

Class-Based Inheritance is Not a Basic Concept

F. J. Hauck

July 1993

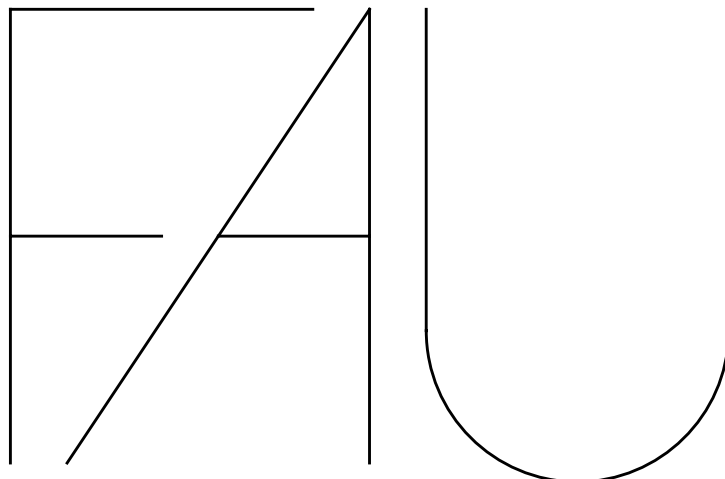
TR-I4-6-93

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This paper was submitted and accepted as a position paper to the OOPSLA '93 Workshop W19 'Understanding Object-Model Concepts'

Class-Based Inheritance is Not a Basic Concept

OOPSLA '93 Workshop W19 – Position Paper

Franz J. Hauck¹

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

Abstract. Class-based inheritance is normally seen as a basic concept and a prerequisite of object-oriented programming. This position paper states that class-based inheritance is not essential for an object-oriented programming language. Instead of class-based inheritance aggregation, parametrical bindings, and aliasing can be used. Aggregation and parametrical bindings are basic concepts of an object model which are not only useful for inheritance. Aliasing is syntactic sugar for forwarding of method invocations to other objects.

1 Introduction

According to the dimensions of Peter Wegner an object-oriented language has to have the notion of objects, classes, and class-based inheritance [1]. This position paper argues that class-based inheritance is not essential for an object-oriented programming language. Inheritance can be modeled by aggregation and parametrical bindings. Aggregation is the composition of objects by references. Parametrical bindings are variables which can be set at the object creation time.

In the second chapter an object model is introduced which forms the basis for our approach. The model supports objects, variables, and references to objects. Types are independent from classes. The third chapter explains how to model inheritance by aggregation. Chapter four concludes.

2 Object Model

2.1 Composition by Aggregation

An object consists of named variables. A reference to another object (or to itself) can be bound to a variable

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

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++* [2] and to variables in *Eiffel* [3]. Constant bindings are represented by object declarations in *C++* and by *expanded objects* in *Eiffel*.

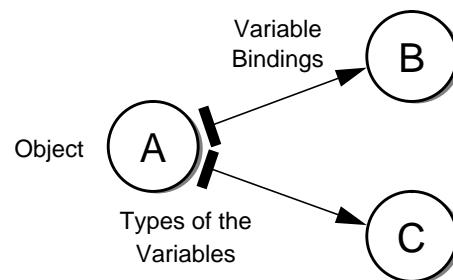


Fig. 2.1 Bindings between objects – Aggregation

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*.

Classes are not essential for our approach, but they are used in the forthcoming examples. Thus, the object model has the notion of classes to describe objects of the same shape. Then, classes are templates for object creation.

2.2 Typing

Languages as *C++* and *Eiffel* combine the concepts of types and classes. The separation of both concepts has meanwhile been largely accepted, see in [4] for a motivation.

In the following sections we assume that types have their own representation in the 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 Inheritance

3.1 What is Inheritance?

To model inheritance by explicit bindings we have to analyze the abilities of inheritance. Inheritance has three important properties for a subclass:

- 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 the subclass can access the old and replaced methods by explicit invocations, e.g. by a keyword *super* as known from *Smalltalk* [5].

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

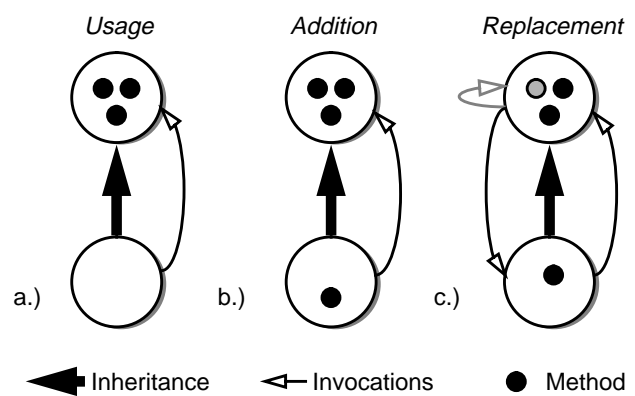


Fig. 3.1 Relationships from a subclass to a base class

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

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

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

Fig. 3.3 A class for buffered 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.

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 constant¹. 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 shown².

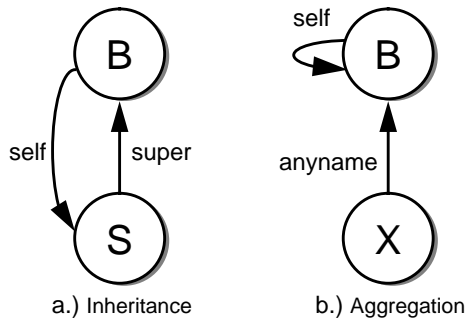


Fig. 3.4 Explicit binding of *self* and *super* in an inheritance and in an aggregation relationship

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-

1. Loosening this constraint could allow the modeling of dynamic inheritance relationships, but this is not a topic of this paper.
 2. 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.

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 Example

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*³. 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).

```
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. 3.5 Type *port_t* and class *port* with a parametrical binding of the variable *self*

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

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

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. 3.6 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=self);

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

  Putline :- super.Putline;

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

Fig. 3.7 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*¹. 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* vari-

ables 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 the same object as *self* in *bufferedport* refers to². This allows correct repeated inheritance from *bufferedport*. There, both *self* variables are changed by a parametrical binding.

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. Aliases are syntactic short hands for forwarding methods.

3.4 Multiple Inheritance

Multiple inheritance, i.e. inheritance with more than one base class, is easily described with the 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.

3.5 Benefits

What are the advantages and disadvantages of the aggregational approach to inheritance? An important advantage is the possibility of typing. Typed inheritance

2. 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*.

1. We do not introduce any notion for type-inheritance in this paper.

relationships allow a type-safe exchange of base classes in class libraries. This makes maintenance of class libraries much easier and safer. See in [6] for further details.

A disadvantage is that the programmer has to decide which method calls are invoked via *self* and which via *here*. This corresponds to the decision of a C programmer who has to define a method *virtual* or not. C programmers have to decide on a per method basis. *Smalltalk* programmers cannot decide at all. In *Smalltalk* this is an disadvantage because programmers cannot prevent a method from redefinition in a base class.

We believe that one advantage of the approach is the similarity to aggregation. Thus, incidental inheritance [7] cannot happen because it does not need a parametrical binding and in our approach inheritance without using a parametrical binding is aggregation only.

Another interesting fact is that the approach outlines how to distribute an object of a base class. All classes of the class hierarchy are used in an aggregational way. For aggregation the distribution of objects is well understood and implemented in many distributed systems. Thus, the distribution of inheriting objects seems to come for free. Certainly it is not always useful to distribute these objects but it is possible.

4 Conclusion

This position paper showed how to model inheritance by aggregation and parametrical bindings. A prerequisite is a separate type system which is independent of classes. The modeling technique can be used in a programming language without a lack of expressivity as shown in chapter 3.

The approach has several advantages as typing of inheritance, full control of redefined method calls, elimination of incidental inheritance, and easy distribution.

5 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] M.A. Ellis, B. Stroustrup: *The annotated C++ reference manual – ANSI base document*; Addison-Wesley, Reading, Mass., USA; 1990

[3] B. Meyer: *Eiffel: the language*; Prentice Hall, New York, NY; 1992

[4] 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

[5] A. Goldberg, D. Robson: *Smalltalk-80: the language and its implementation*; Addison-Wesley, Reading, Mass., USA; 1983

[6] F. J. Hauck: "Inheritance modeled with explicit bindings: an approach to typed inheritance"; *Proc. of the Conf. on Object-Oriented Progr. Sys., Lang., and Appl.* – OOPSLA, (Washington, D.C., Sep. 26-Oct. 1, 1993); SIGPLAN Notices 28(10), Oct. 1993

[7] M. Sakkinen: "Disciplined inheritance"; In: *Proc. of the Eur. Conf. on Object-Oriented Progr.* – ECOOP, (Nottingham, UK, July 10-14, 1989); Cambridge University Press, Cambridge, UK; 1989 – pp. 39-56