objects in the terminology of Sakinnen) are to be modified over time with subsequent new components, the common ancestor dilemma may strike at any point of time when any pair of two components share a common ancestor. If the dilemma occurs, however, then the common ancestor has been instantiated already. The ancestor therefore cannot be split anymore.

5 Towards an elegant solution

We consider three alternative approaches to address the aforementioned problem.

The first approach would address a solution that disallows specifying any explicit inheritance relationship. The idea is that by disallowing specification of explicit inheritance relationships the common ancestor dilemma will not occur in the first place and as such no solution would be required. Pure mixin-based inheritance (where explicit inheritance relationships between mixin classes is not allowed) is an example. However this is a fake solution because, as argued in Section 3, direct repeated inheritance faces the same issues of the common ancestor dilemma. As such the problem still needs to be solved in order to gain the expressive power of direct repeated inheritance.

A second approach would be to advance the technological state-of-the-art in virtual machine support to allow splitting subobjects while the complex object is running. We think working towards this goal is neither realistic nor a good idea. First, there are the problems related to integrity violation as mentioned by Sakinnen. Secondly, although there exists work about run-time support for changing classes [9] or adding aspectual behavior during execution[17], splitting classes at run-time seems extremely difficult to do without incurring a lot of other problems.

The third and only option left for dealing with the common ancestor dilemma in the light of delegation is the original solution from [18]: namely to side-step the splitting problem by explicitly dividing the common ancestor into two classes *S* and *R* during software design. The class *S* contains attributes to be shared and the class *R* is to be replicated (see Figure 3). What is obviously needed is the expressive power that enables the modelling of such an ancestor structure.

This is where the power of the hybrid approach comes into play. It provides a natural solution for respectively expressing sharing and replication without interfering with each other. Sharing is realized by means of delegating to the *S* subobject, while replication is realized by means of inheriting the *R* class. The code below (based on the programming model introduced in

Section 4.1.1) illustrates how a structure similar to Figure 3 can be easily implemented: class `UniversityEmployee` is effectively split in two classes during class design: class `Person` encodes the *S* part and `Only-UniversityEmployee` encodes the *R* part. Since university employees are persons, class `Only-UniversityEmployee` obviously needs to extend the Person class. Since the `Person` class represents the *S* part, this incremental modification should be expressed by means of delegation.

```
public class Person {
   private Name name;
   private Address address;

   public String getName() {...}
   public Address getAddress() {...}

}

public class Only-UniversityEmployee wraps Person{
   private Seniority seniority;

   public Seniority getSeniority() {...}

}
```

The classes, representing different forms of employment, however, need to be defined as an incremental modification by means of class-based inheritance because `Only-UniversityEmployee` needs to be replicated.

```
public class Lecturer extends Only-UniversityEmployee {
   String title;

   public Lecturer(String title) {
     this.title = title
   }

   public String getName() {
      return title + super.getName();
   }

  public void foo() {...}
       self.getSeniority();
  }
}

public class AdministrativeStaff extends Only-UniversityEmployee {
   String jobtitle;

   public void bar() {...}
       self.getSeniority();
  }

  public String getName() {
    return super.getName() + “, ” + jobtitle;
}
```

Finally, the scenario in Section 4.2 that `Lecturer` Jeff is suddenly employed as `AdministrativeStaff` can easily be accommodated by wrapping the jeff object and reassigning the result to the `jeff` variable.

```
//main
Only-UniversityEmployee jeff = new Lecturer("Prof. dr".)<new
Person(...)>;
```

```
...
jeff = new AdministrativeStaff(...)<jeff>;
((Lecturer)jeff)foo();
```

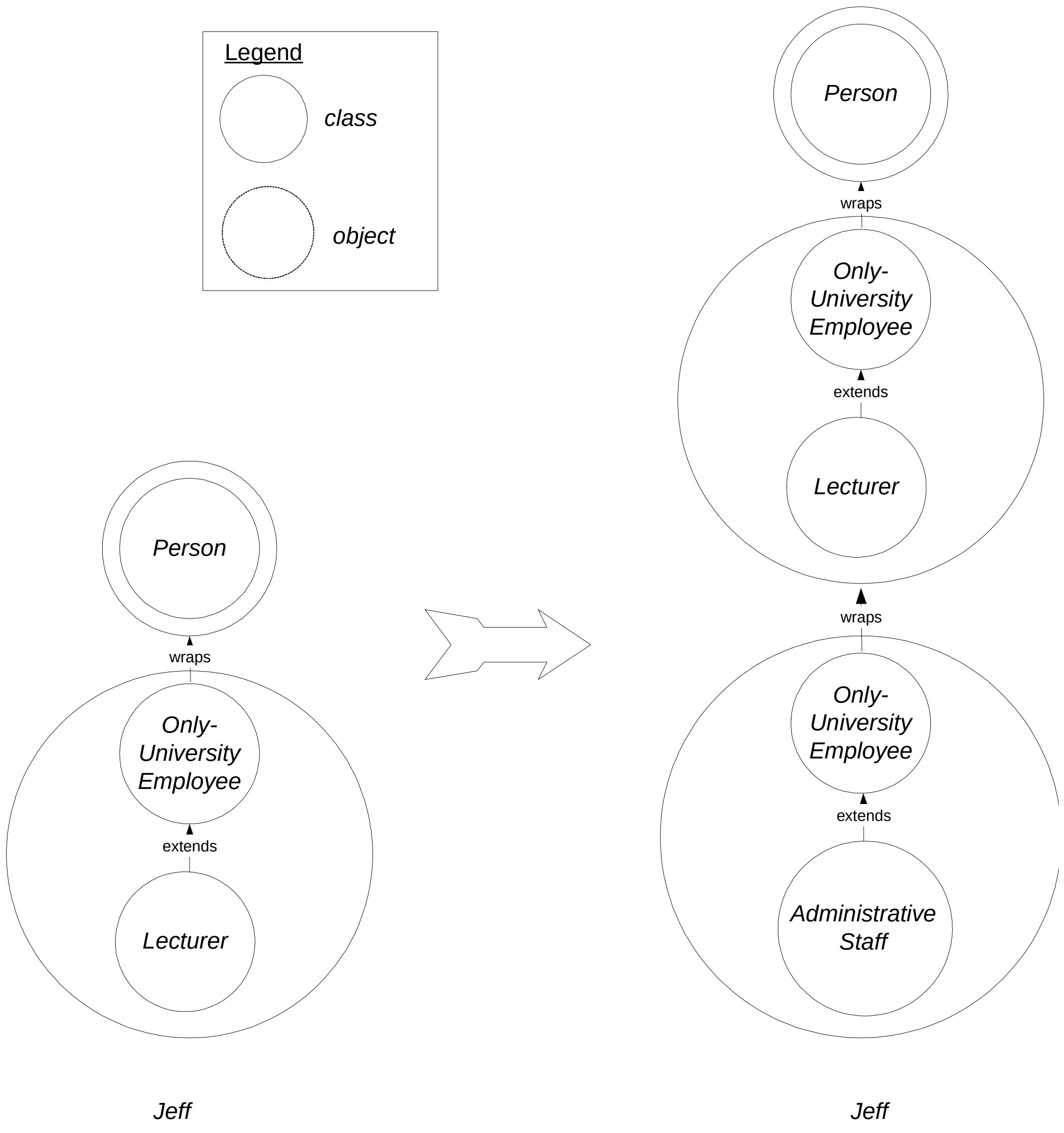

Figure 7. Jeff becomes administrative staff

This is further illustrated in Figure 7. Before wrapping it with `AdministrativeStaff`, complex object `jeff` consists of an object hierarchy of which the `Lecturer` subobject encloses an `Only-UniversityEmployee` subobject. After wrapping complex object `jeff`, both `Lecturer` and `AdministrativeStaff` subobjects contain a duplicate of the `Only-UniversityEmployee` class.

6 Keeping replicated attributes separated

Replicated attributes must be kept separated from each other because they are considered to belong to two different subobjects of a single complex object. Coping with this problem equals at first sight to the problem of coping with name collisions in general that occur when two independently developed components accidentally use the same name for different attributes; Attributes involved in such ordinary name collisions are often called *homonymous attributes* [11].

Although the hybrid approach effectively deals with ordinary name collisions, it fails to separate replicated methods as will be explained. This section subsequently discusses various existing solutions to this particular problem.

6.1 The problem

In the hybrid approach replicated state variables can easily and effectively be kept separated from each other if they are declared as non-public attributes. Dealing with replicated methods is more difficult as will be explained below.

To deal with homonymous methods in the context of delegation, Günter Kniesel proposes the following adapted rule for method overriding[6]:

> For a message *recv.n*(*args*) (i.e. `self.getSeniority()`) a method with signature σ (i.e. `getSeniority()`) from type *T* (i.e. `AdministrativeStaff` overrides the matching method from the static type of *recv*, T_{stat}(i.e. `Lecturer`) if there is some common declared supertype of *T* and T_{stat} (i.e. `Only-UniversityEmployee`) that contains σ.

The adapted rule for method overriding essentially boils down to the existence of *a common declared supertype*. This exact feature, however, renders the adapted rule completely useless for keeping replicated methods separated from each other. This is because replicated methods are declared by a common ancestor class and, therefore, the adapted rule would incorrectly enable overriding between replicated methods. The adapted method rule does work for homonymous methods because these methods stem from different unrelated ancestors.

Let us apply the adapted rule to the running example to illustrate our point. Consider in Figure 7, the complex object `jeff`, that consists of a `Person` subobject, a `Lecturer` subobject with an enclosed `Only-UniversityEmployee` subobject, and an `AdministrativeStaff` subobject with a second enclosed `Only-UniversityEmployee` subobject. Here overriding between the `Lecturer`-specific and `AdministrativeStaff`-specific methods of the `getSeniority()` operation is enabled according to the adapted rule. This is because there exists a common declared supertype of `Lecturer` and `AdministrativeStaff` (i.e. `Only-UniversityEmployee`) that declares `getSeniority()`. As a result, the self call `to` `getSeniority()` depicted in the above example from within the `Lecturer` subobject will be incorrectly redirected to the `AdministrativeStaff` object. As such it is clear that the adapted rule for method overriding incorrectly enables overriding between replicated methods. Note that the adapted rule also breaks in the case of direct repeated inheritance.

6.2 Existing solutions

This section discusses various solutions to the problem of keeping replicated methods separated in the hybrid approach. Since our solution of expressing replication and sharing is based on a redesign, it is worth to consider other redesign options for maintaining replicated methods in mutually invisible scopes. We also investigate solutions based on additional machinery in programming language. Basically, a good solution makes a good trade-off between providing additional machinery that the average programmer can understand, and creating software designs that are easy to maintain and evolve.

6.2.1 Disjunctive wrapping

An astute reader might notice that replicated methods can easily be kept separated from each other by disjunctively wrapping the `AdministrativeStaff` and `Lecturer` objects (instead of conjunctively wrapping them). One could indeed have one instance of `Lecturer` and one instance of `AdministrativeStaff`, each delegating to a `Person` object, but not to each other. By letting the client have different references “jeffAsLecturer” and “jeffAsAdministrator” to the two delegating objects, one could already achieve the desired separation between replicated methods. This of course also works for self calls because self is dynamically bound to the original message receiver.

This solution is very elegant from a conceptual modeling point of view and is also in line with a frequently occurring situation in role-based design. In the latter it frequently occurs that an object plays different roles in different contexts and within each context the object plays never more than one role. The role an object plays depends thus on the context in which the object is currently being used.

From a compositional point of view, however, disjunctive wrapping does not support combination of methods that override a method of the shared part *S*. In the running example both `Lecturer` and `AdministrativeStaff` override the `getName()` operation of `Person`. In a disjunctive wrapping style it is not possible to invoke the full-combined behavior of both overriding methods.

6.2.2 Replacing class-based inheritance with aggregation

Another solution is to model the *R* part of the common ancestor as a “real aggregated” subobject of the delegating objects. Thus instead of inheriting the *R* part one aggregates the *R* part. Suppose in the running example `Lecturer` and `AdministrativeStaff` would directly delegate to `Person`, whereas the `Only-UniversityEmployee` class is completely independent of the `Person` hierarchy. Then replication could simply be expressed by having the delegating objects aggregate a different `Only-UniversityEmployee` object. Having moved the "seniority feature" to different aggregated subobjects, we obviously do not get any overriding of `getSeniority()` methods any more.

This approach solves the issue of separating replicated methods quiet nicely for self calls. For non-self calls, however, there is the problem of the necessary plumbing that must be manually programmed in order to allow clients access to the appropriate replicated subobject. This also implies that the client has to manually navigate through the delegation hierarchy to find the appropriate subobject he is currently interested in.

6.2.3 Multiple delegation from a proxy object

Letting a surrogate / proxy object multiply delegate to the `Lecturer` and to the `AdministrativeStaff` object (which themselves share the identical `Person` parent) expresses exactly the desired sharing and replication semantics and lets it simply be understood from the shape of the delegation hierarchy as depicted in Figure 3. The desired separation between replicated methods can be easily achieved in multiple delegation by using the “sender path tiebreaker rule", used in the design of the Self programming language[25], or by renaming of selectors[10], used in the design of Lava[24]. Moreover, these solutions nicely complement the disjunctive wrapping style. Combination of methods that override a shared method can be accommodated by explicit local redefinitions at the proxy.

The disadvantage of the approach is that the creation of the proxy class, renaming and explicit local redefinitions puts an extra burden on the programmer. Furthermore, the approach is not scalable enough to cope with the situation that multiple common ancestor dilemmas must be resolved within the same complex object.

6.2.4 Scope identifiers

The Rondo object model[12, 11] effectively supports separating replicated methods by means of a mechanism based on so called *scope identifiers*. This mechanism is uniform in the sense that it resolves both kinds of conflicts (replicated methods as homonymous methods) in identical the same way. Although the Rondo model has not been developed in the context of delegation-based systems, a mapping to delegation is straightforward. Scope identifiers are constructed as follows: each child object is marked with a unique label and the scope identifier of a method simply concatenates the labels of child object that are in the visibility scope of that method.

Although the Rondo model is elegant from a language run-time engineering point of view, the mechanism of scope identifiers does not provide the right abstraction for dealing with name collisions that occur when non-self calls are sent from message-passing clients. This is because the labels of child objects that are used to construct scope identifiers are implicitly generated by the internal structures of the Rondo engine and therefore do not have a meaning in the domain of message-passing clients. As such it seems that although the mechanism of scope identifiers sufficiently applies separation of concerns at the language design space to effectively control distinct visibility scopes for replicated methods, the mechanism does not have a meaning to message-passing clients.

The other solutions discussed above do not suffer from this problem because they model visibility scopes in the domain of the application and, therefore, have a clear meaning in the domain of message-passing clients. We believe that a programming language whose execution environment is based on the Rondo model and whose programming model provides sufficient expressive power to model scope identifiers in the domain of the application is a very powerful solution. Future work in this context is to study to which extent the notion of dependent types[4, 14, 15] is feasible to serve this purpose. The notion of dependent types implies that an object can aggregate one or more inner classes that are virtual, meaning that the type of these inner classes is dependent on the identity and type of the aggregating outer object.

7 Conclusion

This paper has focused on dynamic intra-object composition. We look at the composition of aspects in one object. Each component stems from a different aspect and the different components are composed by placing them in an incremental modification hierarchy, very similar to a linear mixin-based inheritance hierarchy.

We have discussed the scope of the common ancestor dilemma problem from this perspective. Specifically, in aspect-oriented programming when two aspects extend (by means of any available incremental modification relationship) a common aspect, their composition obviously faces a similar problem. We have highlighted the limitations of existing solutions to the common ancestor dilemma problem in the light of dynamic aspects. We have illustrated the strength of hybrid models that integrate delegation in a class based programming model. We have shown that the hybrid approach naturally provides an elegant solution for expressing replication and sharing. As such this solution applies to any delegation-based aspect-oriented technology. We have documented the challenge of separating replicated methods and we have discussed the existing solutions to this problem.

Acknowledgments

Many of the ideas in this paper were inspired by the discussion of the common ancestor dilemma in the dissertations of Mira Mezini[11] and Patrick Steyaert[21]. The structure of the paper was helped along by discussions with Nico Janssens. We would like to thank the anonymous reviewers for their very informative and helpful comments and for pointing out some of the existing solutions to separating replicated methods.

References

[1] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

[2] Martin Büchi and Wolfgang Weck. Generic Wrappers. In Elisa Bertino, editor, *ECOOP 2000, 14th European Conference on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 201–225. Springer-Verlag, 2000.

[3] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA'89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 211–214. ACM Press, 1989.

[4] Erik Ernst. Family polymorphism. In *ECOOP 2001—Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer-Verlag, June 2001.

[5] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer, October 2002.

[6] Günther Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP '99—Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 1999.

[7] Jørgen Lindskov Knudsen. Name Collision in Multiple Classification Hierarchies. In S. Gjessing and K. Nygaard, editors, *Proceedings of the ECOOP '88 European Conference on Object-oriented Programming*, LNCS 322, pages 93–109. Springer Verlag, August 1988.

[8] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA'86*, pages 214–223. ACM SIGPLAN Notices 21(11), 1986.

[9] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *ECOOP 2000 – Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer-Verlag, June 2000.

[10] B. Meyer and N. Wilson. Eiffel: The Reference. Technical Report TR-EI-41/ER, ISE, 1995.

[11] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publishers, 1998.

[12] Mira Mezini. Dynamic object evolution without name collisions. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 190–219. Springer, 1997.

[13] Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[14] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003—Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer-Verlag, July 2003.

[15] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002—Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 2002.

[16] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, September 2001.

[17] A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*. ACM press, March 2003.

[18] Markku Sakkinen. Disciplined Inheritance. In *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 39–56, Nottingham, July 1989. Cambridge University Press.

[19] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behavior. In *ECOOP 2003 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, July 2003.

[20] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, Series in Computer Systems, pages 165–188. The MIT Press, 1987.

[21] P. Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

[22] Peter Sweeney and Joseph Gil. Space-and time-efficient memory layout for multiple inheritance. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10) of *ACM Sigplan Notices*, pages 256–275, N. Y., November 1–5 1999. ACM Press.

[23] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and selective combination of extensions in Component-Based applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, May12–19 2001.

[24] Kniesel, *Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems*, Technical Report IAI-TR-94-3, Computer Science Department III, University of Bonn, Oct. 1994 (revised April 1995).

[25] C. Chambers, D. Ungar, B.-W. Chang and U. Hölzle, "Parents and shared parts of Objects: Inheritance and Encapsulation in SELF," *Lisp and Symbolic Computation: An International Journal,* vol. 4, no. 3, 1991, pp. 21-36.