

Inheritance by Aggregation

T. Eirich, F. J. Hauck

November 1991

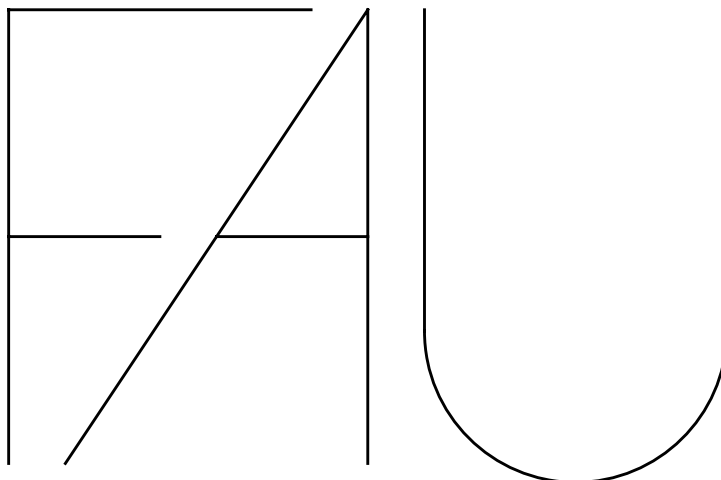
TR-I4-4-91

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This paper was submitted to the 1991 Conference on Object-Oriented Programming Systems, Languages, and Applications — OOPSLA.

Inheritance by Aggregation

Thomas Eirich

eirich@informatik.uni-erlangen.de

Franz J. Hauck

hauck@informatik.uni-erlangen.de

University Erlangen-Nürnberg, IMMD 4

Martensstraße 1

D-W-8520 Erlangen

Abstract

Inheritance is an important concept in object-oriented programming. The mechanisms provided by many existing programming languages combine reuse of types and reuse of implementations. These mechanisms often violate encapsulation. Changes to an inheritance hierarchy affect related implementations.

We separate type and class (implementation) and identify visibility of attributes and the binding of attributes to names as basic dimensions of class-based inheritance. To retain encapsulation [12] we state requirements to an inheritance mechanism and show how to fulfill them with an approach to inheritance by aggregation. Initialization is used to change bindings of attributes.

1. Introduction

Inheritance is an important concept in object-oriented programming languages. It is used for differential programming, refinement and reuse. In existing languages inheritance is realized by a special mechanism which ought to fulfill these aspects.

The major advantage of object-oriented programming is the encapsulation of objects. Each object may only be manipulated through interfaces. An interface describes the publicly usable behavior of an object and is thus a contract between the user of an object and its implementor. The user may at most rely on the contract and the implementor must at least fulfill the contract. The implementation of an object may be changed without affecting any users as long as the contract remains fulfilled. Usual inheritance mechanisms often violate the encapsulation, e.g. by allowing access to instance variables of a superclass or by visibility of the inheritance hierarchy.

This paper wants to work out the effects of the concept of inheritance and we will state requirements to save encapsulation in case of inheritance. One step towards this goal is to separate the notions of type and implementation. The consequences of this step are separate inheritance mechanisms for type and implementation.

Under the condition of retaining encapsulation even on inheritance and separating types and implementations, we will express inheritance between objects by aggregation and initialization. It is not necessary to introduce a particular mechanism to realize the concepts of inheritance.

Chapter two explains our basic concepts. Chapter three motivates the separation of type and implementation and shows how to reach this goal. Chapter four discusses some dimensions of class-based inheritance. From the discussion we deduce some requirements for class-based inheritance and we present our approach based upon aggregation

and initialization in chapter five. Finally chapter six summarizes our approach.

2. Basic concepts

We are assuming a class-based object model. In this model objects are typed and classes are used to create objects. The following paragraphs describe basic properties of types, objects and classes.

- The concepts to model a problem [6] are expressed by types. A type specifies the behavior of a concept and abstracts from the details of a specific implementation. The description of a type at least contains the signature of the operations. It should comprise a formal specification of the abstract behavior. Unfortunately a complete formal specification is rather complex and not fully under control. There are approaches to enhance the type by assertions, as in Eiffel [7], or predicates, as in POOL-I [1]. Informal specifications in form of comments may be added.
- Objects have the following properties: identity, integrity, types and attributes [9]. Integrity means strict encapsulation of the attributes. Access to the object is only possible by operations (methods). The invocation mechanism like message passing or (remote) procedure call is not important for our considerations.
- A class describes an implementation of types and is used to create objects of these types.

3. Separating Type and Implementation

In most object-oriented programming languages type and implementation are combined in classes. This approach has two severe disadvantages.

First a type cannot have multiple implementations e.g. a type **Set** cannot be implemented by a hash table, an array or a linked list, since a new class defines a new type. In most object-oriented languages this flaw is corrected by abstract and concrete classes: deferred classes in Eiffel [7], abstract classes with virtual functions in C++ [13]. Descriptive classes proposed by Sandberg [8] are a similar approach. Abstract classes are degenerated classes which are used to describe a type by omitting the implementation. Concrete classes provide the implementation and they must inherit from abstract ones to become a subtype. Thus the abstract class serves as common supertype of all its concrete subclasses.

The use of abstract classes has some drawbacks. Each class must be separated into an abstract and a concrete one, otherwise there will be no way to have multiple implementations subsequently.

Code reuse requires multiple inheritance, because a class must inherit from its abstract and another concrete class. This causes the usual name conflicts of multiple inheritance. Inheritance between classes is used to express type relationships (inheritance between abstract classes), code reuse (between concrete classes) and the relation between implementations and a type (between abstract and concrete classes). Fig. 2 illustrates these various applications.

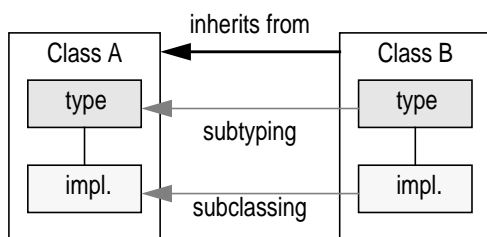


Fig. 1: Inheritance between classes and the implied relationships

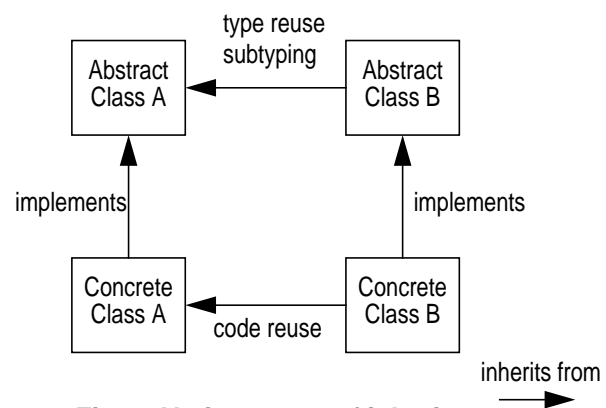


Fig. 2: Various uses of inheritance

The second disadvantage is, that the implementation and the type hierarchy are identical, since inheritance between classes implies subtyping as well as code reuse (Fig. 1). But both hierarchies may be contrary as shown by some examples in the literature, e.g. circle-ellipse in [4] or stack-deque in [12]. This problem has already been identified by Snyder [11] and Canning et. al. [2]. The hierarchies are separated in the language POOL-I [1], since America distinguishes clearly between inheritance and subtyping.

We advocate the separation of type and implementation. This leads to individual inheritance mechanisms on types and on implementation resulting in separated hierarchies. This enables the programmer to change the implementation hierarchy without being forced to change the type hierarchy, too. The programmer has more freedom to change implementations. The relationships between types and classes are each expressed by their own language constructs and are therefore better coordinated to specific requirements.

The most obvious effect is an n-to-m relationship between types and classes. A type may be implemented by several unrelated classes as well as a class may implement multiple types. The different types of a class expose separate aspects of its behavior e.g. test and debug interface to an object, lower and upper service access point to a protocol object or a separate interface for inheritance. This treatment observes the need-to-know principle. An operating system may associate access rights with the different interfaces of a class to install access privileges.

Types and classes are described separately. The language must provide the ability to link classes with types. The linkage means that the class implements the type. It must be part of the description of the class, since it refers to internals of the class. These internals should not be made public to save encapsulation.

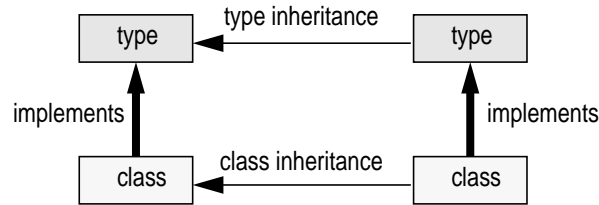


Fig. 3: Inheritance and linkage between classes and types

The separate description of types and classes is more expensive than formulating the type together with a class, but using concrete and abstract classes leads to comparable expense. In languages, which have combined the notions of type and class, all those inheritance relationships which use type and class in the same way are more simple to express. This is the case when subtyping and subclassing coincide.

In the rest of this chapter we will explain how to link classes to types by means of an imaginary language.

The first example depicts a type **2Dpt** (1a) and a possible implementation **2Dpoint** (1b)¹.

```

1a type 2Dpt {
    X() → ( Real );
    Y() → ( Real );
    PlaceAt( Real, Real ) → ();
}

1b class 2Dpoint {
1c   support 2Dpt {
1d     bind Y to gety, PlaceAt to placeAt;
    };

    Real cx, cy;

    x() → ( Real ) { return cx; }
    gety() → ( Real ) { return cy; }
    placeAt( Real x, Real y ) → () { cx := x; cy := y; }
}

```

Example 1

The **support**-clause (1c) describes the linkage (depicted by the *implements* arrow in Fig. 3) between the class and its types. Operations de-

¹ For reasons of readability types and names in type descriptions start with capital all others with lower case letters. To provide an easy way to refer to the examples they are partially numbered, e.g. 1a, 2b etc.

defined in the type are bound to operations of the class (1d). If the names of both operations are the same, an explicit linkage may be omitted as shown for the operation x .

Multiple and independent implementations per type are simply realized by writing some classes which support the same type. The class **2DpolarPt** (2a) is another implementation for the type **2Dpt**. Objects of both implementations may coexist in a single program.

```

2a class 2DpolarPt {
    support 2Dpt;

    Real alpha, radius;

    x() → ( Real ) { return radius * cos( alpha ); }
    y() → ( Real ) { return radius * sin( alpha); }

    placeAt( Real x, Real, y ) → ( ) {
        alpha := arctan2( x, y );
        radius := sqrt( x*x + y*y );
    }
}

```

Example 2

Multiple types per class are obtained by multiple **support** clauses (3c), (3d). The class **point** (3b) implements a type **3Dpt** (3a) and **2Dpt** (1a). The two types do not conform to each other in any direction, since the operation **PlaceAt** has a different number of arguments. With multiple types per class it is possible to write a class which implements both types.

4. Class-Based Inheritance

As depicted in Fig. 3 there are three relations between types and classes. The linkage between types and classes has been discussed in chapter 3. The type inheritance is not considered in this paper instead we focus on class-based inheritance.

```

3a type 3Dpt {
    X() → ( Real )
    Y() → ( Real )
    Z() → ( Real )
    PlaceAt ( Real, Real, Real ) → ()
}

3b class point {
3c   support 2Dpt { bind PlaceAt to 2DplaceAt; };
3d   support 3Dpt { bind PlaceAt to 3DplaceAt; };

    Real cx, cy, cz;

    x() → ( Real ) { return cx; }
    y() → ( Real ) { return cy; }
    z() → ( Real ) { return cz; }

    2DplaceAt( Real x, Real, y ) → () {
        cx := x; cy := y;
    }

    3DplaceAt( Real x, Real, y, Real z ) → () {
        2DplaceAt(x,y); cz := z;
    }
}

```

Example 3

This chapter will show some dimensions of inheritance. From Snyder's discussion about inheritance and encapsulation, [11] and [12], we infer requirements for class-based inheritance.

The main dimensions of inheritance are visibility and binding.

Visibility

The attributes of a class comprise both variables and methods. Classes access attributes by names. These constitute a name space of visible attributes. The name space of a class contains the names of all attributes defined in the class itself. Classes determine which of their visible attributes become visible to clients.

If the class inherits from others its name space is extended by parts of the name spaces of the superclasses. It varies in each object-oriented language which attributes become visible and how they are named.

The set of attributes becoming visible to a subclass ranges from all attributes of the superclasses to the restricted view of a client. In-between

this range there are C++ [13] and Trellis/Owl [10] which restrict visibility for clients but not for subclasses with the predicate *protected*. All *public* and *protected* attributes are subclass visible. Other languages permit the subclass to access all attributes of the superclass.

The construction of the new name space may lead to vertical or horizontal name collisions [5]. Collisions must be solved in any case. A collision may be an error. One attribute may override another. The conflict is solved by renaming or by compound names.

A subclass inherits from its superclasses and these may in turn inherit from other classes. These inheritance relationships form a graph starting at a root class. In case of single inheritance the graph is linear, in case of multiple inheritance it is a tree. If instances of the same superclass are merged it becomes an acyclic graph. The structure of the inheritance graph has impact on the visibility of attributes in the root class.

The name spaces of a class and its ascendants do not change if it is later used as a superclass.

Binding

Binding is the relation of names to attributes. We distinguish fixed and variable relations. Variable bindings are changed by inheritance. The binding in a part of an inheritance graph is determined by the complete graph. Often the names are bound to the first occurrence of an attribute with the same name which is found when the graph is searched starting at the root class. The name space of a class and thus visibility is inferable from the description of that class, in contrary to the binding. The binding may change if the class is subject to inheritance.

In most languages variable bindings are denoted by keywords like *self*, *me* or *this*. Cook et. al. identify recursive structures with variable bindings [3].

4.1 Requirements to Visibility

This section points out some restrictions on visibility to preserve encapsulation of classes.

First of all subclasses must interact with superclasses by an interface. This implies that no direct access to instance variables of superclasses is possible. Operations must be called instead. Syntactic sugar may be introduced to recover conciseness and inlining of code to recover efficiency. An interface contributes to retain strict encapsulation and specifies the asserted properties. It should be possible to expose more or less details of the implementation. The more details are uncovered the more difficult it is to change the implementation. The interface should be independent of the client interface.

Second, in contrast to the type hierarchy the class hierarchy must not be visible. The type hierarchy must be visible to enable polymorphism and subtyping. The decision of a programmer to reuse implementation from classes by inheritance should be as private as other implementation details. This saves the possibility to change the implementation without adversely affecting subclasses. The consequence is that classes only know their direct superclasses. The behavior of the superclasses is for the subclass like its very own. It is not necessary for the subclass to know how the behavior of the superclasses is achieved.

The third requirement applies to merging. We refer to [12] and advocate that shared ancestors should be avoided. A solution of the problem is the use of “mixin”-classes [12].

4.2 Requirements to Binding

A language must support fixed binding. Otherwise a class would not be able to prevent inadvertent rebinding of attributes.

A subclass must be able but not forced to change variable bindings of a superclass. Consider instrumenting a class through the use of inheritance. Only external calls should be counted. In this case redirection should not take place because calls internal to the superclass would increment the counters in the subclass. Another problem might be to count every call. Thus redirection should take place. In most languages there is no way to express solutions for both problems, since subclasses may not decide whether or not to change variable bindings.

5. Inheritance by Aggregation and Initialization

The last section stated some requirements to class-based inheritance. This chapter will introduce our approach to inheritance by aggregation and initialization and we will demonstrate that it fulfills all demands.

Aggregation is a composition of objects. An object refers to its aggregated objects by instance variables. Aggregated objects are always part of the aggregating object.

The aspect of visibility corresponds to aggregation. Binding is realized by initialization. First we consider aggregation and how to fulfill the requirements for visibility. Then we add initialization and show how to achieve variable bindings with respect to the requirements.

5.1 Interface for Inheritance

Assume a raw port which puts out a character to a hardware. The type and an implementation is given with (4a) and (4b).

We want to add a method to send a complete string. First of all we introduce a new type **Port** (5a) which conforms to **RawPort** and offers an

```

4a type RawPort {
    PutChar( Char ) → ();
}
4b class rawPort {
    support RawPort;
    putChar( Char c ) → () {
        ... }
}

```

Example 4

additional method **PutLine**. **Port** is related to **RawPort** by the **is** clause which expresses type inheritance (5b).

```

5a type Port {
5b   is RawPort;
    PutLine( String ) → ();
}

```

Example 5

An implementation of **Port** wants to exploit the existing class **rawPort**. In the same way as the types are related an implementation of **Port** may be related to an implementation of **RawPort**. The class **port** (6a) illustrates the relation on the implementation side expressed by aggregation.

```

6a class port {
6b   support Port { bind subtype RawPort to p; }
6c   RawPort:rawPort p;
    putLine( String str ) → () {
        for( ... ) {
6d       p.PutChar( c );
        }
    }
}

```

Example 6

The **bind** statement (6b) links all operations of the type **RawPort** to the instance variable **p**. The invocation of the operation **PutChar** on type **Port** leads to the invocation of **p.PutChar**. The instance variable **p** is defined by its type **RawPort** and its class **rawPort** separated by a colon (6c).

The instance variable **p** serves as instance of the “superclass” **rawPort**. The type **RawPort** of **p** serves as interface for inheritance. In this exam-

ple it is not different to the client interface. We will see an interface different to the client interface in the next section.

5.2 Invisible Tree Structured Hierarchy

The relationship between aggregated objects and its aggregating object is a tree structured hierarchy.

Multiple inheritance is achieved by multiple aggregation and multiple **bind** statements. Direct access to a “superclass” is qualified by the name of its instance variable. Even multiple superclasses of the same type can be accessed individually.

The name conflicts related to external interfaces are resolved by **support** clauses. Internal name conflicts cannot occur since attributes of the “superclasses” are addressed by compound names.

Since a “subclass” accesses a “superclass” only through a well-defined interface it cannot know its internal structure. The “superclasses” in turn may use aggregation to realize the specified behavior. But it is not visible.

5.3 Fixed and Variable Binding

To access an attribute of a “superclass” by the name of its instance variable corresponds to fixed binding (6d). It is not possible to express variable bindings solely by aggregation. Therefore we need a way to influence existing class descriptions. Variable binding is realized by parameters given to initialize new objects.

In our next example we want to enhance the class **port** to have a buffered output behavior. The new **putLine** method should have a buffered behavior too. There is no way to realize that with a new class using an

instance of class **port** as shown previously and reusing the implementation of **putLine**. This is because the implementor of **port** did use only fixed bindings. For a buffered **port** we need a variable binding to **putChar**.

A second implementation of the type **Port** is given in (7b). It is prepared for the desired reuse.

```

7a type Port_Inh {
    Init( RawPort ) → ();
}

7b class port2( RawPort ) {
7c   support Port { bind subtype RawPort to p; }
7d   support Port_Inh;

    RawPort:rawPort p;
7e   RawPort s := RawPort:self;

    putLine( String str ) → () {
7f     for( ... ) {
        s.PutChar( c );
      }
    }

7g   init( RawPort parm ) → () {
        if( parm!=nil )
            s:= parm;
    }
}

```

Example 7

Port_Inh is an additional type to class **port2** for inheritance. The operation **Init** enables the redirection of variable bindings. The keyword **self** (7e) denotes the current instance of class **port2**. The Variable **s** is connected to the current instance viewed through the interface of type **RawPort** (7e). By an invocation of **init** (7g) the connection of **s** might be changed to redirect variable bindings (7f).

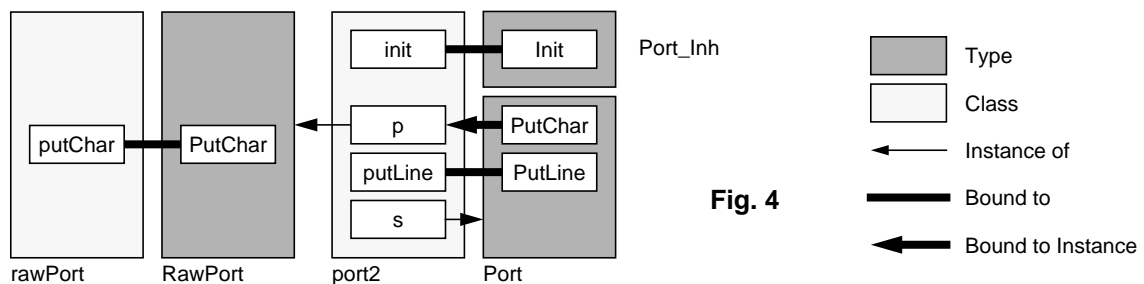


Fig. 4

Fig. 4 shows the dependencies between the related types and classes. Types are drawn dark, classes light. Thin arrows mean instance relationships, thick lines mean bindings from type to class names. Thick arrows bind type names to an instance variable.

The class **bufferedPort** is now implemented using an instance of class **port2** and redirecting the variable binding (Example 8).

```

8a type BufferedPort {
    is Port;
    Flush() → ();
}

8b type BufferedPort_Inh {
    Init( BufferedPort ) → ();
}

8c class bufferedPort( BufferedPort ) {
8d   support BufferedPort {
        bind subtype Port
        except PutChar to p; }
    support BufferedPort_Inh;

    Port:port2 p;
    BufferedPort s := self;
    Buffer:buffer b;

    putChar( Char c ) → () {
        if( b.Full )
            s.Flush;
        b.Insert( c );
    }

    flush() → () {
        for( ... ) {
            p.PutChar( c );
        }
    }

    init( BufferedPort parm ) {
        if( parm!=nil )
            s:= parm;

        Port_Inh:p.init( RawPort:s );
    }
}

```

Example 8

The new class overrides the **PutChar** operation and implements it with a buffered behavior. An instance of class **port2** is created to reuse the implementation of **PutLine** and **PutChar**. To redirect the call of **PutChar** in **PutLine** (7f) the instance of the new class is supplied as parameter to **init** of **p** (8f).²

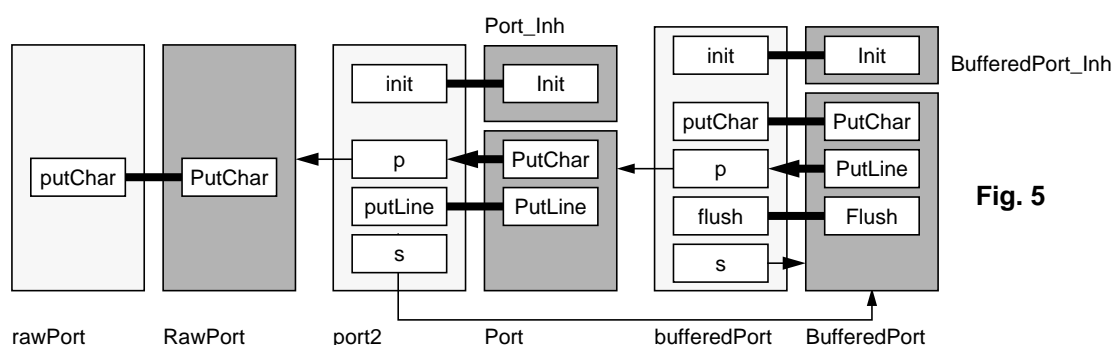


Fig. 5

² The instance **b** is omitted from Fig. 5 due to readability.

The binding of **PutChar** in **Flush** is variable and may be redirected for reuse. This redirection could be decoupled from the redirection of **PutChar** by providing two different parameters to **init**.

5.4 Decision of Rebinding

A class reusing others may decide to redirect variable bindings or not. There is no need to redirect all variable bindings. A good example is an instrumented **Port** which counts the invocations of **putChar** and **putLine** operations (Example 9).

```

9a class instrumentedPort( RawPort ) {
    support InstrumentedPort;
    support InstrumentedPort_Inh;

    Port:port2 p;
    Int putCharCnt, putLineCnt;

    putChar( Char c ) → () {
        putCharCnt.increment;
        p.PutChar( c );
    }

    putLine( String s ) → () {
        putLineCnt.increment;
        p.PutLine( s );
    }
}

9b
...
init( RawPort parm ) → () {
    putCharCnt:= 0;
    putLineCnt:= 0;

    if( parm==nil )
        Port_Inh:p.init( nil );
    else
        Port_Inh:p.init( parm );
}

```

Example 9

Only external calls are counted. The internal calls, the call of **putChar** in **putLine** (7f), may also be counted with a small change of the **init** function, (9b) to (9b’).

```

9b'
...
if( parm==nil )
    Port_Inh:p.init( self );
else
    ...

```

Example 10

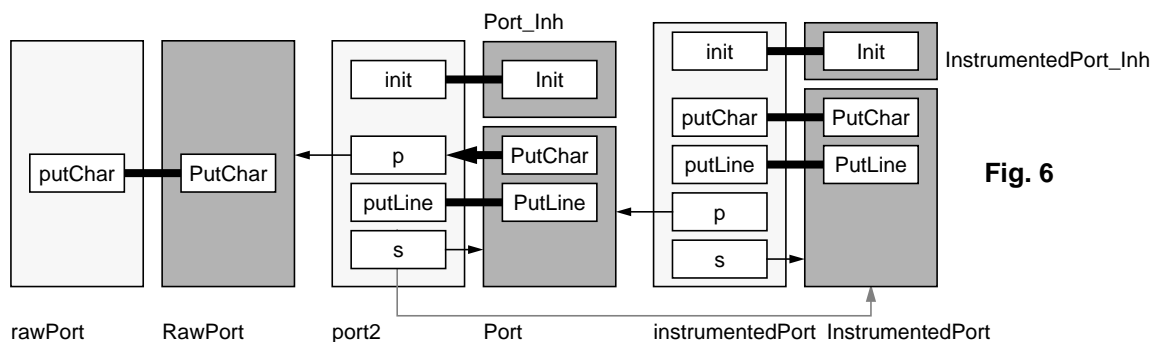


Fig. 6

The grey thin arrow shows the change of the instance relationship caused by a different **init** function³.

There could be a second parameter to decide which version is to create. In most languages without changing the implementation of the super-class only one of the two versions could be built.

6. Conclusion

We showed that there is no need for an extra mechanism to realize class-based inheritance. Aggregation and initialization in addition to type based inheritance is sufficient.

The main goal was to save encapsulation even on class-based inheritance. We stated some requirements to achieve this goal and showed that they are satisfied by aggregation and initialization.

A central aspect in our approach is the interface for inheritance. The types of aggregated objects serve as interfaces. These could be normal client interfaces or special ones exposing more implementation details.

The separation of type and class simplifies the programming language because the relation between type and implementation is more clearly expressed than with abstract classes.

Inheritance modelled by aggregation and initialization increases the expressivity. Subclasses could decide to change variable bindings or not. There could be more than one group of variable bindings which may be changed individually. This is more powerful than the usual **self**-mechanism in existing programming languages.

³ The counters and the operations to get the counter results are omitted from Fig. 6 due to readability.

Type inheritance and its impact to type conformance will be considered in further work. In this paper we considered only variable binding to operations (attributes). Additionally variable bindings to classes and types should be solved and integrated to inheritance with respect to encapsulation.

The use of “mixin” classes to avoid merging is also possible in our approach. Initialization provides a way to connect instances by redirecting variable bindings. Thus instances of “mixin” classes may be connected horizontally. The use of “mixins” will be demonstrated in another paper.

7. Acknowledgments

This work was partially supported by the Deutsche Forschungsgesellschaft (DFG) in SFB 182. The considerations of this work are part of a project to design a programming environment for large scaled distributed object-oriented systems, called PM.

We would like to thank our colleagues for comments on this paper. The discussions with Michael Fäustle helped clarifying our concepts.

8. References

- 1 America, Pierre; van der Linden, Frank; “*A Parallel Object-Oriented Language with Inheritance and Subtyping*”; OOPSLA ‘90 Proceedings; pp. 161-168
- 2 Canning, Peter S.; Cook, William R.; Hill, Walter L.; Olthoff, Walter G.; “*Interfaces for Strongly-Typed Object-Oriented Programming*”; OOPSLA ‘89 Proceedings; 1989; pp. 457-467
- 3 Cook, W.; “*A Denotational Semantics of Inheritance and its Correctness*”; OOPSLA ‘89 Proceedings; 1989; pp. 433-443
- 4 Halbert, Daniel C.; O’Brien, Patrick D.; “*Using Types and Inheritance in Object-Oriented Languages*”; ECOOP ‘87 Proceedings, 1987; pp. 20-31
- 5 Knudsen, Jørgen Lindskov; “*Name Collision in Multiple Classification Hierarchies*”; ECOOP ‘88 Proceedings, 1988; pp. 93-109
- 6 Madsen, Ole Lehrmann; Møller-Pedersen, Birger; “*What object-oriented programming may be - and what it does not have to be*”; ECOOP ‘88 Proceedings, 1988; pp. 1-20

- 7 Meyer, Bertrand; *“Object-oriented Software Construction”*; Prentice Hall International; Hemel Hempstead, Hertfordshire; 1988
- 8 Sandberg, David; *“An Alternative To Subclassing”*; OOPSLA ‘86 Proceedings, 1986; pp. 424-428
- 9 Sakkinen, Markku; *“Disciplined Inheritance”*; ECOOP ‘89 Proceedings, 1989; pp. 39-56
- 10 Schaffert, Craig; Cooper, Topher; Bullis, Bruce; Kilian, Mike; Wilpolt, Carrie; *“An Introduction to Trellis/Owl”*; OOPSLA ‘86 Proceedings, 1986; pp. 9-16
- 11 Snyder, Alan; *“Encapsulation and Inheritance in Object-Oriented Programming Languages”*; OOPSLA ‘86 Proceedings, 1986; pp. 38-45
- 12 Snyder, Alan; *“Inheritance and the Development of Encapsulated Software Components”* in Research Directions in Object-Oriented Programming; Shriver B., Wegner P. (Eds.); MIT Press 1987; pp. 165-188
- 13 Stroustrup, Bjarne; *“The C++ Programming Language”*; Addison-Wesley; 1986