A survey and problem analysis of advanced object-oriented software composition

Eddy Truyen
Departement Computerwetenschappen, K.U.Leuven
Celestijnenlaan 200A
B-3001 Heverlee

November 25, 2004

1 Introduction

This paper helps to improve the understanding of advanced object-oriented software composition techniques

Advanced software composition has become the locus of attention of current research on software engineering and execution platforms. The investments in component-based software development and aspect-oriented software development are witnesses of this trend. We believe current research on advanced software composition is driven by three key goals:

- support for *dynamic and context-sensitive composition of components*: components are integrated at run-time in a context-sensitive manner.
- support for *unanticipated* changes: new components can be added to the system without provision of pre-planned hooks
- support for composition and evolution of *independently developed components* without producing semantically incorrect programs. Generally, name resolution plays an important role in the emergence and handling of composition and evolution conflicts. This goal can be divided in two subgoals:
  - independent extensibility: independently developed components can be composed without needing a global integrity check.
  - independent evolution: components can be modified over time without coordinating with each other

We specifically put our attention at *object-oriented* composition models because they already provide three *basic mechanisms* by means of which the three key goals could be achieved:

- inheritance: an incremental modification mechanism (in support of unanticipated extensibility)

- encapsulation: specification and enforcement of visibility constraints (in support of independent extensibility and independent evolution)
- object wrapping: dynamic object modification (in support of dynamic and context-sensitive composition)

This paper investigates the three principal object-oriented compositional models in relation to which extent they respectively achieve the three key goals. They are *object-based composition*, *class-based inheritance*, and *object-based inheritance* (also know as *static delegation)*.

What can be stated already is that none of the compositional models satisfy all three key goals simultaneously, nor do they completely satisfy them without introducing problems.

It is well-known that these problems arise because the basic mechanisms are not orthogonalized from each other in the design of contemporary programming languages. This lack of separation of concerns appears as well at the language run-time level as at the language programming level. At the language run-time level, the execution environment does not maintain sufficient information to keep the mechanism apart from each other. At the language programming level, wrong programming abstractions are provided so that different issues cannot be expressed in isolation from each other. The existence of this lack of separation of concerns is already well-documented, see for example [NT95] and [Mez97] that specifically discusses the problem in the context of the above mechanisms.

This paper contains an exhaustive reference list of problems that appear as a symptom of this lack of separation of concerns, i.e. it describes how the three key goals are affected by the lack of separation of concerns at the language design space. We perform a separate problem analysis for each compositional model. After that, the entire problem analysis is repeated for so called *hybrid approaches*. Hybrid approaches propose advanced compositional models that make a combination of the three basic composition models while attempting to remove their respective problems.

2 Object-based composition

In particular we refer to the use of *wrappers*. The use of wrappers is a well-known approach to dynamic composition and adaptation [Höl93]. Well-known design patterns such as Decorator, Adapter [GHJV95] and Role Object[BRSW00] that all support dynamic composition and adaptation are as well based on the use of wrappers. However, using wrappers to customize applications on a large scale has not yet found much success in class-based object-oriented programming and some of the technical challenges connected to context-sensitive customization therefore cannot be met. We analyze these problems in this section.

Before we proceed, we will take a moment to introduce our terminology: we will refer to a wrapped object as a wrappee. The product of combining a wrapper and a wrappee is called an aggregate. An object reference's declared type is referred to as its static type. The type of the actually referenced object is called the object reference's dynamic type. Likewise, we also distinguish between the static and the dynamic wrappee type. The static type of the

wrappee is the class that appears in the wrapper definition. The dynamic type is the actual type of the wrappee.

The wrapper-based approach at first sight supports surprisingly well a lot of the challenges that are involved in supporting dynamic and context-sensitive composition of behavioral extensions:

- Modularity: wrappers support non-invasive integration of behavioral extensions to a minimal core object. Code needed for implementing a collaboration can be completely encapsulated into a set of wrappers that each are enclosed around a different wrappee.

- Run-time customization: the wrapper-based approach enables instance-level composition and therefore run-time customization.

- Closure: Closure means that a wrappee which has been composed with a wrapper can be further composed with other wrappers. This can easily be supported by conjunctive wrapping. Conjunctive (also called additive or recursive) wrapping applies multiple wrappers around each other. For example, we might wrap an object in a wrapper A and the latter with a wrapper B.

- Simultaneous client-specific views: Isolation is supported by disjunctive wrapping. Disjunctive wrapping presents the same object with different wrappers to different clients. For example one client has a reference to wrapper A around an object, while another client has a reference to wrapper B around the same object. As such disjunctive wrapping makes it possible to enforce different client-specific customizations simultaneously without interference.

- Mutual state consistency: the solution to providing mutual state consistency lies in assuming that wrappers define incremental additive changes where all wrapper methods that modify or observe the state of the wrappee must be implemented by calling methods of the wrappee. This condition ensures that modification made via calls to the wrapper are visible to clients of the wrappee, and that modifications made via calls to the wrappee, are visible to users of the wrapper[ALS03]. In order for this to work, however, it is important that the the design of the wrappee is complete in the sense that any useful customization can be defined in terms of the wrappee's public interface[Ode00].

- Ease of use: Ease of use is well supported from a programming perspective since wrapper-based design patterns such as Decorator[GHJV95] and Role Object[BRSW00] are widely known and well-documented. Reflective techniques such as Meta-Object Protocols (MOPs) [Mae87] have known uses for customization, but they are complex and thus difficult to use for application programmers who are typically no experts in meta-programming.

Dynamic and client-specific combination of behavioral extensions could in theory be supported by *disjunctive wrapper chaining* which is joint use of disjunctive wrapping and conjunctive wrapping. This is shown in the left part of figure 1. Several disjunctive wrapper

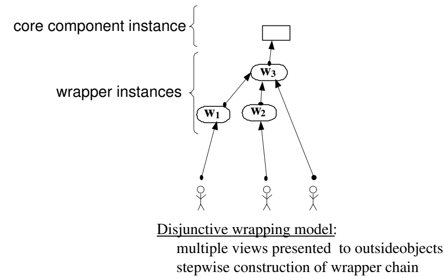

Figure 1: Disjunctive wrapping

chains are constructed around a core object and each client selects one of the wrapper chains that matches best the features desired by that client.

However this approach is completely not feasible using class-based object-oriented programming. This is because one cannot rely on the object identity of wrapped objects anymore. This object identity problem introduces a new set of problems which render disjunctive wrapper chaining completely unfeasible for supporting context-sensitive customization of large applications.

2.1 The spaghetti problem

Consider the following application program consisting of two interfaces:

```
public interface Document {
   public void print();
   public Printer getPrinter();
   public void setPrinter(Printer p);
}

public interface Printer{
   public void print(Document d)
}

public class DocumentImpl implements Document {
   private Printer printer;

   public void print() {
       prepareDocument();
       printer.print(this);
   }

   public Printer getPrinter() {
     return printer;
   }
```

```
    public void setPrinter(Printer p) {
      printer = p;
    }

    ...
}

public class PrinterImpl implements Printer {
  ...
}
```

Suppose the following simple extensions for Printer objects:

- A pretty printing extension
- Another extension A
- Another extension B

Each extension is implemented by a separate wrapper class. For example the pretty printing extension would be implemented according to the Decorator design pattern[GHJV95] as follows:

```
public class PrettyPrinter implements Printer  {
  protected Printer printer;
  protected int counter;

  public PrettyPrinter(Printer p) {
     printer = p;
     counter = 0;
  }

  public void print(Document d) {
     //make document d pretty
     printer.print(d);
     counter = counter + 1;
  }
}
```

The other extensions A and B are implemented by similar wrapper classes, say APrinter and BPrinter.

One of the basic problems with wrapper objects is that they have an object identity of their own. As such a PrettyPrinter wrapper object hides the real Printer object and therefore clients cannot rely anymore on the object identity of the original Printer object. As a consequence, if a client wants to dynamically integrate the PrettyPrinter object around the real Printer object, the client has to manually redirect the reference from the real Printer object to the PrettyPrinter wrapper instance. Maintaining this indirection is tedious and becomes extremely complex in the case of disjunctive wrapper chaining to support selective combination of Printer extensions. For example, consider the following basic client program

```
Document d = new DocumentImpl(); d.print();
```

If this client program wishes to selectively combine some of the existing extensions, it has to rely on conjunctive wrapping around the printer object encapsulated in d. The corresponding wrapper composition logic must be manually implemented within the client program.

```
Document d = new DocumentImpl();
Printer p = d.getPrinter();
Printer desiredPrinter;
if (...)
  desiredPrinter = new PrettyPrinter(p);
else if (...)
  desiredPrinter = new APrinter(new PrettyPrinter(p));
else ...

d.setPrinter(desiredPrinter);
d.print();
```

If different client programs wish to select different combinations of extensions independently from each other, we must rely on disjunctive wrapper chaining. The problem with this is that each disjunctive wrapper chain has an object identity of its own. Client programs must maintain the references to these different chains, since for every method invoked on the Printer object, the client object must select through which wrapper chain the method invocation should go through. Maintaining this indirection is tedious and will lead to severe scalability problems when considering interaction refinement. A wrapper that implements an interaction refinement invokes another object that may also have been wrapped with other wrappers As a consequence, wrappers themselves may need to refer to the different wrapper chains of other objects. This leads to a vast spaghetti of object references Managing this spaghetti in the presence of wrap updates is of exponential complexity.

2.2 Duplicated state

Further note that with disjunctive wrapper chaining, different wrapper chains contain duplicate instances of the same wrapper class. For example, in the above client program there are two PrettyPrinter objects wrapped around the same Printer object p. This becomes a problem when wrappers encapsulate state. For example, the PrettyPrinter wrapper defines a `counter` instance variable to accumulate the number of documents being printed in pretty mode. As a result, we must ensure the consistency and conceptual common identity of the counter variable, which is really difficult to do.

2.3 Factory objects

There is a way to solve these problems, at least partially. The idea is to introduce a level of indirection at every wrapper object construction. Instead of directly creating a wrapper object, every wrapper object is created via a call to a factory object[GHJV95].

By encapsulating the wrapper object creation in the factory object, it becomes possible to make the redirection of object references transparent for the client programs. In this line of thoughts, we could define a factory class for PrettyPrinter wrappers as follows:

```
public interface PrinterFactory {
  public Printer wrap(Printer wrappee);
}

public class PrettyPrinterFactory implements PrinterFactory {
  protected static HashMap PrettyPrinterExtensions = new HashMap();
  protected PrinterFactory next;

  public PrettyPrinterFactory() {
     next = null;
  }

  public PrettyPrinterFactory(PrinterFactory next) {
     this.next = next;
  }

  public Printer wrap(Printer wrappee) {
    Printer result = PrettyPrinterExtensions.get(wrappee);
    if (result == null)  {
       result = new PrettyPrinter();
       PrettyPrinterExtensions.put(wrappee, result);
    }
    if (next == null)
       result.setWrappee(wrappee);
    else
       result.setWrappee(next.wrap(wrappee));
    return result;
  }
}
```

Similar PrinterFactory classes must be created for APrinter and BPrinter wrappers. In order to enable dynamic combination of wrappers, the different PrinterFactory classes must be composable with each other. This is simply achieved by applying conjunctive wrapping to PrinterFactory objects themselves. Each PrinterFactory wrapper has a reference to the next PrinterFactory (the ”next” variable). Every time a PrinterFactory object is requested to create a wrapper for a Printer object p, it creates an instance of its associated wrapper class and wraps it around the result of applying the next PrinterFactory in line to Printer object p.

To avoid state duplication, the factory object canonicalizes wrappers[Höl93]. With canonicalized wrappers, exactly one wrapper exists per wrappee, and thus there is no problem of maintaining duplicate state. Every time the PrinterFactory object is requested to wrap a Printer object, we have to check a global dictionary to see if the object already has a wrapper. If the object has not been wrapped before, we must create a new wrapper and enter it into the dictionary.

The client program now looks slightly simpler. First it has to compose a PrinterFactory object, called desiredPrinterFactory, that reflects a desired combination of wrappers. A Printer object p can then be extended by passing it to the `wrap()` operation of the desired-

PrinterFactory object.

```
PrinterFactory desiredPrinterFactory; if (...)
  destiredPrinterFactory = new PrettyPrinterFactory();
else if (...)
  desiredPrinterFactory = new PrettyPrinterFactory(
                                  new APrinterFactory()
                               );
else ...

Document d = new DocumentImpl();
Printer p = d.getPrinter();
d.setPrinter(desiredPrinterFactory.wrap(p));
d.print();

...

desiredPrinterFactory = new BPrinterFactory();
d.setPrinter(desiredPrinterFactory.wrap(p)); d.print();
```

To compose the desiredPrinterFactory object, we apply in a sense disjunctive wrapper chaining to Factory objects. The question arises then whether we are only shifting the spaghetti reference problem to the Factory object level. Not really, because factory objects are all stateless; the only state involved - the wrapper dictionaries - are static variables that are stored in the corresponding Factory class definition. As a result it is perfectly possible to create multiple instances of the same Factory class in different wrapper chains without having to worry about duplicate state. As a second consequence, we don't need to recycle Factory objects in the way is necessary with normal wrappers. Once we used them we can throw them away and create another Factory object for another customization later. So there is no overhead of maintaining an indirection to an already created chain of Factory objects.

In fact, the spaghetti reference problem has become less intrusive then in the client program that is without factory objects. This is because the desiredPrinterfactory object completely encapsulates the wrapper creation and composition logic. The client program can simply change the desired wrapper combination at run-time by replacing the desiredPrinterFactory object with another PrinterFactory wrapper chain. In other words, encapsulating the wrapper composition logic in a separate software component enables separation of concerns and therefore results in a clean design of a client-specific customization mechanism. This is an important lesson on which the thesis of this dissertation is inspired.

However, the Factory object solution is still far from ideal. First of all factory objects must be anticipated in the design. We must introduce a separate factory object for every (wrappee type, wrapper class) pair. Besides adding a considerable amount of code, factory objects may also slowdown execution. Every wrapper construction involves a lookup in the dictionary. Since wrappees are used as key by the lookup in the dictionary, the lookup may become extremely expensive in applications that have large sets of objects. In other words, though factory objects can solve some problems, they do not appear to scale well either.

Furthermore, although the Factory object solution renders the spaghetti reference problem less intrusive, it does not remove the underlying object identity problem which is the real

source of the problem. A wrapper object still has an object identity of its own and therefore hides the identity of its wrappee. As such the programmer needs to program with wrappers in a very disciplined way, because otherwise the danger for duplicate state and wrapper chain management will be created anyway. For example in programming languages that use pointer equality the following program produces a duplicate state problem:

```
Document d = new DocumentImpl();
Printer p = d.getPrinter();
Printer p1 = (new APrinterFactory()).wrap(p);
Printer p2 = (new PrettyPrinterFactory()).wrap(p1);
Printer p3 =(new PrettyPrinterFactory()).wrap(p);
```

Although objects p1 and p logically denote the same object, they appear to be non-equal. This will cause the lookup in the PrettyPrinter dictionary on p and p1 to return different instances of the PrettyPrinter class which results into duplicated state.

As a result wrappers are no longer transparent unless pointer equality is strictly avoided. This restriction is simple to enforce in object-oriented languages like SELF[US87] and Java[AG96], where equality is a user-defined operation rather than a pointer comparison, so that we can define equality of wrappers to be equality of the wrapped objects. In other languages, wrappers require strict coding discipline, thereby introducing the possibility of programming errors. [Höl93]

2.4 No type transparency

Even though the performance problems caused by wrappers can be severe, wrappers could still be practical if their use could be restricted to a few small areas in the program so that overall performance impact would be small. Unfortunately, this is not likely to happen [Höl93]: wrappers have a tendency to spread, "infecting" everything they touch with so called *empty wrappers*. An empty wrapper does not contain any real functionality of its own but is only necessary to get the real wrappers appropriately applied to the program. This wide-spread scattering of empty wrappers throughout the program is a symptom of the lack of type transparency from which wrappers in contemporary programming languages tend to suffer[BW00].

Suppose a class extends the Printer interface with an additional method. For example, the ColorPrinter class below has an additional method for setting the coloring on and off:

```
public class ColorPrinter implements Printer {

    public void print(Document d) {
        ...
    }

    public void setColoring(boolean on) {
        ...
    }
}
```

Now the problem is that applying the PrettyPrinter wrapper to a ColorPrinter wrappee

results in a aggregate that is a subtype of the `Printer` type but not of the `ColorPrinter` type. Indeed, executing the following code will fail:

```
Printer p = new ColorPrinter(); PrettyPrinter p1 = new
PrettyPrinterFactory().wrap(p);
((ColorPrinter)p1).setColoring(false); //this will fail!
```

The problem is that the PrettyPrinter wrapper is only a subtype of the static type of its wrappee , but not the dynamic type of its wrappee. As such, a wrapper hides the actual type of its wrappee.

This *lack of type transparency*[BW00] is the cause why wrappers have the tendency to spread . Existing classes, such as the ColoredDocument class below, that depend on the ColorPrinter type cannot be integrated with the PrettyPrinter wrapper. The ColoredDocument class expects an instance of class ColorPrinter, but PrettyPrinter is not a subtype of ColorPrinter and therefore PrettyPrinter wrappers cannot be used where ColorPrinters are expected.

```
public class ColoredDocument extends DocumentImpl {

  ColoredDocument(ColorPrinter p) {
     printer = p;
  }

  public void print() {
     if (!coloringDesired) {
        ((ColorPrinter)printer).setColoring(false);
        super();
  }
}
```

Comtemporary programming languages provide us with two options to work around this problem, but all are unsatisfactory. The first option - invasively changing the source code of ColoredDocument class so that it can work with a PrettyPrinter object violates the modularity principle and therefore is unacceptable in the context of achieving dynamic and context-sensitive customization. As such we are forced to take the other option: wrapping the ColoredDocument class with a new 'empty' wrapper that adapts it to PrettyPrinter wrapper. For example the PrettyColoredDocument class below illustrates such an empty wrapper:

```
public class PrettyColoredDocument implements Document {
   protected Document document;

   public PrettyColoredDocument(ColoredDocument d) {
     document = d;
   }

   public void print() {
     ColorPrinter p = (ColorPrinter)document.getPrinter();
     if (!document.coloringDesired) {
        p.setColoring(false);
     }
```

```
        ((new PrettyPrinterFactory()).wrap(p)).print(document);
    }
}

public static void main(String[] args) {
    Document d = new ColoredDocument(new ColorPrinter());
    /***********************************************
    Execution of this code will exit with a type error when
    the print method is invoked on variable d:
    PrettyPrinter p1 = (new PrettyPrinterFactory()).wrap(d.getPrinter());
    d.setPrinter(p1);
    d.print();
    ************************************************/
    Document d1 = (new PrettyColoredDocumentFactory()).wrap(d);
    d1.print();
}
```

So, even though we wanted to wrap only Printer objects, we ended up having to create an *empty wrapper for another class*. Even worse, if another class also used ColorPrinter we would have to create a second empty wrapper for that other class as well. As such it is clear that wrappers have to tendency to spread. A second problem is that empty wrappers often have to duplicate existing code from their wrappee classes. Duplication of code is in general not desired because it leads to severe maintenance problems whenever the implementation of the wrappee class evolves. For example, the `print()` method of `PrettyColoredDocument` duplicates the code of the `ColoredDocument` class' `print()` method. Whenever the latter method is updated, the former must be accordingly updated.

What appears to a better solution - writing a wrapper class that extends the PrettyPrinter class with the `setColoring()` method - doesn't work. It is after all not possible to make that wrapper a subtype of `ColorPrinter` because ColorPrinter is a class, not an interface. Another clearly fake solution - offering a generic method `getWrappee()` for retrieving the wrappee of a wrapper - may be useful for other purposes, but doesn't work here either. If we passed the wrappee of the PrettyPrinter wrapper to ColoredDocument instead, the PrettyPrinter extension would not be executed after all.

We can conclude that the lack of type transparency lies at the base of why wrappers have the tendency to spread, introducing empty wrappers everywhere they touch. As such, there exists only one clean approach to resolve the problem, namely that the wrapper simply does not hide the dynamic type of its wrappee but is a subtype thereof.

2.5 The object schizophrenia problem

The term object schizophrenia has been introduced in publications on subject-oriented programming [HO93]. This means that serious problems can arise when the functionality that conceptually belongs in a single application object is split across multiple objects. These problems are due the fact that the separate objects have their own, separate identities; even if they cooperate, they don't truly behave like a single application object unless great care is taken[HO02].

[CHOT99] illustrates this point. Although an behavioral extension, say logging, could be modularly represented and composed using the wrapper-based approach, the problem is that in order to ensure that logging occurs *consistently*, it is necessary that *all* messages to all objects go through the wrapper and not directly to the object itself: once a method on an object is invoked, that method may invoke others (i.e self calls), which in turn must go through the wrapper. This means that the object must know about its decorators, which introduces a new form of coupling and tangling (i.e. each class must include code to implement the interaction with the decorator).

2.6 Conclusion

Though wrappers support context-sensitive customization, they do not appear to scale well at all. The underlying cause of this is that a wrapper hides the identity and type of its wrappee.

Partial remedies to the problems exist but these that are all either unsatisfactory because they violate modularity, or they introduce a severe performance overhead and require strict coding discipline. As a consequence these remedies lead to complicated and unmanageable software systems, even for relatively small toy examples.

3 Class-based inheritance

A study of the problems of inheritance with supporting independent extensibility and independent evolution is presented. We focus on discussing problems. Existing solutions to these problems are only discussed if it contributes to the understanding of the problems. It suffices to say here that there exist multiple solutions to each of these problems but none of them are completely satisfactory.

Multiple inheritance and mixin-based inheritance[BC90] are inheritance techniques that support composition of independently developed components. With multiple inheritance an inheritor can have multiple parents which may be mutually ignorant. With mixin-based inheritance, components are delivered in the form of mixin-classes. In contrary to ordinary inheritance where the incremental modification of an existing parent are embedded directly in the subclass, in the mixin-based approach they stay free in mixin-classes. These are components existing independently of the parent they modify. Mixin-classes are not structurally bound to any specific place in the inheritance hierarchy and therefore can be applied (mixed in) to a set of different parents[Mez98].

3.1 Encapsulation and the semantic fragile base class problem

Inheritance suffers from an inherent encapsulation problems with respect to specialization interfaces. This needs some further explanation.

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external *interfaces*. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers. If clients depend only on the external interface, the module can be reimplemented without affecting any clients, so long as the new implementation supports the

same (or an upward compatible) external interface. Thus, the effects of compatible changes can be confined[Sny86].

In most object-oriented programming languages objects are considered as abstract data objects and the external interface of an object is the set of operations defined upon it. Most object-oriented languages limit external access to an object to invoking the operations defined on the object, and thus support encapsulation. Changes to the data representation of an object or the implementation of its operations can be made without affecting users of the object, so long as the externally-visible behavior of the operations is unchanged[Sny86].

Inheritance complicates the situation by introducing a new category of client for a class. In addition to clients that simply instantiate objects of the class and perform operations on them, there are other clients (class definitions) that inherit from the class[Sny86]. We call the former category of client *message-passing* clients and the latter category *inheriting* clients. Consequently we distinguish between a *client* interface, used by the message-passing clients, and a *specialization* interface, used by inheriting clients. The specialization interface is just as important as the client interface as it serves as a contract between the class and its inheriting clients, and thus limits the degree to which the designer can safely make changes to the class[Sny86].

With current programming language technology is very hard to design specialization interfaces that provide sufficient encapsulation without limiting the power of inheritance too much. Although encapsulation towards inheritors is an important issue, inheritors often need to breach the encapsulation of their ancestors to a certain degree [SM95].

The simplest form of breaching encapsulation is where an inheriting client can directly access the instance variables of its ancestor. The encapsulation problem is here that the implementation of the inheriting client now relies on particular implementation details of the ancestor. As such, the ancestor class cannot freely decide to change its implementation (e.g renaming its instance variables), because any such change potentially breaks the implementation of all inheriting clients and therefore all inheriting clients need to change their implementation as well. This problem has been referred to as the *semantic fragile base class problem* [Szy98]. This problem is difficult to solve because in contrast to message-passing clients, inheriting clients are privileged, in the sense that the functionality of the ancestors they modify is not encapsulated. Not allowing subclasses to see any details of their superclass would totally neglect the power of inheritance[Mez98].

Another variant of the semantic fragile base class problem is associated with the late binding of self that is inherent to inheritance. Take for example the Person class below. In this example the Female subclass breaches the encapsulation of its Person super class by overriding the getName() method, i.e. the Person class cannot change its implementation without adversely affecting the implementation of the Female subclass. If for, example, the Person class decides to implement its print() method in the style of the AlternativePerson class then the Female class will need to adapt its implementation also[SM95]. Here, the Female subclass has no direct access to the private instance variables of its superclass, but it needs to see all the methods sent to self by its superclass, in order to know which superclass methods it can invoke and what will be the effect of overriding methods.

```
public class Person {
```

```
  private String name;

  public String getName() {
    return name;
  }

  public void print() {
    System.out.println(this.getName());
  }

}

public class Female extends Person {
  public String getName() {
    return "Ms. " + super.getName();
  }
}

public class AlternativePerson {
  private String name;

  public String getName() {
    return name;
  }

  public void print() {
    System.out.println(name);
  }
}
```

In [Sny86] the conflict between inheritance and encapsulation is resolved by reverting to a weaker definition of encapsulation for specialization interfaces. Here, the specialization interface of a class is the client interface extended by all methods called via self in the class and its super classes [Kni98]. Further work on "typing" the specialization interface points out that this specification should be augmented with an interaction contract specifying for each method which other methods it invokes via self[Lam93, SLMD96].

Unfortunately, contemporary object-oriented language technology does not present the necessary linguistic means to explicitly specify the specialization interface according to the above definition. Instead, the code of a class must be inspected in order to determine its specialization interface. For practical reasons, many current object-oriented languages have adopted another policy, requiring explicit specification of visibility constraints. E.g in C++ and Java the methods and variables in the client interface of a class are explicitly marked with the keyword *public*, those in the specialization interface with the keyword *protected*, and those visible only to the class itself with the keyword *private*[Kni98]. As such the semantic fragile base class problem associated with the late binding of self cannot be excluded in contemporary languages. To blindfold disallow the late binding of self is again unacceptable because than the power of inheritance is not used.

The above problem occurs in single inheritance hierarchies. It is clear that the problem

becomes much more complex in multiple inheritance hierarchies because different specialization interfaces, stemming from independently developed components, must be combined and possibly reconciled.

3.2 Name collisions

Name collisions possibly arise with the inheritance techniques that support composition of independently developed components .

There are two categories of name collisions treated in the literature: the *homonymous attributes problem* and the *common ancestor dilemma problem*. These have originally been studied in the context of multiple inheritance but, as we argue below, name collisions arise with any inheritance technique supporting composition of independently developed components. There are two kinds of conflicting attributes: *methods* and *state* variables.

Homonymous attributes

A name conflict occurs when components are composed that accidently have attributes with the same name. The term *homonymous attributes*[BL91, MvL96] is used in the literature to describe this conflict. The solution to this conflict lies in the recognition that homonymous attributes have completely different meanings and, therefore, they should be regarded as two independent versions which coexist within the definition of a single object, which yet are visible within limited, mutually disjunct scopes[Mez98].

Partial solutions. To deal with homonymous attributes, some form of *class qualification* (also known as *qualified message passing*) must be used. Class qualification means that every message (either an 'ordinary' message or an invocation of a parent operation) can be qualified with the name of a class. Any class that is in the inheritance chain of a receiver can be used as a qualifier. Due to this qualification, method lookup starts from the specified class rather than directly from the class of the receiver of the message[Ste94].

Class qualification is criticized in [CG90] because it violates late binding of self. It allows a client of a class to select a non-most specific definition of some attribute defined on that class (the so called refinement inhibition problem) and it also encodes too much information about the class hierarchy (the so called genericity inhibition problem). These problems can disable further refinement of a certain attributes. The following example due to [Ste94] will illustrate this point. Consider a class A, with two methods x and y. Although at first glance it does not seem so, there is a fundamental difference between sending a message y, from within the method x, to self and sending a message y, from within the method x, to self qualified with class name A. Both have the same behavior for instances of class A, but for inheriting clients of class A it is in the latter case impossible to override the method y.

The authors of [CG90] propose the *point of view notion of multiple inheritance* to solve these problems. Their solution is essentially based on realizing class qualification orthogonal to inheritance.

Eiffel [MW95] applies renaming which is the semantical equivalent of class qualification. Renaming therefore also violates late binding. Furthermore, renaming must be performed manually by the programmer, which is not desired in dynamic environments where compo-

nents must be composed at run-time. Renaming has also been proposed for mixin-based inheritance [BL91]. Due to the expressiveness of mixin-based inheritance, renaming is realized orthogonal to inheritance, and consequently does not violate the late binding of self. However, the programmer still has the obligation to rename attributes[Mez98].

Since the above approaches require hard-coding static information about the inheritance hierarchy and/or explicit intervention by the programmer, they all fall short when considering dynamic customization. Given the requirements of a dynamic customization we need a mechanism that *automatically* solves the name collisions.

Independent evolution

A glimpse of such a solution can be found when looking in the area of independent *evolution*. Independent evolution means that a common code base, on which other code bases depend (through inheritance), independently evolves without coordinating with the other code bases. Consequently different versions of the common code base exist and the other code bases autonomously decide when they upgrade their instance of common code base to a newer version. This situation typically arises with class libraries.

When independent evolution is an issue, name collisions even occur in single inheritance hierarchies. The following example will illustrate this. Suppose we have a library class Base that is inherited from by class Sub (class Sub is part of application software that uses the library):

```
package library;
 public class Base {
   public void foo() {...}
}

package application;
 public class Sub extends Base {
   public void foo() { }

   public void bar();
}
```

Suppose now that some time later a new version of the Base class from the library package is released and that the new version of Base contains a new method that accidently has the same signature as the bar() method from class Sub, but has an entirely different meaning:

```
package library;
 public class Base {
   public void foo() {...bar(); ...}

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

In this case, overriding of `Base:bang()` by `Sub::bang()` will be enabled which is semantically unsound. We have here a variant of the homonymous attributes problem. `Base::bar()`

and `Sub::bar()` are methods with different meanings that accidently have the same name. Since they have completely different meanings, they should be regarded as two independent versions which coexist within the definition of a single object, which yet are visible within limited, mutually disjunct scopes, i.e. respectively Base and Sub.

Fortunately, there exist an elegant solution to this problem. The programming language C#[Gun01] has introduced explicit modification modes that can be individually set for different methods. When a developer introduces a late-bound method in a class, she needs to explicitly state whether she intends to override a base class method or not. If a method is later added to the base class, it cannot inadvertently be overridden by an existing method in the derived class.

```
package library;
 public class Base {
   public virtual void foo() { bar()}
   public virtual void bar()
}

package application;
 public class Sub extends Base {
   public override void foo() { bar() }

   public virtual void bar();
}
```

In order for this to work the method lookup process must take into account the static type of the receiver. Note that in opposition to message qualification and renaming, this mechanism does not violate late binding of self:

```
Base b = new Sub();

b.bar(); //invokes Base.bar(b)

b.foo(); //invokes Sub.foo(b); Sub.foo(b) invokes Sub.bar(b)

Sub s = (Sub)b;

s.bar(); //invokes Sub.bar(s);
```

This is because the mechanism used for specifying and controlling visibility scopes is kept completely orthogonal to the inheritance operator.

The common ancestor dilemma

[Sny87] points out a particular troublesome situation with multiple inheritance, called the "diamond problem" (also known as "fork-join" inheritance[Sak89]), 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 [Knu88], 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. Different solutions to support this in multiple inheritance hierarchies have been proposed. None of the approaches is fully satisfactory however: (due to [Ste94, Mez98]):

- graph multiple inheritance: suffers from encapsulation problems and from the undesired duplicate parent operation [Sny87]
- linear multiple inheritance: all replication is simply not allowed to occur.
- tree multiple inheritance: only supports replication [Knu88]

A more general technique is to use renaming[MW95]. 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[Sak89].

Before proceeding the discussion, we first make two important observations. First, there are two kinds of conflicting attributes: methods and state variables. We focus in this section on conflicting state which is more problematic. Method conflicts can easily be resolved by means of renaming.

Second, the existing literature we have studied [Knu88, Sak89, Ste94, Mez98, SDNB03] indicates that the diamond problem is very difficult to solve for conflicting state. As such, it is tempting to ignore multiple inheritance: given the fact that the diamond 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[DMSV89, SG99] and conceptual problems[SDNB03] anyway. This would however be the wrong thing to do as argued by the following two points. First, as indicated by [MW95], 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. Consequently we cannot ignore this particular form of composition. 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[Ste94]). Second and more importantly, Mira Mezini referred to the diamond problem in her dissertation[Mez98] as the 'common ancestor dilemma' problem, thus indicating that the problem 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, or any other eligible inheritance technique), the problem appears. For example, if inheritance between mixin-classes was allowed, mixin-based inheritance would also have to deal with the problem. Figure 2 illustrates this. Consequently the diamond problem is an instance of the more general 'common ancestor dilemma' problem.

Having indicated the relevancy of the problem, let us now explain why the common ancestor dilemma causes so many hard problems. As demonstrated in [Knu88] it is desirable

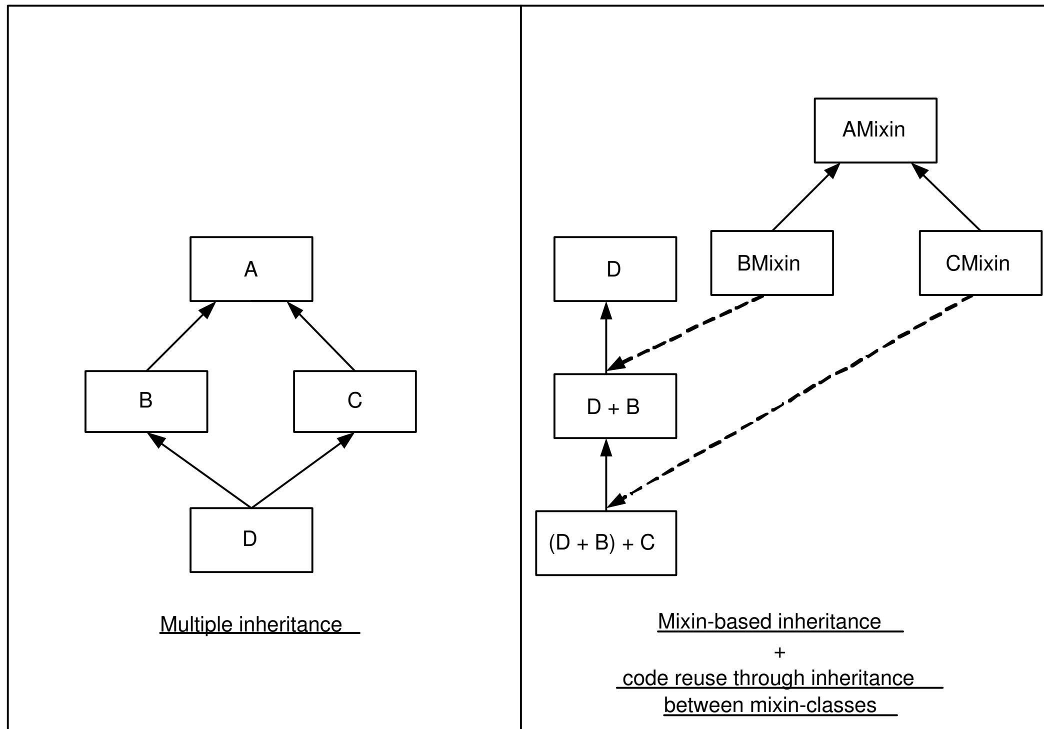

Figure 2: diamond problem ⇒ common ancestor dilemma

to be able to choose the alternative (replication versus sharing) individually for each attribute. Figure 3 (due to [Knu88, Ste94]) 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.

As a result, the common ancestor UniversityEmployee gets effectively split into two. The hard problem with this, according to Markku Sakkinen[Sak89], 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 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[Sak89].

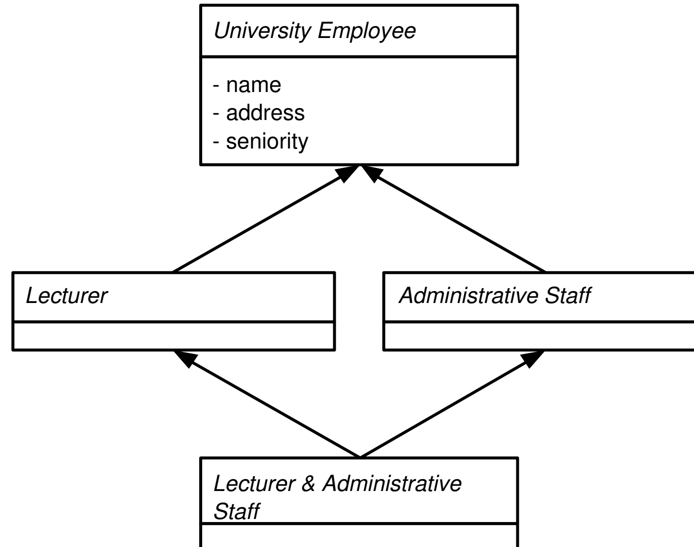

Figure 3: Common ancestor dilemma

If we translate the above example 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 4 illustrates this. The integrity of subobjects is thus violated according to Sakkinnen: "*the really fatal defect is that any operation of [UniversityEmployee], [Lecturer] or [AdministrativeStaff], public, protected or private, that both updates attributes [from the S part] and accesses attributes [from its local R part], may cause unwanted side effects between the [AdministrateStaff] and [Lecturer] subobjects. All operations must therefore be checked, and the code-sharing advantage of inheritance is lost[Sak89]*".

In order to circumvent integrity violations, 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-University-Employee. Lecturer and AdministrativeStaff would inherit without sharing from Only-University-Employee [Sak89]. Figure 5 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 during class design.

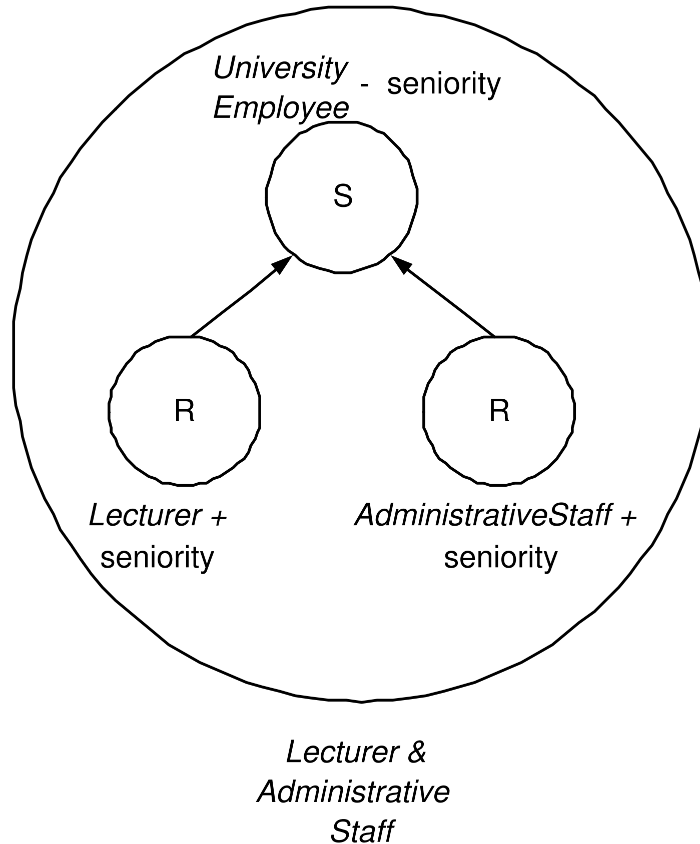

Figure 4: Splitting the common ancestor

4 Object-based inheritance $\neq$ Objects + Inheritance

In this section we discuss static delegation (also known as object-based inheritance). Static delegation takes inheritance to the level of objects.

We first give an introduction to delegation since it is not as well known as inheritance and wrappers. Section 5 compares then the three compositional models (wrapper-based approach, static delegation and inheritance). Last but not least, we discuss a major problem with static delegation that must be dealt with in order to make the use of delegation safe. In order to cope with this problem the use of delegation must be restricted. These restrictions basically boil down to the fact that static delegation must be combined with the object model of class-based languages, leading to a *hybrid approach*.

4.1 Classless delegation

Major part of this section is taken from [Ste94].

Delegation was originally introduced by Lieberman[Lie86] in the framework of a class-less prototype-based language. In prototype-based languages objects are not organized into

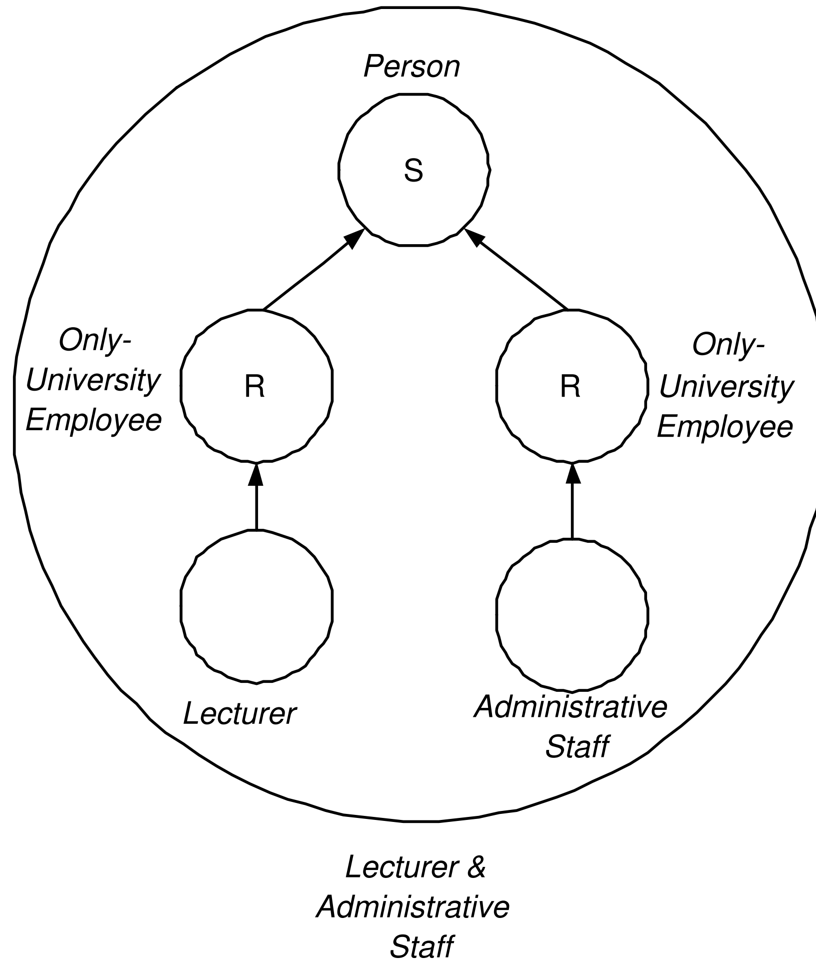

Figure 5: Splitting the common ancestor during class design

classes. Whereas in class-based languages objects are created by first defining a template (i.e. a class) and then instantiating this template, in prototype-based languages objects are created directly: an object is created by listing its public and private attributes (both methods and state variables) and the initial values for instance variables[Ste94]. For example consider the object definition below (due to [Mez98]):

```
JeffAccount = Object {
    private owner = Jeff; idNumber = 324519; amount = 2283;
    public credit(sum) {amount = amount + sum;}
    public debit(sum) {amount = amount - sum;}
    public check {return amount;}
}
```

Furthermore, whereas in class-based languages objects of the same kind are created by taking different instances of one and the same template, in prototype-based languages objects

of the same kind are created by cloning an existing object with the extra advantages that state is copied into the newly created object[Ste94].

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*[Kni99].

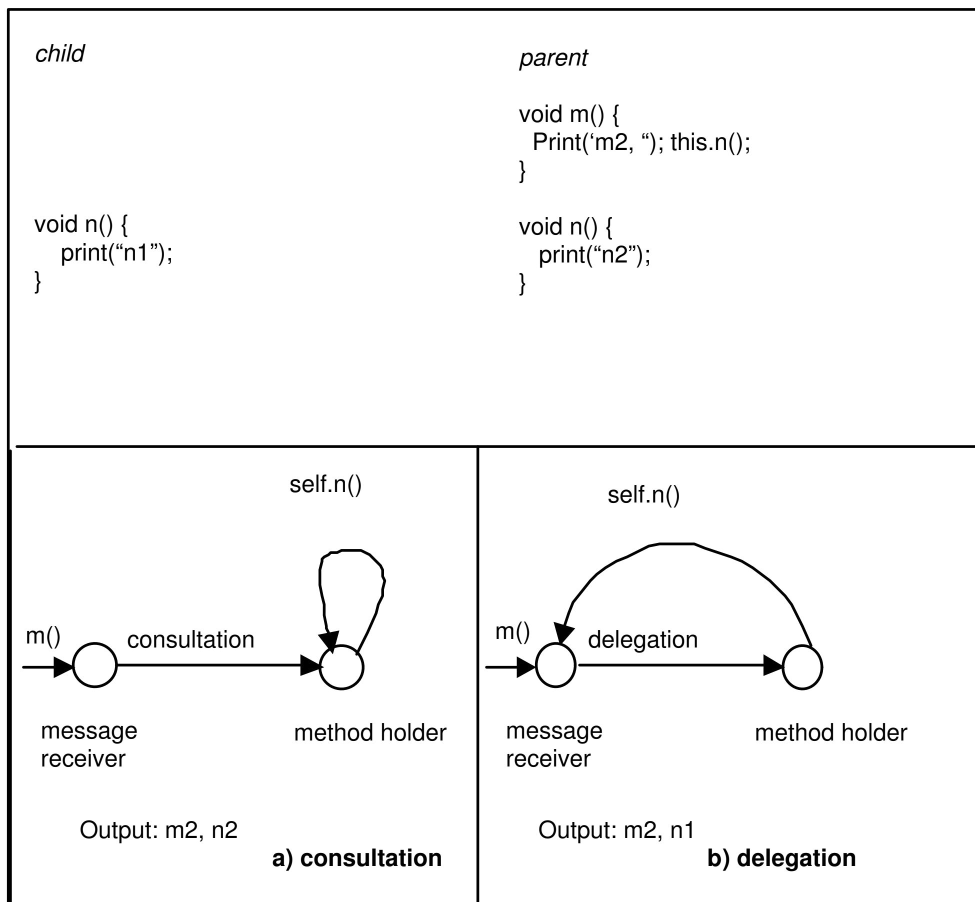

Figure 6: Different effect of delegation and consultation on self

Two variants of delegation

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, this 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. Static delegation can also be interpreted as an incremental modification mechanisms and, therefore, static delegation has also been called *object-based inheritance*[Ste94]. A child and parent respectively correspond inheriting client and ancestor.

Context-sensitive composition

In class-based languages the component instances of an object are completely fixed after the object is created, whereas in static delegation the component instances of an application object can be incrementally added (and again removed) during run-time. This is because the model of delegation requires that any client can always pass a new component instance as the self parameter of an application object and as a result can accommodate for both run-time composition and context-sensitive composition.

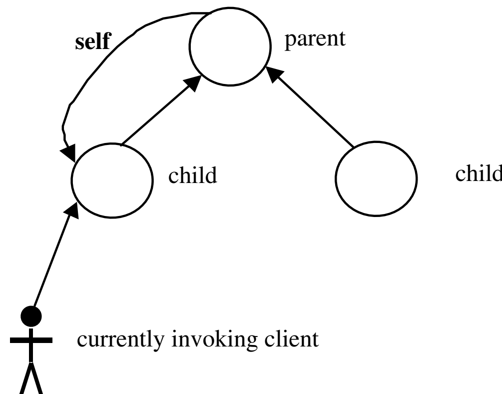

Figure 7: Delegation supports run-time and context-sensitive composition

4.2 The immaculate client interface principle

Delegations introduces a major problem with respect to encapsulation. As indicated in section 3.1 inheritance suffers from the semantic fragile base class problem which is associated to an inherent encapsulation violation with respect to the specialization interface. This violation of encapsulation is due to the late binding of self[SM95].

The semantic fragile base class problem present with inheritance is not a severe problem in class-based languages due to the clean separation between the client and specialization interface and the fact that the inheritance relation is rather static. With delegation however, anyone who can access the client interface of an object can also see its specialization interface.

The very model of object-based inheritance requires after all rebinding of self each time a message is sent. As a result, any client can pass with every message a new extension as the self parameter of an object.

In other words, delegation allows any object to be extended by any other object. But then the semantic fragile base class problem inherent to inheriting clients concerns the instantiating clients as well[SM95]. All clients can thus see the implementation details exported by the specialization interface. Clemens Szyperski [Szy98] referred to this problem as "delegation is making object composition as problematic as implementation inheritance".

The semantic fragile base class problem, originally formulated in the context of class-based inheritance, reappears in object-based inheritance as a lack of support for independent extensibility. Any composition of components may give rise to object behavior with unexpected or erroneous semantics at run-time. Since the delegation relation is dynamic such errors cannot be statically detected. As such, run-time composition cannot be exploited to its fullest benefit.

The only way to deal with this is to restrict the use of delegation (to be more precisely: the late binding of self at the level of objects). As such, it is clear that the use of static delegation must be strictly disciplined. [SM95] has formulated the *immaculate client interface principle* that constrains the design space of programming languages supporting delegation. It states that "*an object should not expose any other interface but the client interface to its message passing clients. It must be able to hide its specialization interface from message passing clients*". The specialization interface of objects must thus be kept separated from their client interface because otherwise all encapsulation problems inherent to specialization concern the object's client interface as well.

5 The three compositional models compared

Delegation is thus a special form of wrapping. What differentiates delegation from wrapping is the interpretation of the self reference. As illustrated in figure 8, the difference between wrapping and delegation is similar to the difference between aggregation and inheritance [Ste94]. Delegation supports a common self across webs of objects, whereas with normal wrapping each object encapsulates its own fixed version of self. As such with delegation self calls are invoked on the web of objects as a whole, whereas with ordinary wrapping each self call is solely invoked on the invoking object.

One could term webs of objects that share a common self as objects of a higher order themselves[Szy98]. The latter concept is conceptually similar to the notion of complex object, as proposed by [Sak89] (see more about this in section 7.3).

The difference between inheritance and delegation is that with inheritance the self parameter of an object is fixed after the object has been created, whereas with delegation the self parameter can vary depending on which object is the current message receiver.

The delegation model in Figure 8 presents a picture of the point in time when a message is sent to object D. As a result, the self parameters of the parents to which D directly (C) or indirectly (A) delegates are all dynamically bound to D. The self parameter of object B is not bound at that time. When a subsequent message is sent to object B, the self parameters

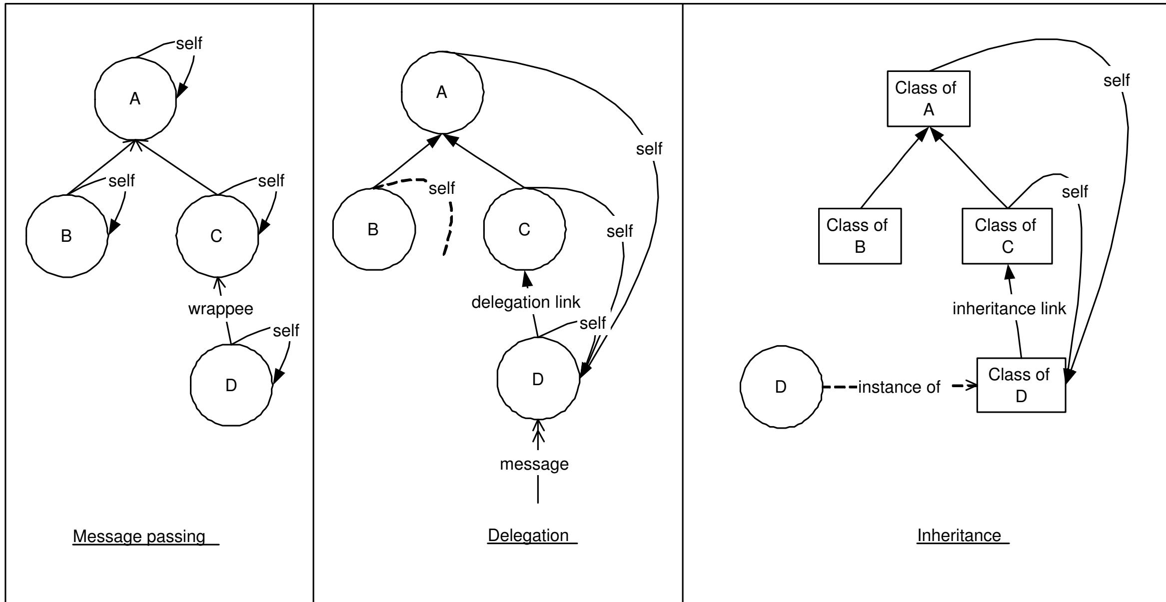

Figure 8: Comparison of compositional models

of objects A and B will be dynamically bound to B, whereas the self parameters of object C and D would be the ones that are left unbound.

In the example code shown below (due to [Mez98]), this is illustrated by the self invocation within debit in JeffCheckingAccount.

```
JeffCheckingAccount = JeffAccount extended with {
    private passwd = xxxyyy; overdrawnLimit = 1000;
    public debit(sum) {
      if (sum <= self.debitLimit()) then super.debit(sum);}

    protected debitLimit() {return overdrawnLimit;}
}

JeffATMAccount = JeffCheckingAccount extended with {
    protected debitLimit {...}
    protected specialLimit {...}
}
```

The self reference may refer here to two different objects, depending on whether the debit message has been received by JeffCheckingAccount or JeffATMAccount - it may denote the JeffATMAccount object or its inherited "subpart" JeffCheckingAccount. If the debit message has been received by JeffAccount instead, the self parameter of JeffCheckingAccount is left unbound. As a consequence, the self parameter within debit of JeffCheckingAccount cannot be fixed to a certain method environment when the JeffCheckingAccount object is created.[Mez98]. Instead, the self parameter must be rebound each time a message is sent[SM95].

In summary, in a model of classless delegation all objects must have an unbound self. At any point in time a message can be delegated to an object, forcing the self-reference of that object to be redirected to the delegating object[Ste94].

6 Hybrid approaches

Programming languages, that enforce the immaculate client interface principle, are able to enforce a disciplined use of static delegation. This means that only a limited set of certificated child components is permitted to delegate to a given parent component, whereas all other components can only send messages to it. Enforcing this restriction basically boils down, as outlined in [SM95], to the design of a *hybrid* programming language that is a marriage between the object model of class-based languages and the compositional model of static delegation. There already exist several such hybrid approaches which vary in the trade-off they make between flexibility and safety. The existing hybrid approaches can be classified according to their suitability for supporting *anticipated extensibility* or *unanticipated extensibility* depending on whether the certificated child components of a given parent component must be statically known in advance or not. At their respective extremes, anticipated extensibility completely sacrifices flexibility for the sake of safety, whereas unanticipated extensibility completely sacrifices safety for the sake of flexibility. Obviously, both extremes are undesired. It is the art to make a sensible trade-off here.

The Split object model [BD96] and Rondo object model[Mez97] provide a special operation for adding and deleting child components dynamically. Here the immaculate client interface principle will be respected if it can be made sure that the operation for adding or deleting child components is not accessible to message-passing clients.

The Lava language [Kni99] and *generic wrappers* approach [BW00] propose provide object-based inheritance in a statically typed class-based environment. Here, objects may only delegate to other objects if the classes of the former have declared the references to the latter as special *delegation attributes*[Kni99] or *wraps relationships* [BW00]. Assuming that classes are not created dynamically, these approaches are safer than pure object-based inheritance, because the possible delegation relations are statically constrained by respectively the declared types of the delegation attributes and the wraps relationships.

7 Revisiting the problem analysis

The above hybrid approaches make something like a combination of inheritance, delegation and the wrapper-based approach. Like some inheritance techniques and delegation do, the hybrid approach supports composition of independently developed components. Like the wrapper-based approach and delegation do, the hybrid approach supports dynamic and context-sensitive composition of components. The problems of the wrapper-based approach, inheritance and delegation are thus combined as well. Some problems are solved, some problems remain the same, some problems are aggravated. As such the problem analysis that was done for all three compositional models must be repeated. In order to keep the discussion

light, an example of a concrete hybrid programming model is first given in the context of which the problems will be revisited.

7.1 A concrete hybrid model

In the remainder of this section we discuss a hybrid programming model that is a slightly adjusted version of the generic wrappers approach*generic wrappers*[BW00]. It allows the three compositional models of figure 5 (class-based inheritance, object-based inheritance and the wrapper-based approach) to be used together.

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() {
      self.bang();
      super.b();
   }

   public void bang() {}
}
```

Child objects are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. 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) must instances of Child 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 [BW00] by `< >`) to class instance creation expressions

```
Parent p = new Parent(...);
DeclaredParent c = new Child(...)<p>;
if (c instanceof Parent) {((Parent)c).bang()}
```

The compiler checks that the declared type of variable `p` is a subtype of the static wrappee type. The wrapper class instance creation expression throws an exception if the value of `p` is null of if `p` were an expression and its evaluation throws an exception.

The particularity of the proposal of [BW00] is that instances of wrapper classes are not only of the static, but also of the actual wrappee type. For example, a `Child` 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.

Methods declared in the wrapper override those in the wrappee analogously to overriding in subclasses.

In constructors and instance methods of child classes, the keyword `wrappee` references the wrappee. It can be treated like an implicitly declared and initialized final instance field. Hence, wrappers can call overridden methods of the parent using the keyword `wrappee` corresponding to `super` for overridden methods of superclasses. For example the b() method of `Child` might look as follows:

```
public class Child wraps DeclaredParent {
   public void b() {
      ...
      wrappee.b();
      ...
   }
}
```

7.2 The immaculate client interface principle

The problem that pure object-based inheritance models suffer from an inherent encapsulation problem with respect to the specialization interface and the discussion how the different hybrid approaches deal with this problem have been respectively presented in section 4.2 and section 6.

7.3 Name collisions

As stated in section 3.2, potential name collisions occur when two independently developed components are composed. Obviously, pure object-based inheritance supports composition of independently developed components since any object can delegate to any other object. In the hybrid approach composition of independently developed components arises in wrapper aggregates where the actual wrappee type is a subtype of the static wrappee type. The subtype relation may be either an inheritance link (see figure 9) or a delegation link (see figure 10).

Homonymous attributes

First, there are two kinds of attributes: methods and state variables. The homonymous attributes problem is naturally resolved for state variables in the hybrid approach if we assume that all state variables are declared as non public attributes. We think this assumption is implicitly advocated as good style for object-oriented programming since it conforms with the notion of encapsulated inheritance (see section 3.1). In the hybrid approaches we get the notion of encapsulated inheritance for free. Since the hybrid approaches reduce inheritance to

the combination of objects and delegation, the information hiding principle, that is inherently associated with objects, can be used with this end in view.

As a result, a natural mechanism is provided for keeping homonymous state variables separated from each other when all state variables are internally encapsulated in the declaring class. Of course it may be important that some state variables are accessible from the outside: if this is the case we assume that the programmer provides an explicit *getter* and *setter* method for reading and writing the corresponding state variable.

For method conflicts the situation is quite different. Consider the following scenario (due to [Kni99]) illustrated in Figure 9 and Figure 10: c, an instance of class Child, delegates to an instance of class Parent; Parent is a subtype of the declared parent class of Child.

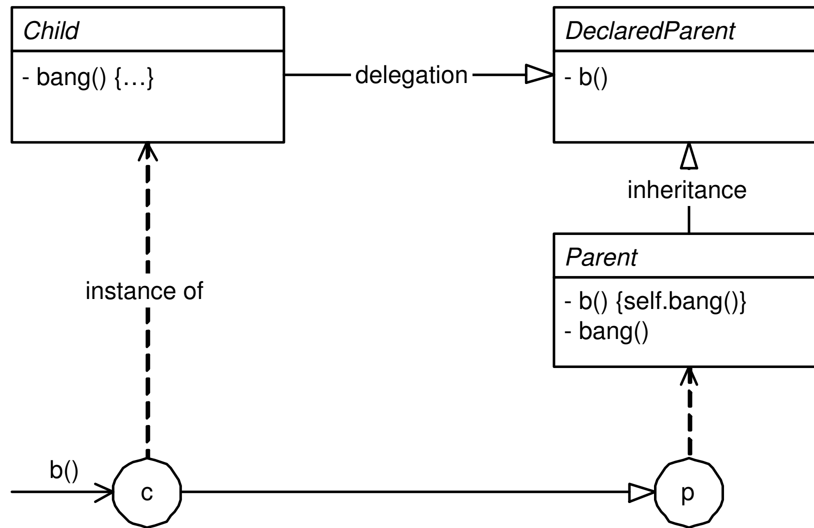

Figure 9: What happens during evaluation of the message c.b()?

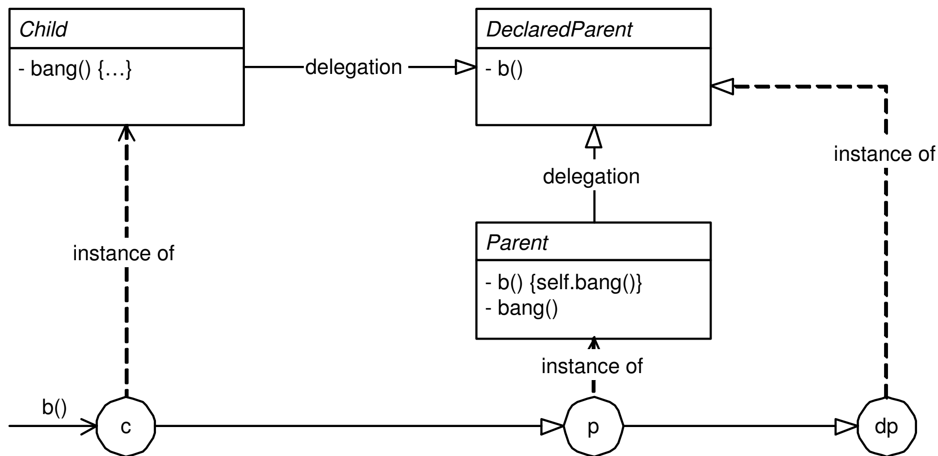

Figure 10: What happens during evaluation of the message c.b()?

In figure 9 and figure 10 the classes `Child` and `Parent` have been developed and compiled independently, knowing only `DeclaredParent` but not each other. The name collision is here that the classes `Child` and `Parent` both introduce a method named `bang()` with different meanings. Overriding of `Parent::bang()` by `Child::bang()` would be semantically unsound. Instead the two methods should be regarded as two independent versions which coexist within the definition of a single object, which yet are visible within limited, mutually disjunct scopes[Mez98].

Pure object-based inheritance cannot enforce such mutually invisible scopes due to late binding of self. Both figure 9 and figure 10 illustrate this: during the evaluation of message `c.b()`, delegated from c to p, the message `self.bang()` in p will be redirected to c, because c is the message receiver of the message `c.b()`.

Günter Kniesel proposes as a solution to this problem the following adapted rule for method overriding :

For a message $recv.n(args)$ (i.e. self.bang()) a method with signature σ from type T (i.e. Child::bang) overrides the matching method from the static type of $recv$, T_{stat} (i.e. Parent::bang) if there is some common declared supertype of T and T_{stat} that contains σ[Kni99].

Thus a method from type T will only override methods from a declared parent type of T. This rule reflects the fact that the common declared supertype (`DeclaredParent` in figure 9 and figure 11) is the common semantic base on which implementors of independently developed components can rely[Kni99].

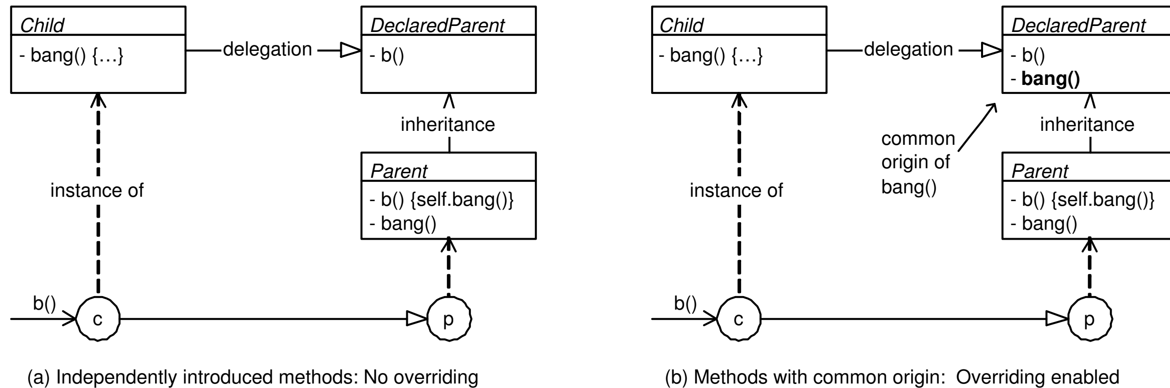

Figure 11: Rule for method overriding

In figure 11a the message `self.bang()` sent from `p` to `c` will not find an applicable method in `c` and be delegated further up the object hierarchy, back to `p` (where the search will succeed). In figure 11b the message will find an applicable method in `c` due to the declaration of method `bang()` in `DeclaredParent`[Kni99].

Opposed to the approaches that deal with the homonymous attributes problem in class-based inheritance hierarchies (qualified message passing and renaming - see section 3.2), the adapted rule for method overriding enables the method lookup and dispatch process to automatically deal with the problem in object-based inheritance hierarchies. In other words, it does not require hard-coding static information about the inheritance hierarchy and/or

explicit intervention by the programmer. Furthermore, as a consequence, the adapted rule does not cause a violation of the late binding of self whereas class qualification and renaming techniques do.

There exist an alternative solution that completely side-steps the homonymous attributes problem. Since the adapted rule for method overriding deviates from normal run-time semantics, an alternative method lookup and dispatch process must be supported by the language execution environment. If such support cannot be easily accommodated in an existing language environment, [BW00] shows that the homonymous attributes problem can be side-stepped in any existing language environment if the programmer adheres to following three coding conventions:

(1) Classes only define (non private) methods declared in implemented or wrapped interfaces

(2) Method calls are only made on variables of interface, but not class types.

(3) No two interfaces, not related by extension, declare methods with the same signature

Thus two components that share the same interface (either by `implements` or `wraps` relationship) are allowed to modify each other.

As [BW00] suggest, we agree that these coding conventions are implicitly advocated as good style for object-oriented programming. Coding conventions (1) and (2) could easily be enforced by a programming language. Coding convention (3) however requires that a component programmer knows in advance all the interfaces that will be used in all applications at all times. This is of course a stringent assumption that strongly constrains independent extensibility. [BW00] notes that instead of (3) a language can require qualified notation for member access instead of merging namespaces of interfaces. But as shown in section 3.2 qualified message passing violates late binding of self. Furthermore as will be shown below, the proposal of [BW00] seriously constrains independent evolution across the delegation link.

Independent evolution

Section 3.2 shows that a variant of the homonymous attributes problem appears in the presence of independent evolution of base classes. Since this problem appears in class hierarchies with single-inheritance, we have to conclude that in the hybrid approach the problem will appear in object hierarchies as well. If in figure 12 the `DeclarentParent` wrappee class was independently modified over time to include an additional `bang()` method, overriding of `DeclaredParent::bang()` by `Child::bang()` would be enabled across the delegation link which is semantically unsound as discussed in section 3.2.

Independent evolution even breaks the solution of the adapted rule for method overriding. If in the example of figure 9 the `DeclarentParent` type was similarly modified to include an additional `bang()` method, overriding between `Parent::bang()` by `Child::bang()` would be suddenly enabled according to the adapted rule for method overriding.

These problems can be sidestepped if the following two code conventions are valid:

(4) child components are only allowed to declare the types of their wrappees as interface types and not as class types

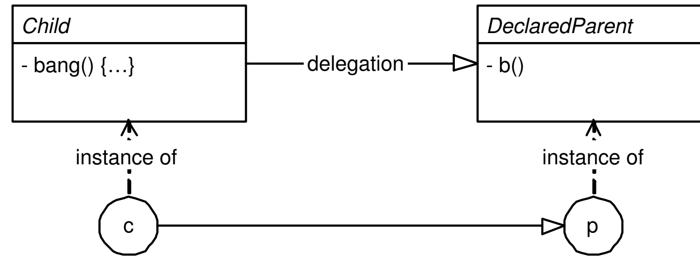

Figure 12: Independent evolution across the delegation link

(5) the set of operations declared by an interface is fixed (i.e. it is a contract that cannot be modified over time)

This is illustrated by the following code example. Instances of class `Child()` are declared to wrap instances of interface type `DeclaredParent`. In this setting any wrappee class implementing the DeclaredParent interface may independently introduce a method `bang()` without being overridden by `Child::bang()`

```
//before independent evolution:
public interface DeclaredParent {
   public void b();
}

public class Parent implements DeclaredParent {
  public void b() {...}
}

public class Child wraps DeclaredParent {
   public void b() {
       ...
       wrappee.b();
       ...
   }

   public void bang();
}
```

Suppose for example that the class `Parent` is modified to include an additional method `bang()` as follows:

```
//after independent evolution:
public class Parent implements DeclaredParent {
  public void b() {... self.bang(); ...}

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

According to the adapted rule, overriding of `Parent::bang()` by `Child::bang()` is disabled for every message, as it ought to be in order to be semantically correct.

For self calls, the static type of `self` in `self.bang()` is `Parent` and there does not exist a common declared supertype of `Parent` and `Child` that contains the `bang()` operation.

For non self calls, the static type of the receiver determines which one of the `bang()` methods will be selected for execution:

```
//demo code for non-self calls:
SimpleParent p = new SimpleParent(..);
Child c = newChild(...)<p>;
c.bang(); //executes Child::bang()
((DeclaredParent)c).bang();//static type error. Operation bang()
                           //is not declared in DeclaredParent
((Parent)c).bang() //executes Parent::bang();
```

Coding convention (5) is implicitly advocated as good style of object-oriented programming because lately extending an existing interface with an additional operation breaks all existing classes implementing that interface, which is clearly undesired.

Code convention (4) stating that only interface types may be used to declare wrappees does not prohibit code reuse across the delegation link[1]. This is because composition of components (and therefore component reuse) is a run-time operation at the object level that is kept completely orthogonal to the declaration of delegation relationships.

Independent evolution versus unanticipated extensibility

Code convention (4) does limit unanticipated extensibility however: it implies that public methods that are not declared in an interface cannot be overridden across the delegation link. This limitation disappears of course if coding convention (1) from [BW00], stating that classes can only define public methods declared in implemented interfaces, is adhered to.

With respect to the solution that is solely based on coding conventions (1), (2) and (3) [BW00], independent evolution and unanticipated extensibility completely collide with each other however: choosing the one implies giving up the other. This is because of coding convention (1) and (3) that basically leave room for selecting between two approaches:

pro independent evolution: a parent class is independently modified with a *non public* method.

pro unanticipated extensibility a parent is modified with a *public* method.

In the first case independent evolution is enabled while unanticipated extensibility is disabled. The lately introduced method is only visible within the scope of the declaring parent class. This is what is desired but at the same time implies that such lately introduced methods can not be further specialized across the delegation link.

[1]if we enforced the same assumption in the context of class-based single inheritance, however, implementation inheritance and thus code reuse would not be possible anymore

```
//before independent evolution:
public interface DeclaredParent {
   public void b();
}


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

public class Child wraps DeclaredParent implements Test {
   public void b() {
      ...
      wrappee.b();
      ...
   }

   public void bang();
}

public interface Test {
  public void bang();
}


//after independent evolution:
public class Parent implements DeclaredParent {
  public void b() {
    self.bang();
  }

  protected void bang()

}
```

The other option - adding a public method to the parent class - disables independent evolution in favor of unanticipated extensibility due to coding conventions (1) and (3).

```
//No independent evolution possible:
public class Parent implements DeclaredParent implements Test {

  public void b() {
    self.bang();
  }

  public void bang() //bang() will be overridden by Child::bang()
}
```

Unanticipated extensibility is enabled because the lately introduced method can be successively incrementally modified by another component. However, all possibilities for independent evolution is ruled out here due to coding convention (3). For example, a public method `bang()` can only be introduced in `Parent` if `Parent` was modified to implement the `Test` interface because `Test` already declares the `bang()` operation and according to coding convention (3) no two interfaces not related by extension are allowed to declare the same operation; As a such `Parent::bang()` and `Child::bang()` would be mutually visible and overriding would be enabled. Although this is not semantically unsound[2], the point we want to make here however is that independent evolution is ruled out here because the `Parent` class can not be independently evolved without coordinating with the rest of the application. When a developer wants to evolve a component with an additional method having signature X, he must investigate whether the applications in which that component has been and will be deployed already use an interface declaring an operation with signature X. If yes, the developer must choose another method signature (i.e. another method name).

As such it seems that in the solution solely based on coding conventions a trade-off must be made between unanticipated extensibility and independent evolution. Either a lately added method is declared non-public (internally encapsulated) enabling independent evolution but disabling any further refinement of the method across the delegation link. Or the method is declared public (placing it in a global visibility scope) enabling unanticipated extensibility but disabling independent evolution.

What is desired instead is that both alternatives can co-exist for the same method. In other words we need a mechanism to make a method controllably visible for one subset of components enabling unanticipated extensibility and, at the same time, controllably hide it for another disjunct subset of components enabling independent evolution.

The underlying reason why this is not possible with the solutions solely based on code conventions is that the same mechanism (public versus non-public declaration of methods) is used for specifying visibility scopes and for governing method overriding, i.e. visibility control is strongly coupled to incremental modification. This lack of clean separation of concerns at the language design space impedes development of components because the component vendor, developing and evolving a component, is continuously faced with an unnecessary balancing exercise between maximizing component reusability and maximizing component extensibility

The adapted rule for method overriding on the other hand augmented with coding conventions (4), (5), and (1) presents a mechanism that allows unanticipated extensibility and independent evolution to co-exist for the same method, as discussed above. This is because the adapted rule for method overriding enriches the design space of the language with a new abstraction for orthogonalizing the issues of visibility control and method overriding.

[2]The `Parent::bang()` and `Child::bang()` methods are semantically compatible because at the time of implementing these methods the respective programmers are aware of the `Test` interface and, therefore, it is expected that both `bang()` methods are semantic refinements of their correspondence declared in the `Test` interface

The common ancestor dilemma

The hybrid approach also has to deal with the common ancestor dilemma. When two or more component instances are composed by means of static delegation, the common ancestor dilemma arises in one of the following two cases as illustrated in figure 13:

(a) inheritance from a common declared superclass

(b) static delegation to a common declared super type

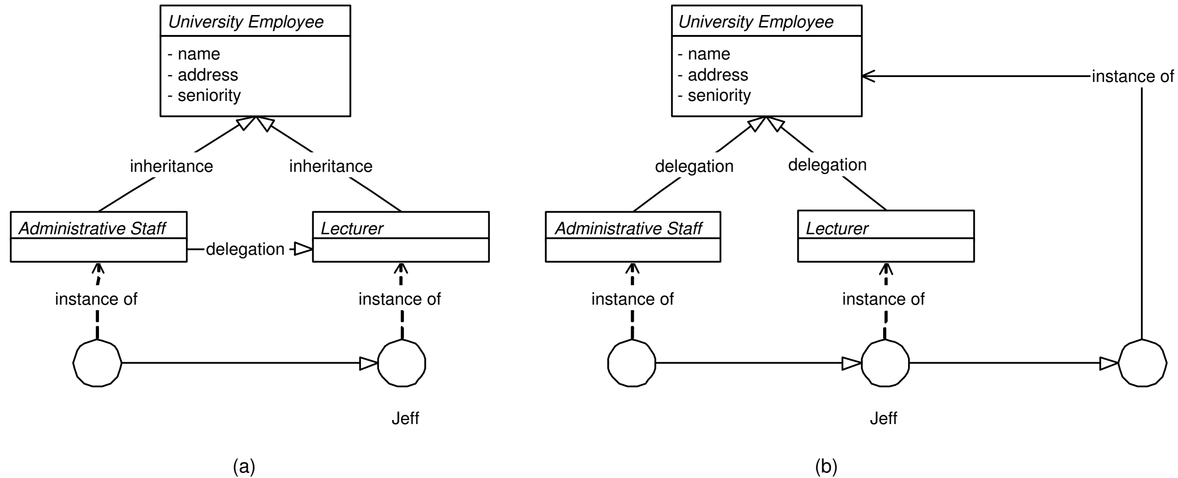

Figure 13: Delegation and the common ancestor dilemma

Since we studied the common ancestor dilemma in the context of Sakkinen's inheritance model[Sak89] (see section 3), we first have to map Sakkinen's model to delegation. This is quite easy to do because Sakkinen reduces inheritance to aggregation: for example in figure 5 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.

Now suppose in figure 13 that 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 3 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.

The problem of integrity violation due to the splitting of subobjects as described by Sakkinen is irrelevant in the context of the delegation model, because there is another problem that precedes the integrity violation: splitting of the common ancestor can a priori *not* be performed because the composition operator provided by delegation operates at run-time at the level of subobjects. Once a subobject has been created it cannot be split anymore.

So, the only option left, as suggested by [Sak89], is to side-step the splitting problem by explicitly dividing the common ancestor into two classes S and R of which the former is to be shared and the latter is to be replicated (see figure 5). The code below illustrates how this scenario can easily implemented in the hybrid approach.

Here comes the power of the hybrid approach into play. Cases (a) and (b) are dual. They present 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.

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

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

}

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

   public Seniority getSeniority() {...}
   public void payOut() {...}
        ...
        self.getSeniority();
        ....
  }
}


public class Lecturer extends Only-UniversityEmployee {
   String title;

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

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

```
public class AdministrativeStaff extends Only-UniversityEmployee
{...}


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

...

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

Replicated methods

The solutions that were discussed in the context of the homonymous attributes problem return to the picture in the context of replicated attributes. Like homonymous attributes, replicated attributes must be kept separated from each other because they are considered to belong to two different subobjects of a single complex object. As such coping with both of the problems reduces to the same basic problem: how to provide and control multiple definitions of an attribute, which coexist within the definition of a single object and yet are visible within limited, mutually disjunct scopes[Mez98].

As already discussed in section 7.3, replicated state variables can easily be kept separated from each other if they are declared as non-public attributes.

Unfortunately the existing solutions to the homonymous attributes problem in the context of the hybrid approaches cannot be used to separate replicated methods from each other. This is because there is an essential difference between homonymous attributes and replicated attributes in the way they are defined. Replicated attributes stem from a common ancestor class, i.e. they are declared by the *same common supertype*. In contrast, two homonymous attributes stem from different ancestors which accidently use the same name for two different meanings. Since the solutions described in section 7.3 (the adapted rule for method overriding[Kni99] and the coding conventions [BW00]) essentially boil down to the existence of a common declared supertype, it is clear that these solutions incorrectly enable overriding between replicated methods i.e. types cannot be used to keep replicated methods separated in mutually invisible scopes.

Let us apply the adapted rule for method overriding to the above code example. In 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, 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 `getSeniority()` method of Administrator.

Note that the existing solutions also break in the case of direct repeated inheritance (see section 3 which is a simpler variant of the common ancestor dilemma.

To conclude, the discussed solutions for the homonymous attributes problem cannot be used for keeping replicated methods separate from each other. This is because the existing solutions use types to maintain several disjunct scope-specific definitions of the same message. Types can not be used for separating replicated attributes, however. As such another mechanism must be used instead.

The Rondo object model[Mez97, Mez98] provides such a mechanism. It partially 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.

7.4 The Rondo object model

The Rondo object model[Mez97, Mez98] presents a sophisticated approach to dynamic and context-dependent object evolution without name collisions. The observation that visibility control and incremental modification must be realized orthogonal to each other is due to this work. The design of Rondo starts from the fundamental observation that inheritance fails to properly solve the semantical mismatch between incremental modification and visibility control because inheritance is overloaded.

The basic philosophy behind Rondo is to support a consequent separation between *basic behavior*, encoded as normal classes, and its context-dependent behavior variations, called *adjustments*. This separation is explicitly supported at the syntactic level by providing dedicated constructs for separately describing basic classes and context-dependent adjustments of objects. Rondo provides the programmer with explicit syntactic constructs to specify the incremental modification relationships between classes and adjustments, i.e. what adjustments modifies what class or adjustment under what context-dependent conditions.

The separation between classes and adjustments is maintained at the semantic level by providing appropriate mechanisms for structuring and composing the separated behavior definitions in a way that guarantees their loose coupling. These mechanisms are organized in two layers. In the *definition layer*, basic classes and adjustments play the role of behavior repositories. Maintaining their incremental modification relationships is the responsibility of so called *managers*. A manager is a kind of dictionary that maintains for each behavior definition module the set of adjustments that modify the module along with the context-dependent condition under which the modification should take place. This is in contrast to conventional object-oriented language execution environments in which class objects directly carry information about their sub- and super-classes. Context-dependent evolution is provided at the definition layer by a special `raise` and `undo` directives (see below for more details about this).

The *composition layer* is populated by *combiners*, responsible for connecting definitions from different modules into fully-formed object behaviors when required. It is within the structures encapsulated by combiners that definition modules are virtually ordered into a behavioral composition. It is this order that is used by the dispatching functionality of the language. Altering the behavior definition of an object is now a matter of updating these structures. It can be performed after the object has been created. Support for dynamic composition is thus provided by the combiners. Context-dependent variation is supported,

as suggested in the previous paragraph, in cooperation with the managers. When an event occurs that should cause a behavior modification, managers are informed by the `raise` or `undo` directive. In response, the managers select the corresponding adjustments and trigger the update of the internal structures of the corresponding combiner, such that the definitions of the chosen adjustments are made part of or removed from the behavior of the underlying object.

It must be noted that adjustment and classes are represented in the composition layer as objects that are kept separated from each other. These objects correspond to the notion of subobjects of Sakkinen's inheritance model whereas the the notion of object at the provision layer corresponds to the notion of complex object. As such it seems that the Rondo object model can also be classified as an hybrid approach that combines the class-based object model with delegation. In this respect, Rondo can be further classified as supporting unanticipated extensibility and enforcing strong object identity.

The structures encapsulated by the combiner define a method environment where messages sent to the object are evaluated. The method environment has the structure of a table with an entry for each message supported by the object. The structure stored in the entry encodes the set of modules that jointly contribute to the definition of the message and the order in which these contributions should be executed. (i.e the corresponding methods implementing the message are related in an incremental modification chain).

We now return to the issue of how name collisions are resolved in the Rondo model. First, the Rondo model approaches conflicting state and conflicting methods differently. Since behavior modules are represented in the composition layer as "subobjects", the Rondo model provides like the hybrid approach a natural solution to homonymous state attributes.

As stated above, the Rondo model uses scope identifiers by means of which conflicting methods are resolved. This mechanism uniformly deals with replicated and homonymous methods.

Scope identifiers are defined as follows. Each method has a separate scope identifier. The scope identifier of a method `m` encodes the visibility scope of m. The visibility scope of `m` is defined as the set of behavior modules within which that method `m` is visible. These are the modules (related in an incremental modification chain) that jointly contribute to the implementation of `m` and all other modules that are specified to modify the former.

Scope identifiers are constructed as follows: each behavior module is marked with a unique label and the scope identifier of a method simply concatenates the labels of modules that are in the visibility scope of that method.

Scope identifiers are used at the composition layer as follows. For messages with several scope-specific definitions (i.e. either replicated methods or homonymous methods) , the corresponding entry in the method environment contains one subentry for each scope-specific definition. This subentry is indexed by the corresponding scope identifier. The structure stored in each subentry encodes the set of modules that jointly contribute to the corresponding scope-specific definition and the order in which these contributions should be executed. Messages with a single definition are a special case of those with multiple scope-specific definitions: their corresponding entry in the method environment has a single subentry.

The scope identifiers stored in the method environment are exploited by the dispatching and method lookup process of Rondo. When self calls from the code of the adjustments

already involved in the behavior definition of the object happen, only those definitions are visible to the calling adjustment whose associated scope identifier is compatible with (includes) the label assigned to the caller adjustment.

Although the above solution is elegant from a language run-time engineering point of view, the Rondo programming model does not provide the right abstractions for dealing with name collisions that occur when non-self calls are sent from message-passing clients. Public methods visible to message-passing clients are just as vulnerable to name collisions, and yet, the above described mechanism of Rondo does not enable clients to specify which scope-specific definition of the message must be selected. This is because the labels of modules 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.

This problem with non-self calls did not appear in the solution of the adapted rule for overriding. This is because types are used here for expressing visibility scopes and types are modelled in the domain of the application and, therefore, have a clear meaning in the domain of message-passing clients as well. As such it seems that although the mechanism of scope identifiers sufficiently applies separation of concerns at the language design space to effectively control visibility scopes, the mechanism does have a meaning in the domain of message-passing clients.

Furthermore, the Rondo object model breaks in the presence of independent evolution. Suppose the situation of a base class with two adjustments that independently define two homonymous methods named `bang()`. If the base class was to be modified over time to include a `bang()` method as well, the `bang()` methods of the two adjustments would suddenly be considered as belonging to the same visibility scope.

Finally, since the Rondo object model supports composition of independently developed components it has to deal with the common ancestor dilemma as well. Given the fact the Rondo object model supports dynamic composition as well and behavior modules are represented as objects it is clear that the Rondo model must also side-step the splitting problem by having the designer explicitly divide common ancestors in two modules (an S and an R module) which respectively are to be shared and replicated in the object.

7.5 Revisiting object-based composition

The hybrid approaches deals quiet good with the problems of the wrapper-based approach. Some hybrid approaches suffer from the spaghetti reference problem and duplicate state problem.

Günther Kniesel argues that the object schizophrenia problem in the wrapper-based approach is due to limitations of the underlying object model. In languages that support object-based inheritance the late binding of self is brought at the level of object composition. As such, delegation can be used to form a common self across webs of objects, one could term such webs themselves as objects of a higher order [Szy98]. Therefore the object schizophrenia problem does not appear in languages that support object-based inheritance.

Since type transparency is inherently provided in the hybrid approach (a wrapper aggregate is a subtype of the dynamic wrappee type), the problems that wrappers have the

tendency to spread due to the lack of type transparency (see section 2.4 is alleviated as well.

With respect to spaghetti references and duplicate state, the existing hybrid approaches can be further classified according to their ability to hide the object identity of component instances or the need to expose it, as either languages that support *strong*[BD96, Mez98] or *weak*[SM95, Kni99] object identity.

Languages, which support weak object identity, allow to express context-sensitive selection of components naturally but as argued in section 2, the usefulness of this approach is limited due to the spaghetti reference problem and the duplicate state problem. In languages with strong object identity, child components lose their full object status. spaghetti references and duplicate state can be controlled, but one needs an additional abstraction for expressing context-sensitive selection [BD96].

8 Conclusion

This paper investigated how well software composition is supported by the three principal object-oriented compositional models: object-based composition, class-based inheritance, and object-based inheritance.

We formulated three key goals in the context of software advanced composition: dynamic and context-sensitive composition, composition and evolution of independently developed components, and unanticipated extensibility.

None of the object-oriented compositional models satisfy all three key goals simultaneously, nor do they completely satisfy them without introducing problems. Each compositional model makes a trade-off: it supports one or two *typical key goals* while sacrificing other key goals. These trade-offs cannot explained in one simple sentence. In reality, they are subtle and difficult to grasp. In this sense, this paper has not presented simple truths. For example, it would be naive to conclude that object-based composition supports dynamic composition and unanticipated extensibility whereas class-based inheritance supports independently developed components and unanticipated extensibility. It would be even more naive to conclude that object-based inheritance supports all three key goals because - as its name suggests - it seemingly combines object-based composition and class-based inheritance. In reality, none of compositional models is able to achieve its typical key goals without introducing a lot of hard problems. In a sense, we would better approximate the reality by concluding that the different compositional models have their own typical problems and object-based inheritance is the worst of all because it combines them all.

It is well-known that these problems arise because the basic mechanisms (incremental modification, visibility control and dynamic object modification) are not orthogonalized from each other.

The hybrid approaches on the other hand are quite successful in achieving the three key goals because it has shown that, although the basic mechanisms within each of the existing compositional models are not orthogonal, the three compositional models are. By making some simple and unified adaptation to the method lookup and dispatch process, it is possible to combine the three compositional models without suffering semantical interference (i.e. they can be used in isolation from each other). As a result the hybrid approach succeeds quite well

in supporting all three key goals simultaneously, because the different compositional models are best at supporting *different* typical key goals and, therefore, their union yields all three key goals. Moreover, the three compositional models nicely complement each other. This is for example witnessed in the common ancestor dilemma case where class-based inheritance is used to express replication while delegation is used to express sharing.

Again this is not a simple truth. Some problems remain to persist in the hybrid approach. First there is a trade-off between independent evolution and unanticipated extensibility. In order to resolve this trade-off, visibility control must be separated from the incremental modification mechanism that governs method overriding. Using interface type information brings you quite far in the hybrid approach, but in order for this to work the programmer must follow some coding conventions.

Type information cannot be used to keep replicated methods separated from each other, nor can it be used to enable direct repeated inheritance. The Rondo object model has introduced scope identifiers to deal with the latter problem. Like type information, scope identifiers should be modelled in the domain of the application such that they have a meaning to message-passing clients.

Second, due to object-based composition, the common ancestor dilemma can only be resolved during class design by explicitly dividing classes in shared and replicated parts in advance. This puts a burden on the programmer but this practice conforms to what is advised in the static class-based inheritance model as well.

Finally, the hybrid approaches that allow strong object identity suffer from the spaghetti reference problem and the duplicate state problem.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, 1996.

[ALS03] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Ninth Workshop on Hot Topic in Operating Systems (HotOS-IX)*, May 2003.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*, pages 303–311, 1990.

[BD96] Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 122–137. ACM Press, 1996.

[BL91] G. Bracha and G. Lindstrom. Modularity Meets Inheritance. Technical Report UUCS-91-017, University of Utah, Dept. of Computer Science, 1991.

[BRSW00] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. Role object. In Niel Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Language of Program Design 4*, pages 15–32. Addison Wesley, 2000.

[BW00] 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.

[CG90] Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. In Norman Meyrowitz, editor, *OOPSLA/ECOOP '90 Proceedings*, pages 312–321. ACM SIGPLAN, 1990.

[CHOT99] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN Notices, 34(10), pages 325–339. ACM Press, 1999.

[DMSV89] 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.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gun01] Eric Gunnerson. *A Programmer's Introduction to C#, Second Edition*. APress, June 2001.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, 1993.

[HO02] William Harrison and Harold Ossher. Member-group relationships among objects. on-line proceedings of the AOSD'2002 workshop on Foundations of Aspect-Oriented Languages, http://www.cs.iastate.edu/ leavens/FOAL/papers-2002/index.html, April 2002.

[Höl93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 36–56, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Kni98] Günther Kniesel. Encapsulation = Visibility + Accessibility. Technical Report IAI-TR-96-12, Institut für Informatik III, Universität Bonn, 1996 (revised 1998).

[Kni99] 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.

[Knu88] 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.

[Lam93] John Lamping. Typing the specialization interface. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 201–214, 1993.

[Lie86] 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.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155. ACM SIGPLAN Notices 22(12), 1987.

[Mez97] 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.

[Mez98] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publishers, 1998.

[MvL96] Tom Mens and Marc van Limberghen. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[MW95] B. Meyer and N. Wilson. Eiffel: The Reference. Technical Report TR-EI-41/ER, ISE, 1995.

[NT95] Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[Ode00] Martin Odersky. Objects + views = components? In *Abstract State Machines 2000*, volume 1912 of *Lecture Notes in Computer Science*, pages 50–68. Springer-Verlag, March 2000.

[Sak89] 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.

[SDNB03] 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.

[SG99] 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.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 268–285. ACM Press, 1996.

[SM95] Patrick Steyaert and Wolfgang De Meuter. A marriage of class-and object-based inheritance without unwanted children. In *ECOOP'95-Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 127–144, August 1995.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86 Conference Proceedings*, ACM SIGPLAN Notices, 21(11), pages 38–45. ACM Press, September 1986.

[Sny87] 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.

[Ste94] P. Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, 1998.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 22(12) of *SIGPLAN Notices*, pages 227–242, December 1987.