

**Inheritance Modeled with
Explicit Bindings:
An Approach to Typed Inheritance**

Franz J. Hauck

June 1993

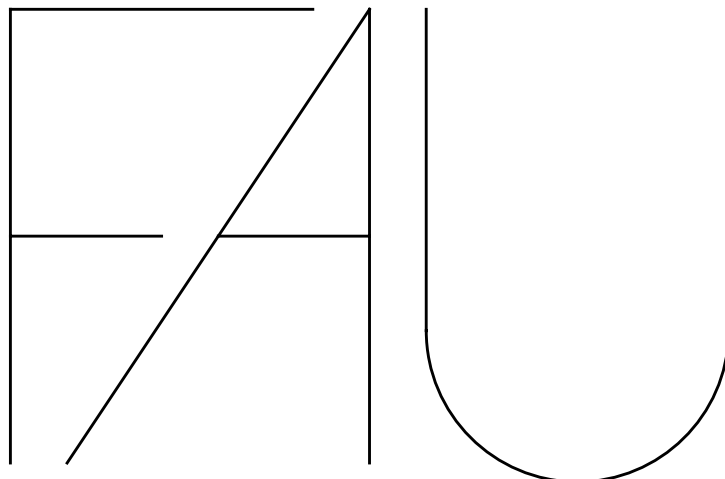
TR-I4-3-93

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This paper is also published as:

Franz J. Hauck: "Inheritance modeled with explicit bindings: an approach to typed inheritance; In: A. Paepcke [Ed.]; *Proc. of the Conf. on Obj.-Oriented Progr. Sys., Lang., and Appl.* – OOPSLA (Washington, D.C., USA, Sep. 26-Oct. 1, 1993); SIGPLAN Notices 28(10); ACM, New York, NY; Oct. 1993

Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance

Franz J. Hauck

hauck@informatik.uni-erlangen.de
University of Erlangen-Nürnberg, IMMD 4
Martensstraße 1, P.O. Box 3429
D-91051 Erlangen, Germany

Abstract. While there are suitable type-systems for aggregation that allow the exchange of objects and classes, the configuration of systems, and dynamic binding, there are no type-systems for inheritance relationships. Thus, maintenance of class libraries becomes difficult or even impossible because changes in a class library can cause changes in the inheriting classes in an application program. Modeling inheritance with explicit and parametrical bindings introduces a typed interface for inheritance which allows type-safe changes in class libraries. Furthermore the approach opens new possibilities for composition and makes programming easier.

1 Introduction

Inheritance is a basic concept of object-oriented programming and is, according to Wegner [1], intrinsic to the definition of *object-oriented*. Languages without inheritance are only called *object-based*. Inheritance composes new classes using already existing ones. Besides inheritance another method for composition is used. This method is *aggregation*. In terms of object-orientation aggregation is the composition of objects by references between objects. References to objects are bound to *variables* defined in objects.

For aggregation there are type-systems which allow abstraction from the concrete object bound to a variable. Variables are typed and all objects bound to these vari-

1. This work is supported by the *Deutsche Forschungsgemeinschaft DFG* in the Sonderforschungsbereich *SFB 182 Project B2*.

Permission to copy without fee all or part of this material is granted that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ables must have a conforming type. Thus, types allow a type-safe dynamic rebinding of references to different objects.

When using inheritance the changing of a base class of a class is not intended. Object references are often changed dynamically at run-time, while inheritance relationships usually are not².

Having a closer look at class libraries we can see cases in which it is necessary to exchange classes within a library. These classes may be used by inheritance in applications. An exchange of classes may be necessary for bug-fixing or improvement. In languages like *Eiffel* [2] or *C++* [3], library maintenance personnel cannot exchange any classes of a class library in a type-safe way. An added new method in a library class can already compromise inheriting classes (subclasses) if a method with the same name accidentally appears in a subclass.

This paper presents an approach to typed inheritance relationships which allows a type-safe exchange of classes in class libraries.

In chapter 2 we introduce a small object model upon which our approach is based. Both methods of composition, aggregation and inheritance, are presented. Reasons for a typed inheritance concept are given. Chapter 3 shows the properties of inheritance and introduces our approach to typed inheritance by modeling it with explicit bindings and aggregation. In chapter 4 we evaluate our approach and emphasize some new possibilities, e. g. parametrical base classes. Chapter 5 concludes the results.

2. Here we do not consider any languages which allow dynamic inheritance relationships.

2 Typed Inheritance

This chapter describes a small object model which introduces both composition methods, aggregation and inheritance. Typed inheritance relationships are motivated. The connection between inheritance and subtyping is explained at the end of this chapter.

2.1 Composition by Aggregation

An object consists of named variables³. A reference to another object (or to itself) can be bound to a variable by assignment. Bindings can be variable or constant. The composition of objects by bindings of references to variables is called *aggregation*.

In fig. 2.1 two objects *B* and *C* are bound to an object *A*, i.e. *A* has a reference to *B* and one to *C*. These bindings may be variable and changed at run-time. This corresponds to the concept of pointers in *C++* and to variables in *Eiffel*. Constant bindings are represented by object declarations in *C++* and by *expanded objects* in *Eiffel*.

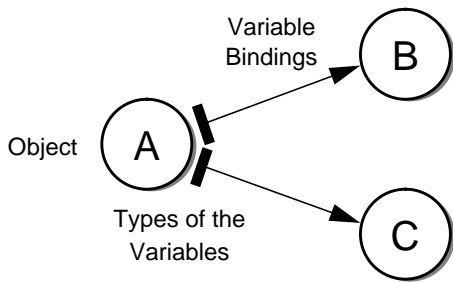


Fig. 2.1 Bindings between objects – Aggregation

Bindings can be established at the creation time of an object. Such bindings are called *initial*. Objects which are initially and constantly bound are called *private*.

In fig. 2.1 the variables are typed. The types are represented by thick cross-beams. Each object has a type. To bind an object to a variable, the object must have a type which conforms to the type of the variable. The types of the variables and objects are a kind of contract between client and server object. The typing of objects and variables allows a type-safe exchange of objects by rebinding. A new object has only to have a conforming type. For conformance the contra-variant type-rule applies

3. These are often called *instance variables*.

[4]. Informally a type *A* conforms to a type *B* when an object of type *A* can be used in every context where an object of type *B* can be used.

2.2 Composition by Inheritance

Inheritance in languages such as *C++* or *Eiffel* is a method for composition between classes rather than between objects. In fig. 2.2a a class *S* is shown which inherits from a class *B*. The usual view is that *S* includes all properties of the base class *B* – represented in the figure as a circle *B* enclosed by a circle *S*.

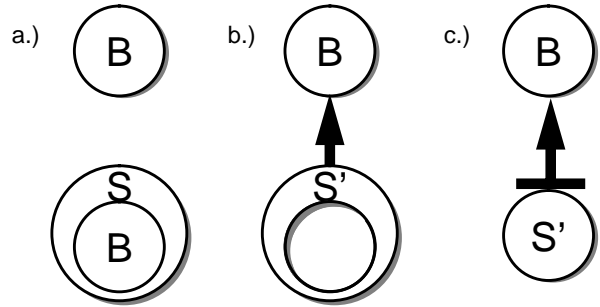


Fig. 2.2 Different graphical representations of inheritance relationships

The objects of class *S* have all the internal properties of an object of class *B*. Thus, fig. 2.2a can be seen as a representation of objects rather than classes: object *S* contains an object *B*.

Fig. 2.2b shows the relationship between class *S* and *B* more explicit. Class *S'* refers to class *B* by an ‘inherits-from’ arrow. *S'* is only a differential class representing the difference between *S* and *B* of fig. 2.2a. Thus, inheritance is a relationship between classes, or differential classes, represented as a thick arrow. Because the graphical representation can also be interpreted for objects rather than classes, fig. 2.2b also shows an object of class *S'* with a special relationship to an object of class *B*. This is not a common view of inheritance, e.g. not the same as in *C++*, but in an implementation it is easy to find the special relationship between the objects: a *C++* implementation creates for each object of class *S'* an object of class *B* which is inlined in the memory of the object of class *S'*.

In fig. 2.2c the class *S'* is drawn as a normal class. Additionally a typed interface is drawn as a cross-beam. There are no such typed interfaces in traditional lan-

guages. Sometimes there are so-called interfaces for inheritance, e.g. in C++, but these interfaces cannot abstract from the definitions of base classes. Particularly, there cannot be two classes with the same or conforming interfaces, i.e. these interfaces are not useful for typing and do not support the exchange of base classes. A typed interface as shown in fig. 2.2c allows a type-safe exchange of a base class, e.g. a class from a class library.

2.3 Why Typed Inheritance?

The introduction of typed interfaces for base classes and subclasses allows a type-safe exchange of a base class without the need of adapting or changing the subclass. Thus, the type of the inheritance interface is a kind of contract between base and subclasses.

An exchange of base classes is necessary for maintenance, e.g. for bug-fixing, in class libraries. In this case application programmers should not be forced to change their programs when a library class is changed. Recompiling or rebinding of the application should be sufficient. This is only possible if there is a typed interface between base and subclasses. An improved base class in a library must be type-conforming to the inheritance interface of the prior class.

In addition to typed interfaces there is the possibility of determining the concrete base class on a per object basis. The base class could be determined in an extra configuration phase similar to the aggregation relationship in distributed systems as shown in [5].

Using a typed interface for inheritance, an object can bind to an object of its base class at creation time. This binding can be procured by a name-server or broker similar to the dynamic bindings of aggregation relationships.

2.4 Remarks to Types

Up to now, we have not considered the representation of types. Languages as C++ and Eiffel combine the concepts of types and classes. The separation of both concepts has meanwhile been largely accepted, see for example [6].

In the following sections we assume that types have their own representation in our object model and that type-based inheritance is independent of class-based inheritance. Type-based inheritance allows the composition of types from existing types, class-based inheritance allows the composition of classes from classes. The separation of classes and types needs a mechanism for combining individually described types and classes. Therefore, each class decides of which type it is and defines a projection from the signatures in its type to its own methods or variables. Inheritance between types can be the basis for subtyping. Thus, subtyping is not related to class-based inheritance.

3 Modeling Inheritance by Explicit Bindings

To model inheritance by explicit bindings we have to analyze the abilities of inheritance. We will show how to model inheritance by aggregation and how a typed interface for inheriting classes is created.

3.1 What is Inheritance?

Inheritance has three important properties for a subclass⁴:

- The subclass can use all accessible components of a base class as in an aggregation relationship (fig. 3.1a). In this figure a thin arrow is drawn from the subclass to the base class.
- The subclass can define additional components – methods and variables (fig. 3.1b). In this figure only methods are shown to simplify matters.
- The subclass can define additional methods which replace methods defined in the base class. This means that all invocations of replaced methods are redirected to the method in the subclass (fig. 3.1c). In this picture a thin black arrow is drawn from the base class to the subclass. The thin grey arrow indicates the original binding of the thin black arrow – without inheritance from the base class. The thin arrow from the subclass to the base class shows that

4. We talk about classes in this section, although only the objects of these classes use the mentioned properties.

the subclass can access the old and replaced methods by explicit invocations, e.g. by a keyword *super* as known from *Smalltalk* [7].

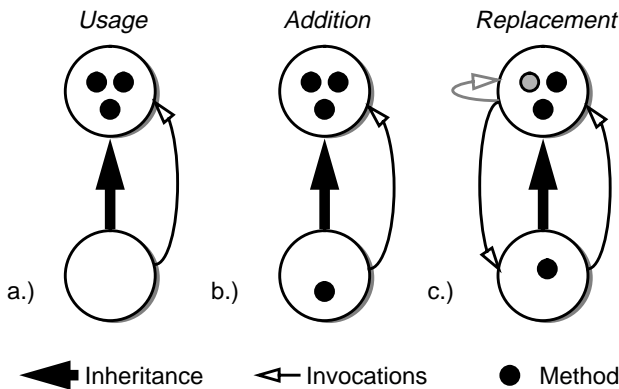


Fig. 3.1 Relationships from a subclass to a base class

Inheritance does not mean here that all subclasses are automatically subtypes, too. Subtyping has to be described independently (cf. section 2.4).

For a better understanding we demonstrate these three properties in an example. Let us think about a class *port*, whose objects have the ability to service a hardware port. We can output some characters to the port. Therefore, the class defines a method named *Putchar*. For the output of lines another method named *Putline* is defined. *Putline* is implemented by successive invocations of *Putchar* to output every single character of a line. In fig. 3.2 a strongly simplified description of the class *port* is given in a fictitious syntax. The keyword *self* is added to each method call for clarity. In most languages there is no need to write *self* explicitly at each call-statement.

```
port : class
{
  Putchar : ()->()
  { ... }

  Putline : ()->()
  {
    ...
    while ...
      self.Putchar();
  }
}
```

Fig. 3.2 A class for port objects – classic approach

In the chosen syntax, names of variables and methods are written on the left side of a colon, declarations or definitions on the right. Declarations of any formal parameters are left out for simplicity.

```
bufferedport : class inherits port
{
  Putchar : ()->()
  {
    ...
    self.Flush();
    ...
  }

  Flush : ()->()
  {
    ...
    while ...
      super.Putchar();
  }
}
```

Fig. 3.3 A class for buffered port objects – classic approach

To build a class which realizes a buffered output behavior for a hardware port we want to use inheritance. The class *bufferedport* (fig. 3.3) redefines and replaces the method *Putchar* with a buffering version. All characters are inserted in a buffer. When the buffer is full an additional method *Flush* is called which invokes the original *Putchar* method to flush all the characters of the buffer. Therefore, the keyword *super* is used.

The invocation of the method *Putline* calls the method of class *port* and the subsequent *Putchar* invocations in *Putline* are directed to the new buffering method of class *bufferedport*.

3.2 Inheritance by Aggregation

The fig. 3.1 and the *port* example from fig. 3.2 and fig. 3.3 already give some hints on how to model inheritance by aggregation. The keywords *self* and *super* are modeled by explicit variables. The variable *super* is bound to a private object of the base class. This variable is initially bound at creation time of the object of a subclass and the binding is constant⁵. The variable *self* is initially and constantly bound to the object of the subclass. Fig. 3.4 shows an object of class *B* in inheritance and aggregation relationships. In fig. 3.4a the bindings for a subclass *S* inheriting from *B* are shown⁶.

Because the variable *self* is defined in the base class even when an object of the base class is not in an inheritance relationship, it is necessary to define how this variable is bound for the use of an object in an aggrega-

5. Loosening this constraint allows the modeling of dynamic inheritance relationships, but this is not a topic of this paper.
6. For simplicity, in this figure only the binding for the *self* variable of the base class and not the *self* variable of the subclass is represented.

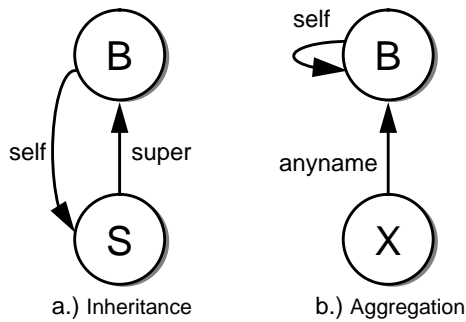


Fig. 3.4 Explicit binding of *self* and *super* in an inheritance and in an aggregation relationship

tion relationship only. For this case the variable is bound to the object itself. This situation is shown in fig. 3.4b.

The *self* variable of an object of the base class is variously bound depending on the aggregation or inheritance relationship the object is in. The binding is always initial and constant. Thus, it can be seen as a parameter for the creation of an object.

3.3 Typing

The typing of inheritance relationships has to cover simultaneously the bindings of the *super* variable of the subclass object and the *self* variable of the base class object. The type of the *super* variable describes the interface of the base class object. This type can be extended to describe the interface to inheritance. Therefore, the parametrical binding of the *self* variable is included into the type as a kind of publicly accessible variable. It is different from usual public variables because it is bound at creation time and constant for the remaining lifetime of the object.

According to the contra-variant type-rule a type with an additional parametrical binding conforms if all other methods and parametrical bindings conform. Thus, an improved library class can define additional methods and additional parametrical bindings while maintaining a conforming type.

Fig. 3.5 is the same as fig. 3.4 but shows the types of the variables *super* and *anyname* as a thick beam. The binding of *self* in the base class object is attached to the type of the *super* variable as a public variable, shown as a thin line from the beam to the binding arrow. Fig. 3.5a

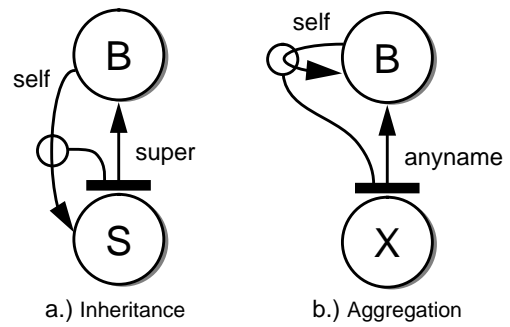


Fig. 3.5 Typed interface of the binding in an inheritance and in an aggregation relationship

shows the object *S* and *B* in an inheritance relationship and fig. 3.5b sketches the objects *X* and *B* in an aggregation relationship.

```
port_t : type
{
  self : parm port_t;

  Putchar : ()->();
  Putline : ()->();
}

port : class of type port_t
{
  self : parm port_t = here;

  Putchar : ()->()
  { ... }

  Putline : ()->()
  {
    ...
    while ...
      self.Putchar();
  }
}
```

Fig. 3.6 Type *port_t* and class *port* with a parametrical binding of the variable *self*

Our approach can now be demonstrated by using the example from section 3.1. Fig. 3.6 shows the same example with a special aggregation relationship instead of inheritance. First of all a type for a port-object is described and named *port_t*⁷. The class *port* is defined to be of type *port_t*. The projection of names in the type to the names in the class definition is here implicitly done between methods and variables with the same name. Additional syntactical statements are necessary to allow explicit projections (renaming).

The class *port* defines an additional variable *self*. It is of type *port_t* which means that it can bind every object

7. We name types according to an old C tradition with a suffix '*_t*'.

with a type conforming to *port_t*. The keyword *parm* indicates that the binding is constant and initial and can be set at the creation time of the object. The variable is part of the type. *Self* is defined with a default binding to the created object itself. Therefore, the keyword *here* is used to name the current object. Contrary to the traditional keyword *self* (e.g. in *Smalltalk*), *here* is not affected by inheritance, but always names the object in which it is used. The definitions of all other components of class *port* remain the same as in fig. 3.2.

```
var : port_t = port.new;
```

Fig. 3.7 A variable *var* bound to a new object of class *port*.

An object of class *port* can be created in an aggregation relationship as shown in Fig. 3.7. The *self* variable of the object is bound to the object itself according to the definition in the class *port*. This corresponds to fig. 3.5b.

```
bufferedport_t : type o> port_t
{
  self : parm bufferedport_t;

  Putchar : ()->();
  Putline : ()->();
  Flush : ()->();
}

bufferedport : class of type bufferedport_t
{
  self : parm bufferedport_t= here;
  super : priv port_t= port.new(self=here);

  Putchar : ()->()
  {
    ...
    self.Flush();
    ...
  }

  Putline :- super.Putline;

  Flush : ()->()
  {
    ...
    while ...
      super.Putchar();
  }
}
```

Fig. 3.8 Type *bufferedport_t* and class *bufferedport* using a private object of class *port* due to inheritance

For the class *bufferedport* we define a type *bufferedport_t* which conforms to the type *port_t*. This is defined using the operator ‘o>’ which means ‘conforms to’. The operator ‘<o’ means conformance in the opposite direction.

The type *bufferedport_t* conforms to *port_t* although it has a public variable *self* which has a different type than the corresponding *self* variable in *port_t*. This is possible because the types of *self* conform and the *self* variables are read-only. Variables of a type corresponds to a read and a write method. A read-only variable only has a read method.

In the class *bufferedport* we do not use a keyword to inherit from class *port*, but we define a *super* variable, which is bound to a private object of class *port*. Privacy is declared by the keyword *priv*. Within the parentheses the parametrical binding of *self* in *port* is set to *here* which denotes the current object of class *bufferedport*.

Because objects of class *bufferedport* do not have a method with the name *Putline*, the name *Putline* is declared in *bufferedport* referring to the *Putline* method of the object bound to *super*. Therefore, the operator ‘:-’ instead of a colon is used. This operator defines an alias for the name at the right side.

The type *port_t* serves here as a type for the inheritance relationship between the classes *bufferedport* and *port*. It declares the parametrical binding of the *self* variable of *port* objects. The type of this variable indicates that in an inheritance relationship *port* objects call methods in the subclass just as they are defined in type *port_t*. Indeed *port* objects only call the *Putchar* method via *self*.

Improved or changed versions of class *port* have to conform to type *port_t*. Then they could be used instead of class *port* for the initial binding of *super* in *bufferedport*. This means that an improved version must not call additional methods via *self* as those defined in type *port_t*. If this is necessary a second parametrical binding has to be introduced to the new class.

3.4 Repeated Inheritance

Up to now we have only considered two classes in an inheritance relationship, but classes can inherit from classes which are subclasses in another inheritance relationship. Thus, there can be chains of subclasses. Our approach to model inheritance by aggregation has not covered this case. In the example (fig. 3.6) there was a parametrical binding for *self*, but it would not have the desired effect for a subclass of *bufferedport*. Normally

all subclasses in an inheritance chain have the same bindings for their individual *self* variable. The variables are bound to the last subclass in the chain. This has to be modeled accordingly. Our example will only work when *bufferedport* is the last subclass. Then, both *self* variables are bound to the *bufferedport* object.

```
bufferedport : class of type bufferedport_t
{
  self : parm bufferedport_t= here;
  super : priv port_t=port.new(self=self);
  ...
}
```

Fig. 3.9 Correction for repeated inheritance

For the general case the code of class *bufferedport* in the example has to be changed as shown in fig. 3.9. *Self* is defined as before, but the definition of *super* has been changed. Instead of binding the *self* variable in *port* to *here* of *bufferedport* it is bound to the same object as the *self* variable of *bufferedport*⁸. Thus, *self* of *port* has even the same binding as *self* of *bufferedport* when *bufferedport* is used as a base class for subsequent inheritance.

3.5 Multiple Inheritance

Multiple inheritance, i.e. inheritance with more than one base class, is easily described with our approach. We use multiple *super* variables, one for each base class, and they all have parametrical bindings for their *self* variables. The variables for the private base class objects must have different names. There is no need to call them *super*, but the names can be chosen in an arbitrary way.

It is even possible to have two private base class objects of the same class, e.g. a window class inheriting twice from a border class. This is not possible in C++, for example, because a subclass must name its base classes rather than base class objects. Thus, a subclass can have only one base class object per class.

8. In the syntax of the setting of a parametrical binding (the assignment in parentheses) there has to be a name on the left side of the equal sign which is valid in the base class – here *self* of *port* – and there has to be an expression using names of the defining class on the right side – here *self* of *bufferedport*.

3.6 Smaller Types for the *Self* Variables

As found at the end of section 3.3 the *self* variable in class *port* is defined with type *port_t*, but only the *Putchar* method is invoked via *self*. This shows that the type of the *self* variables could be smaller than the type of the class where they are defined. A larger type conforms to a smaller type.

```
selfport_t : type <o port_t
{
  Putchar : ()->();
}

port : class of type port_t
{
  self : parm selfport_t= here;
  ...
}
```

Fig. 3.10 Minimal type *selfport_t* for *self* in *port* and the changed class *port*

In our example we introduce a type *selfport_t* which covers the minimal (smallest) type for *self* in *port* (fig. 3.10). The type *port_t* is declared conforming to *selfport_t*. The *self* variable gets the new type. The type *port_t* has to be corrected in the same way.

```
selfbufferedport_t : type <o bufferedport_t,
                      o> selfport_t
{
  Putchar : ()->();
  Flush : ()->();
}

bufferedport : class of type bufferedport_t
{
  self : parm selfbufferedport_t= here;
  ...
}
```

Fig. 3.11 Minimal type *selfbufferedport_t* for *self* in *bufferedport* and the changed class *bufferedport*

The class *bufferedport* and the corresponding type must be changed accordingly. *Selfbufferedport_t* is declared conforming to *selfport_t* and *bufferedport_t* is declared conforming to *selfbufferedport_t*. These declarations allow the correct typing of the parametrical bindings of *super* and *self*. There are several statements in the code of the example in which the declared conformance relationships are needed. They are sketched in fig. 3.12.

```
port_t <o bufferedport_t
needed to use bufferedport objects as port objects
```

$selfport_t <_O port_t$
 needed for the assignment of *here* to *self* in *port*

$selfbufferedport_t <_O bufferedport_t$
 needed for the assignment of *here* to *self* in *bufferedport*

$selfport_t <_O selfbufferedport_t$
 needed for the parametrical binding of *self* in *port* to *self* in *bufferedport*

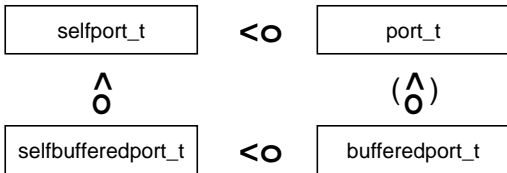


Fig. 3.12 Conformance relationships between the introduced types

We can argue that the conformance relationships sketched in fig. 3.12 must be valid for all inheritance relationships. The types of the *self* variable are smaller or equal to the type of the object in which they are defined. This observation was made before by Cook in his theoretical approach to the semantics of inheritance [8].

The conformance of *bufferedport_t* to *port_t* is not necessary for the definitions of the classes *bufferedport* and *port*. Thus, this conformance is not necessary for all inheritance relationships and inheritance must not always cause type-conformance. Only the conformance between *selfbufferedport_t* and *selfport_t* is required.

4 Evaluation of the Aggregation Approach

4.1 Inheritance

Besides the type-safe exchange of base classes the aggregation approach to inheritance has additional advantages. It is possible to decide for each method call whether the call can be redirected to the last subclass in an inheritance chain (call via *self*) or not (call via *here*). In C++ the method declaration decides a possible redirection of method calls – when the keyword *virtual* is placed in the declaration all method calls are executed via *self* and not via *here*. In *Smalltalk* the programmer cannot decide at all. All method calls are executed in the last subclass.

It is possible to have more than one parametrical binding for different “*self*-like” variables. These can serve different purposes and the inheriting subclass can decide which bindings are set by parameters and which are left with their default bindings. The mechanism of parametrical bindings allows a subclass to control which method calls of the base classes are invoked in itself.

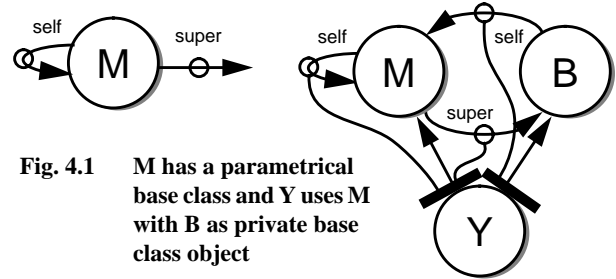


Fig. 4.1 M has a parametrical base class and Y uses M with B as private base class object

It is also possible to have a parametrical base class using a parametrical *super* binding in addition to the *self* binding (fig. 4.1). In this case the *super* binding has no default binding and has to be set at creation time. In fig. 4.1 an object *M* has a parametrical *super* binding. An object *Y* uses (creates) an object *M* and an object *B*. *B* is created as a private base class object for *M* and bound to the parametrical *super* binding of *M*. Therefore, *B* has to be type-conforming to the type of the *super* variable in *M*. Fig. 4.2 shows the class *y* of object *Y*. The class *m* of object *M* is here a kind of *mixin* which is an abstract subclass. Further research is necessary to compare the possibilities of the aggregation approach with *mixin*-based inheritance, e.g. the approach of [9].

```

y : class
{
  M : m_t= m.new( self= M, super= B );
  B : b_t= b.new( self= M );
  ...
}

```

Fig. 4.2 Syntactic version of the class *y* of object *Y*

Fork-join-inheritance can be modeled in a similar way. This is multiple inheritance with shared base class objects [10].

4.2 Composition

Independent of inheritance, parametrical bindings can be used for composition beyond traditional inheritance. Thus, complex object structures can be defined by connecting new objects with parametrical bindings.

Composition of an object system is no longer done by two different mechanisms – inheritance and aggregation. Instead, the programmer always uses aggregation and adds parametrical bindings when needed.

4.3 Distribution

One side effect of the proposed approach is the easy distribution of objects in an inheritance relationship. The process of distribution of objects in an aggregation relationship using special mechanisms for object addressing is well known, e.g. using client and server stubs. The inheritance relationship is, in fact, an aggregation relationship and, thus, distributable.

Up to now, there have been no investigations of how to distribute objects of base class and subclass. Now, distribution is possible but certainly not always useful.

4.4 Syntactic Support

With a suitable programming language it should be easy to provide the usual inheritance syntax and transform it according to the presented approach. This makes the programming of base classes easier because the *self* variable is implicitly declared and the program text gets smaller. At the same time all mentioned advantages could be used, but there is a need for more exploration in the area of the type system because a syntactic support will need automatic type inference for the type of the *self* variable.

5 Conclusion

We presented a concept to model class-based inheritance by aggregation. Our approach provides a type-concept for inheritance relationships which is a must for type-safe exchange of base classes in class libraries.

The introduced parametrical bindings which are set at object creation time and which are integrated to the type of an object allow compositions of objects beyond the traditional inheritance facilities.

6 Acknowledgments

I would like to thank my colleague and friend, Thomas Eirich, who is also responsible for early ideas on the

topic. I also want to thank all the other colleagues, especially of the PM project group, who helped to improve my understanding of the topic through many discussions. Especially I want to thank Patrick R. Steyaert and the OOPSLA reviewers for the valuable comments from outside my group.

7 References

- [1] P. Wegner: “Dimensions of object-based language design”; In: *Proc. of the Conf. on Obj.-Oriented Progr. Sys., Lang., and Appl.* – OOPSLA; N. Meyrowitz [Ed.], (Orlando, Fla., Oct. 4-8, 1987); SIGPLAN Notices 22(12); ACM, New York, NY, USA; Dec. 1987 – pp. 168-182
- [2] B. Meyer: *Eiffel: the language*; Prentice Hall, New York, NY; 1992
- [3] M.A. Ellis, B. Stroustrup: *The annotated C++ reference manual – ANSI base document*; Addison-Wesley, Reading, Mass., USA; 1990
- [4] L. Cardelli: “A semantics of multiple inheritance”; In: *Semantics of Data Types*; G. Kahn, D. B. McQueen, G. Plotkin [Eds.]; Lecture Notes on Comp. Sci. 173; Springer, 1984 – pp. 51-68
- [5] M. Fäustle: *Beschreibung der Verteilung in objektorientierten Systemen*; Dissertation, IMMD, Univ. of Erlangen-Nürnberg; Arbeitsber. d. IMMD 25(8); Erlangen, Germany; Sep. 1992
- [6] W. R. Cook, W. L. Hill, P. S. Canning: “Inheritance is not subtyping”; *Conf. record of the 17th Symp. on Princ. of Progr. Lang.* – POPL, (San Francisco, Cal., Jan. 17-19, 1990); 1990 – pp.125-135
- [7] A. Goldberg, D. Robson: *Smalltalk-80: the language and its implementation*; Addison-Wesley, Reading, Mass., USA; 1983
- [8] W. R. Cook: *A denotational semantics of inheritance*; PhD Thesis, Brown University, 1989
- [9] G. Bracha, W. Cook: “Mixin-based inheritance”; In: *Proc. of the Conf. on Obj.-Oriented Progr. Sys., Lang., and Appl. / Eur. Conf. on Obj.-Oriented Progr.* – OOPSLA / ECOOP (Ottawa, Ont., Canada, Oct. 21-25, 1990); SIGPLAN Notices 25(10); ACM, New York, NY, USA; Oct. 1990 – pp. 303-311
- [10] M. Sakkinen: “A critique of the inheritance principles of C++”; *USENIX, Comp. Sys.* 5(1); Univ. of Calif. Press, Berkeley, Cal., USA; 1992 – pp.69-110