

# Virtual Classes

## A powerful mechanism in object-oriented programming

Ole Lehrmann Madsen

Computer Science Department, Aarhus University  
Ny Munkegade, DK-8000 Aarhus C, Denmark  
Tlf.: +45 6 12 71 88 - E-mail: olmadsen@daimi.dk

Birger Møller-Pedersen

Norwegian Computing Center  
P.O. Box 114, Blindern, N-0314 Oslo 3, Norway  
Tlf.: +47 2 45 35 00 - E-mail: birger@nr.uninett.no

### Abstract

The notions of class, subclass and virtual procedure are fairly well understood and recognized as some of the key concepts in object-oriented programming. The possibility of modifying a virtual procedure in a subclass is a powerful technique for specializing the general properties of the superclass.

In most object-oriented languages, the attributes of an object may be references to objects and (virtual) procedures. In Simula and BETA it is also possible to have class attributes. The power of class attributes has not yet been widely recognized. In BETA a class may also have *virtual class attributes*. This makes it possible to defer part of the specification of a class attribute to a subclass. In this sense virtual classes are analogous to virtual procedures. Virtual classes are mainly interesting within strongly typed languages where they provide a mechanism for defining general parameterized classes such as set, vector and list. In this sense they provide an alternative to generics.

Although the notion of virtual class originates from BETA, it is presented as a general language mechanism.

**Keywords:** languages, virtual procedure, virtual class, strong typing, parameterized class, generics, BETA, Simula, Eiffel, C++, Smalltalk

### 1 Introduction

The notions of class and subclass are some of the key language concepts associated with object-oriented pro-

gramming. Classes support the classification of objects with the same properties, and subclassing supports the specialization of the general properties. A class defines a set of attributes associated with each instance of the class. An attribute may be either an object reference (or just reference for short) or a procedure.

In a subclass it is possible to specialize the general properties defined in the superclass. This can be done by adding references and/or procedures. However, it is also possible to modify the procedures defined in the superclass. Modification can take place in different ways. In Simula 67 [4] a procedure attribute may be declared virtual. A virtual procedure may then be redefined in a subclass. A non-virtual procedure cannot be redefined<sup>1</sup>. This is essentially the same scheme adapted by C++ [16] and Eiffel [13]. In Smalltalk [6] any procedure is virtual in the sense that it can be redefined in a subclass, and even the parameters of a procedure may be redefined.

In BETA [8] a virtual procedure cannot be redefined in a subclass, but it may be further defined by an *extended* definition. The extended procedure is a "sub-procedure" (in the same way as for subclass) of the procedure defined in the superclass. This implies that the actions of a virtual procedure definition are automatically combined with the actions of the extended procedure in a subclass. This is the case for all levels of subclasses that further defines a virtual procedure. In Smalltalk and C++ it is the responsibility of the programmer to combine a redefined virtual procedure with the corresponding virtual procedure of the superclass. This is of course more flexible, since the programmer can ignore the procedure in the superclass. However, it is also a potential source of error since the programmer may forget to execute the virtual procedure from the superclass.

Using the terminology from [18] a class in BETA

Permission to copy without fee all or part of this material is granted provided 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.

© 1989 ACM 089791-333-7/89/0010/0397 \$1.50

<sup>1</sup>In Simula a subclass may declare a *new* procedure with the same name as a procedure defined in a superclass. This does not have the effect of a redefinition as in Smalltalk.

is *structural compatible* with its superclass whereas a Smalltalk class is *name compatible* with its superclass. Behavioral compatibility can only be obtained by proving that the further definitions do not violate the invariants of the superclass. For a more detailed discussion of virtual procedures in BETA see [9].

Simula and BETA are block-structured languages. This implies that classes and procedures can be arbitrarily nested. I.e. in Simula and BETA an object may have *class attributes* in addition to references and procedures. In [10] the usefulness of class attributes is discussed.

The subject of this paper is the introduction of the notion of *virtual class*<sup>2</sup>. A virtual class is similar to a virtual procedure in the sense that it may be extended in a subclass. Virtual classes are useful for defining "parameterized" general classes within a strongly typed language. Examples of such classes are sets, vectors and lists. Decisions about the element type of such classes should be deferred to the subclasses of the general class. The notion of virtual procedure makes it possible to defer part of the definition of a procedure to a subclass. In the same way a virtual class makes it possible to defer part of the definition of the type of references (instance variables). Virtual classes may be seen as an alternative to "generic" types as found in Ada and Eiffel.

The notion of virtual class is introduced in the context of strongly typed languages. The modification of a virtual class in a subclass should then be structurally compatible with the definition of the virtual class in the superclass. As for virtual procedures, a virtual class cannot be redefined in a subclass, but its definition may be extended.

The notion of virtual class has been developed as part of the BETA language. It is a consequence of the unification of classes, procedures, functions and types into one general abstraction mechanism, *the pattern*. The implications of this have been discussed elsewhere [8, 9]. As there is only one language mechanism for classes and procedures, the notion of subpattern applies equally well to classes and procedures. The notion of virtual pattern was developed as a generalization of the Simula notion of virtual procedure. Since a pattern may also be used as a class, this generalization was designed to include the notion of virtual class.

The notion of virtual class is, in this paper, introduced as a general language mechanism. It is attempted to present the idea as independent of BETA as possible. The language used in this paper is a modified version of BETA with two kinds of patterns: classes and procedures. In addition a certain amount of syntactic sugar has been added. The very terse syntax of BETA can

<sup>2</sup>The notion of virtual class introduced here has nothing to do with the notion of virtual class defined in [17]

```
CC: class C (# Decl1; Decl2; ...; Decln #)
```

Figure 1: Class declaration

often be a hindrance for readers unfamiliar with BETA. All the examples in the paper can be expressed in BETA by a simple replacement of keywords.

The notion of virtual class will be compared with generics as found in Ada and with the type system of Eiffel including the simple generics.

Certain language aspects of object-oriented languages will be ignored since they are irrelevant for this paper. This includes the notion of information hiding. Another aspect is the discussion of code sharing versus types. Subclassing is very often used for code sharing; the term *inheritance* underlines this. As pointed out in [2], this usage of subclassing may be conflicting with defining types. We tend to agree with this. However, in BETA subclassing is intended for modelling types (or concepts). The Mjølner BETA System [5] includes facilities for separating a class definition from its implementation.

## 2 Classes and virtual procedures

In this section the notation for classes, procedures and virtual procedures will be presented.

### 2.1 Class and subclasses

A class definition has the form described in Figure 1. It is a declaration of a class `CC` with superclass `C`. (If no superclass is specified, the superclass is `Object`.) `Decl1; Decl2; ... Decln` are declarations of attributes. An instance of `CC` will have these attributes in addition to those inherited from `C`. An attribute may be either a *reference* (to an object), a *(virtual) procedure* or a *(virtual) class*.

In Figure 2 an example of a class is given. Class `Window` is described as a subclass of class `Stream`. In addition to the attributes inherited from `Stream`, it contains two *static references* `UpperLeft` and `LowerRight`, one *dynamic reference* `Label`, a procedure attribute `Move` and a virtual procedure attribute `Display`.

The dynamic reference `Label` may denote instances of class `Text` and its subclasses. A dynamic reference may denote different objects during its life-time. In this way they are similar to qualified references in Simula, instance variables in Smalltalk, and non simple variables in Eiffel.

```

Window: class Stream
  (# UpperLeft,LowerRight: @ Point;
   Label: ~ Text;
   Move: proc (# ... #);
   Display: virtual proc (# ... #);
  #)

```

Figure 2: Example of class declaration

A static reference denote part-objects. A static reference will constantly denote the same object during the life-time of the enclosing object. Part objects are generated together with the generation of the enclosing object. A `Window` object will thus have two part-objects of class `Point`.

References have an associated "type" in the form of a class name. This class name will be referred to as the *qualification* of the reference. The reference `Label` is e.g. qualified by `Text`.

A *reference assignment* has the form

```
aTextObject [] -> aWindow.Label []
```

which has the effect that the object denoted by `aTextObject` is also denoted by the `Label` reference attribute of the object denoted by `aWindow`. The source denotation is an example of a *remote identifier* used to identify attributes of objects. Remote identifiers are also used for denoting procedure and class attributes.

As thoroughly discussed in [13], the difference between "reference" and "value" semantics of assignment and equality is important. BETA has both forms. A *value assignment* has the form:

```
aTextObject -> aWindow.Label
```

which describes that a "copy"<sup>3</sup> of the object `aTextObject` is assigned to `aWindow.Label`. As it may be seen, the syntax clearly distinguishes between reference and value assignment. The box `[]` indicates that the reference is assigned. In this paper value assignment is used for simple classes like `Integer`. Otherwise reference assignment is used.

## 2.2 Procedures and subprocedures

A procedure declaration has the form described in Figure 3. The procedure `PP` is a subprocedure of the procedure `P`. As for classes, `Decl1, Decl2, ... Decln` describe a set of attributes associated with a procedure object in addition to those inherited from the superprocedure. The *enter-part* (`enter In`) describes the input parameters, the *do-part* (`do Imp`) describes the actions

<sup>3</sup>In [8] it is described what is actually meant by "copy"

```

PP: proc P
  (# Decl1; Decl2; ...; Decln
   enter In
   do Imp
   exit Out
  #)

```

Figure 3: Procedure Declaration

```

OpenRecord: proc
  (# ID: ~Text; R: ~Record
   enter ID []
   do ID [] -> theDataBase.Open -> R [];
   INNER;
   R.Close
  #);
OpenWritableRecord: proc OpenRecord
  (# do R.Lock; INNER; R.Free #);
Foo: proc OpenWritableRecord
  (#
   do someData [] -> R.put
  #)

```

Figure 4: Example of prefixed procedure

to be executed and the *exit-part* (`exit Out`) describes the output parameters. A subprocedure will have *enter/exit* lists that are the concatenations of the lists from the superprocedure and the lists specified for the subprocedure.

In Figure 4 three procedure declarations are shown. `OpenWritableRecord` is a subprocedure of `OpenRecord` and `Foo` is a subprocedure of `OpenWritableRecord`

The `INNER` construct enables the actions of the superprocedure to be combined with the actions of the main part of the procedure. Consider the following procedure call:

```
someId [] -> Foo
```

An execution of `Foo` proceeds as follows:

1. An instance of `Foo` (a procedure-object) is created.
2. The actual input parameter (`someId`) is assigned to the reference `ID`.
3. `Foo` is executed:
  - (a) Execution starts with the actions of the top-most superclass, i.e. `OpenRecord`.
  - (b) Execution of `INNER` in `OpenRecord` will imply execution of the actions in the subprocedure, i.e. `OpenWritableRecord`.

```

Record: class
  (# Key: ^KeyType;
   Display: virtual proc
    (#
     do Key.Display; INNER;
    #)
  #);
Person: class Record
  (# Name: ^Text; Sex: ^SexType;
   Display: extended proc
    (#
     do Name.Display; Sex.Display; INNER
    #)
  #);
P: ^Person

```

Figure 5: Classes with virtual procedures

- (c) Execution of INNER in `OpenWritableRecord` will imply execution of the actions in `Foo`.
- (d) After execution of the actions in `Foo`, the actions following INNER in `OpenWritableRecord` and `OpenRecord` will be executed in that order.

Execution of `Foo` implies thus that the following sequence of actions is executed:

```

ID[] -> theDataBase.Open -> R[];
R.lock;
someData[] -> R.put;
R.Free;
R.Close

```

### 2.3 Virtual procedures

A procedure attribute may be specified to be virtual. This implies that its definition may be extended in a subclass of the class in which it is virtual. The extended virtual procedure will have the virtual procedure from the superclass as a superprocedure. A virtual procedure that has been extended is still virtual in the sense that it can be further extended in subclasses.

Consider the classes defined in Figure 5. The `Display`-procedure of a `Person`-object will be a subprocedure of the `Display`-procedure defined in class `Record`. Execution of `P.Display` will imply execution of `Key.Display`, `Name.Display`, `Sex.Display`, and possibly more since `P` is known to denote at least `Person`-objects.

### 2.4 Classes as attributes

As mentioned above an object may have class-attributes. As an example of class attributes consider Figure 6 which is the grammar example from [10].

```

Grammar: class
  (#
   Symbol: class Object
   (# isTerminal: proc (# ...#);
    isNonTerminal: proc (# ... #);
   #);
   ...
  #);
AdaGram: @ Grammar; S1,S2: @ AdaGram.Symbol;
PascalGram: @ Grammar; X1,X2: @ PascalGram.Symbol

```

Figure 6: Class Grammar

Class `Grammar` describes the structure of grammar objects. Part of the description of a grammar is the notion of *symbol* associated with a grammar. The class attribute `Symbol` describes the structure of symbols associated with a particular `Grammar`-instance. In the example two `Grammar` instances are declared: `AdaGram` and `PascalGram`. Each of these `Grammar` objects has an associated `Symbol` class as an attribute. It is possible to declare instances of the class `AdaGram.Symbol`. In the example the references `S1` and `S2` denote such objects. Similarly it is possible to declare instances of class `PascalGram.Symbol`, in the example `X1` and `X2`. By declaring `Symbol` local to `Grammar` it is possible to distinguish between symbols from different grammars. `S1`, `S2` and `X1`, `X2` are not instances of the same class. Also a `Symbol` class has no existence without a grammar object. From a modelling point of view this seems intuitively correct. From a technical point of view, a `Symbol` object may refer to (global) attributes in the enclosing `Grammar` object.

## 3 Virtual classes

The notion of virtual procedure introduced above resembles similar mechanisms in other languages, the difference being that a redefinition of a virtual procedure is an extension of the definition of the virtual procedure in the superclass.

In this section the property of being virtual is applied to class attributes, and a more comprehensive treatment of virtuals in general is given.

### 3.1 Locally qualified virtual classes

The specification of a virtual class is completely analogous to the specification of a virtual procedure. Consider Figure 7. The class `Graph` has class attributes `Node` and `Link` which define the elements of a graph. `Node` and `Link` are specified as virtual classes. Subclasses of `Graph`

```

Graph: class
  (# Node: virtual class
    (# Connected: @boolean #);
  Link: virtual class
    (# Source, Dest: ~ Node #);
  Root: ~ Node;
  Connect: virtual proc
    (# S,D: ~ Node; L: ~ Link
    enter(S[],D[])
    do new Link[] -> L[];
      S[] -> L.source[]; D[]-> L.Dest[];
      True -> S.Connected -> D.Connected;
      INNER
    #);
  #);
DisplayableGraph: class Graph
  (# Node: extended class
    (# DispSymb: ~DisplaySymbol #);
  Link: extended class
    (# DispLine: ~DisplayLine #);
  Connect: extended proc
    (# DL: ~DisplayLine
    enter DL[]
    do DL[] -> L.DispLine[]; INNER
    #);
  Display: virtual proc (# .... #)
  #);
TravellingSalesmanGraph: class Graph
  (# Node: extended class (# Name: ~Text #);
  Link: extended class
    (# Distance: @ Integer #);
  Connect: extended proc
    (# D: @Integer
    enter D
    do D -> L.Distance; INNER
    #);
  #);
DG: ~ DisplayableGraph;
TG: ~ TravellingSalesmanGraph

```

Figure 7: Locally qualified virtual classes

may extend the definitions of `Node` and `Link` corresponding to specific different kinds of graphs. Instances of `Node` will therefore always have the attribute `Connected`, and instances of `Link` will have the attributes `Source` and `Dest`. As it may be seen, the virtual procedure `Connect` makes use of these attributes.

Notice that the classes `Node` in two different instance of `Graph` are different classes, as they have different context. A `Node` object from one `Graph` object cannot become part of another `Graph`.

In the subclass `DisplayableGraph`, the definitions of `Node` and `Link` have been extended. This is reflected in the extended definition of `Connect` which has an addi-

```

Set: class
  (# SetType: virtual class Record;
  A: [100] ~ SetType; Top :@ Integer;
    {A is an array of 100 references}
  Insert:...;
  Remove: ...;
  Display: proc
    (# do (for Inx:Top repeat
      A[Inx].Display
    for) #);
  #);
PersonSet: class Set
  (# SetType: extended class Person #)

```

Figure 8: General Set class

tional parameter `DL`. The execution of `Connect` implies the generation of a `Link` object (`new Link`). This `Link` object will be an instance of the extended `Link` class and the reference `L` will denote this instance. I.e. `L.DispLine` is a valid expression. The reference `DG.Root` is known to denote an instance of the extended `Node` class. I.e. an expression like `DG.Root.DispSymb` is valid.

The definition of `TravellingSalesman` is similar, but with different extensions of `Node`, `Link` and `Connect`.

### 3.2 Globally qualified virtual classes

There is a more general form for declaring virtual classes (and procedures) than that presented so far. This is explained for classes in this paper, but it also applies to virtual procedures.

Consider the general class `Set` in Figure 8. The virtual class attribute `SetType` defines the type (or class) of the objects that can be members of the set. `SetType` is a virtual class with the *qualification* `Record`. (See Figure 5.) This means that in subclasses of `Set` the virtual `SetType` may be redefined, but now as extensions of `Record` and not just as an immediate extension of the virtual definition itself.

In a `Set` object, the members may be instances of `Record` and its subclasses. The procedure `Display` displays the whole set. As the elements of the set are known to be at least `Record`-objects, they are known to have a `Display` attribute.

A reference like `A[inx]` is qualified by the virtual class `SetType`. The fact that `SetType` will be `Record` or further extensions of `Record` is used in accessing the `Display` attribute. Since instances of `Record` have a `Display` attribute, instances of `SetType` will also have this attribute.

In the class `PersonSet` the members are restricted to being instances of class `Person`. This is expressed by

```

Init: proc
  (# Open: virtual proc OpenRecord
  do...; someId[] -> Open; ...
  #);
InitW: proc Init
  (# Open: extended proc OpenWritableRecord #);
Bar:
  (#...
  do... ;
  Init(# Open: extended proc OpenWritableRecord #);
  ...
  #)

```

Figure 9: Virtual procedures as formal procedure parameter

extending the virtual class `SetType` to be a subclass of class `Record`. This implies that a reference like `A[Inx]` will correspondingly have its qualification extended to `Person`. This means that in the `PersonSet` subclass of `Set` the following expressions will be valid: `A[Inx].Name` and `A[Inx].Sex`.

The members of a `Set` are restricted to being instances of `Record` and its subclasses. In practice a `Set` class should be able to include any object. This may be obtained by replacing `Record` by `Object` in the definition of `SetType`. The `Display` procedure must then be removed, since `Object` does not (in BETA) have a `Display` attribute.

### 3.3 Virtual classes (and procedures) revisited

The examples of virtual procedures shown so far have all been defined using the syntactic form corresponding to locally defined classes. This is like Simula, C++ and Eiffel. As shown in Figure 9, a virtual procedure may also be defined using a form corresponding to globally defined virtual classes. The procedure `Init` has a virtual procedure attribute qualified by `OpenRecord`. In the subprocedure `InitW`, `Open` is extended to be `OpenWritableRecord`. This implies that execution of `Open` will be an execution of `OpenWritableRecord`. A virtual procedure used this way corresponds to a formal procedure parameter. The constraint on the actual parameter is that it must be a subprocedure of `OpenRecord`. It is, of course, awkward to have to declare a subprocedure like `InitW` in order to make a procedure call. In BETA it is possible to describe a single object directly. In `Bar`, a call of a singular procedure object is described directly. This corresponds to a prefixed block in Simula, which again is a generalization of an Algol inner block.

As can be seen, classes and procedures are completely

```

T: <proc/class> T0
  (# P: virtual <proc/class> A(# ... #);
  Q: virtual <proc/class> B;
  #);
TT: <proc/class> T
  (# P: <extended/fixed> <proc/class> (# ... #);
  Q: <extended/fixed> <proc/class> B1;
  {alternative:}
  Q: <extended/fixed> <proc/class> (# ... #)
  #)

```

Figure 10: Scheme for declaration of virtual attributes

analogous with respect to virtual attributes. In Figure 10, the different forms for declaring and extending virtual attributes are shown. The super-class/procedure `A` of `P` is optional. In this case super will be `Object`. The extension of `Q` may be specified as either a globally defined class or a locally defined class. The meaning of `fixed` will be explained later.

## 4 Virtual classes and generics/packages

The simple example above on a general set of objects or `Record` objects indicates the use of virtual classes as "type parameters" of the enclosing class.

In [12] an interesting comparison between genericity and inheritance is given. It is shown that, in general, inheritance cannot be simulated by genericity. On the other hand it is shown that genericity can be simulated by inheritance. However, it is concluded that the techniques for simulating so-called *unconstrained genericity* becomes rather heavy. For this reason unconstrained genericity has been included in Eiffel.

In this section it will be shown to what extent virtual classes can replace genericity. This will be done by giving definitions of a general class `Ring` [13] with attributes `Zero`, `One`, `Plus` and `Mult`. The class `Ring` is then used to define subclasses `Complex` and a general class `Vector` parameterized by `Ring`. The `Vector` class may in turn be used for defining a `ComplexVector` class.

The first version of class `Ring` is defined in a *pure object-oriented* style. By this is meant that operations like `a + b` are asymmetrical. Using Smalltalk terminology: the message `+ b` is sent to the object `a`. The second version of class `Ring` is defined in a *functional* style. Here the `+` is defined as a function of two arguments.

### 4.1 Pure object-oriented definition of class `Ring`

```

Ring: class
  (# ThisClass: virtual class Ring;
  Plus: virtual proc
    (# A: ~ThisClass enter A[] do INNER #);
  Mult: virtual proc
    (# A: ~ThisClass enter A[] do INNER #);
  Zero: virtual proc (# do INNER #);
  Unity: virtual proc (# do INNER #)
#);
Complex: class Ring
  (# ThisClass: extended class Complex;
  I,R: @ Real;
  Plus: extended proc
    (# do A.I->I.Plus; A.R->R.Plus #);
  Mult: extended proc (# ... #);
  Zero: extended proc (# do 0 -> I -> R #);
  Unity: extended proc (# ... #)
#);
Vector: class Ring
  (# ThisClass: extended class Vector;
  ElementType: virtual class Ring;
  R: [100] ~ ElementType;
  Plus: extended proc
    (#do (for i: 100 repeat
      A.R[i] -> R[i].Plus
    for)#);
  Mult: ... Zero: ... Unity: ...
#);
ComplexVector: class Vector
  (# ThisClass: extended class ComplexVector;
  ElementType: extended class Complex
  #)
C1,C2: @ Complex;
V1,V2: @ ComplexVector
...
C1.Unity; C2.Zero; C1[] -> C2.Plus;
V1.Unity; V2.Unity; V1[] -> V2.Plus;

```

Figure 11: Object oriented definition of class Ring

The general class `Ring` defines the virtual procedure attributes `Zero`, `Unity`, `Plus`, and `Mult`. In addition a virtual class attribute `ThisClass` (will be explained below) is included. The class `Complex` is one example of a subclass of `Ring`.

A more interesting subclass of `Ring` is the class `Vector`. This class includes a virtual class attribute `ElementType` qualified by `Ring`. `ElementType` defines the class of the elements of the vector, i.e. the elements of the vector have all the properties of a ring. Class `ComplexVector` is a subclass of `Vector` where the virtual class `ElementType` is extended to be class `Complex`. (In this example a vector consists of 100 elements. By using a virtual procedure, yielding an integer value, it is straightforward to parameterize the size of the vector.)

The virtual class `ThisClass` is used to ensure that the

argument of, say `Plus`, is always of the same type as the current class. In `Complex` it is therefore extended to be a `Complex`, and in `Vector` it is extended to `Vector`. If the reference `A` in the definition of `Plus` in class `Ring` was defined as `A: ~ Ring`, then in the extension of `Plus` in `Complex` the reference `A` might refer to any `Ring` object. An explicit check will be needed to ensure that `A` refers to a `Complex` object. In addition an operation like `V1[] -> C2.Plus` would be valid. Instead of explicitly defining a virtual class like `ThisClass`, it would be more convenient to have a predefined name for this. In [7] the name `# this Ring` is used. In the Smalltalk proposal in [3] the name `<self>` is used and in Eiffel this would correspond to like `current`.

## 4.2 Functional definition of class Ring

Even though a language is object-oriented there is no reason that it should not support the functional style of programming. Object-oriented languages are often criticized because even simple expressions like addition of two numbers `i, j` have to have the asymmetrical form `j -> i.plus`.

In languages with a package concept it is possible to define packages that collect the definition of a type and the operations on this type. A package is not a class, but rather a definition of a single object. A generic package on the other hand resembles a class. This is very limited however. In object-oriented terminology, a generic package can only be used for creating a single instance (a package). It is actually just templates that are elaborated at compile time. It is not possible to add properties like in subclasses.

It is possible to model a generic package by a class with virtual class and virtual procedure attributes representing the formal types and formal operations of the package.

In Figure 12 a functional definition of class `Ring` is given together with a subclass `ComplexRing` that defines the type `complex` and operations on complex objects. The virtual class attribute plays the role of the type. The operations on the type are defined in a functional (symmetrical) way on instances of class `Type`. Class `Type` is extended in subclasses of class `Ring`. To use a `ComplexRing` it is necessary to create an instance of it. In the example `CR` is such an instance. All complex references and operation calls are referred to as attributes of `CR`. Class `Ring` and `ComplexRing` may be compared to generic packages in Ada and `CR` may be compared to a generic instantiation. The next example further illustrates this.

In Figure 13 a vector is defined using a functional class. The important thing to notice is that the element type of a vector ring is not a virtual class. Instead

```

Ring: class
  (# Type: virtual class (# #);
  Plus: virtual proc
    (# I, Y, Z: ^Type
    enter(X[],Y[])
    do new Type[] -> Z[];
      INNER
    exit Z[]
    #);
  Mult: ... Zero: ... Unity: ...
#)
ComplexRing: class Ring
  (# Type: extended class (# I,R: @ Real #);
  Plus: extended proc
    (#do X.I + Y.I -> Z.I; X.R + Y.R -> Z.R #);
  Mult: ... Zero: ... Unity: ...
#);
CR: @ ComplexRing;
C1,C2,C3: ^CR.Type
...
CR.Unity -> C1[]; CR.Zero -> C2[];
(C1[],C2[]) -> CR.Plus -> C3[]

```

Figure 12: Functional definition of Ring

```

VectorRing: class Ring
  (# ElementRing: virtual class Ring;
  actualElementRing: ^ElementRing;
  Type: extended class
    (# V: [100] ^actualElementRing.Type #);
  Init: virtual proc
    (# aRing: ^ElementRing
    enter aRing[]
    do aRing[] -> actualElementRing[]
    #);
  Plus: extended proc
    (#
    do (for i: 100 repeat
      (X.V[i] [],Y.V[i] [])
      -> actualElementRing.Plus
      -> Z.V[i]
    for)
    #);
  Mult: ... Zero: ... Unity: ...
#);
ComplexVectorRing: class VectorRing
  (# ElementRing: extended class ComplexRing #);
CVR: @ ComplexVectorRing;
A,B,C: @ CVR.Type
...
CR[] -> CVR.Init

```

Figure 13: Functional definition of class Vector

it is described by the reference actualElementRing. The

```

ComplexRing: class Ring
  (# Type: extended class
    (# I,R:@ Real;
    Incr: proc (# do I+1->I; R+1->R #)
    #);
  ...
#);

```

Figure 14: Complex with local Incr operation

reason is that a `VectorRing` instance must be parameterized by a specific ring, i.e. an instance of `ElementRing`. Otherwise the elements of a vector may include say complex numbers from different complex rings. This does not seem right in this example. (However, it is possible to model this if desired.) In the example the reference `actualElementRing` is given a value when `init` is executed. (`CR` is the `ComplexRing` from Figure 12.) This is, however, not satisfactory, since `actualElementRing` should not change value after the initialization. It should denote the same `ComplexRing` during the life time of the `VectorRing`. This can be obtained by making `actualElementRing` a "call-by-const"<sup>4</sup> parameter of class `VectorRing`. It may then be bound when instantiating a `VectorRing` (or one of its subclasses) and not modified afterwards. Since such parameter mechanisms are well known it will not be further elaborated.

### 4.3 Class attributes versus type attributes

It could be argued that the definition of `ComplexRing` does not demonstrate the need for or usefulness of class attributes. The `Type` attribute could also be defined using a pure (record) type, as in Pascal. Such record objects could e.g. only be assignable and comparable, but not have procedure and class attributes as classes have.

However, by using a class attribute it is possible to combine the object-oriented style and the functional style. The `Type` class of `ComplexRing` may have a procedure attribute `Incr` that increments a complex number by 1, see Figure 14. It seems more natural to express such an operation in an object-oriented style than in a functional style.

With the addition of the `Incr` it is possible, in addition to functional expressions to specify evaluations like

```
...; C1.Incr; ...
```

<sup>4</sup> "Call-by-const" was used in the first version of Pascal.

```

VectorOfVector: class Vector
  (# ElementType: fixed class Vector
   (# ElementType: fixed class Elm #)
   Elm: virtual class Ring;
   ThisClass: extended class VectorOfVector
  #);
VectorOfVectorOfComplex: class VectorOfVector
  (# Elm: extended class Complex #)

```

Figure 15: Class VectorOfVector

```

Publication: class Record
  (# Author:...; Title:...; Date:...#);
Book: class Publication (# ... #);
Article: class Publication
  (# Keywords: ...#);

```

Figure 16: Subclass hierarchy of publications

#### 4.4 More on extending virtual classes

In this section, the `Vector` class of Figure 11 will be further elaborated. As shown in Figure 15 a class `VectorOfVector` parameterized by `Vector` is defined. A new virtual class `Elm` has been introduced to stand for the parameter of class `VectorOfVector`. The use of `fixed` instead of `extend` (cf. Figure 10) specifies that this is the final extension of `ElementType`, i.e. it is no longer virtual. In general it is useful to be able to specify that a virtual attribute can no longer be extended.

A note on syntax may seem appropriate here. The syntax for defining and extending virtuals in examples like the `Ring` may be found too heavy. Instead a usual positional notation for definition and extension of virtuals might be introduced.

## 5 Virtual superclasses

It is often desirable to add a set of attributes to all classes in a subclass hierarchy. Consider the hierarchy in Figure 16. Suppose that we want to implement a library of publications. For this purpose we want to add an attribute `ArcNo` to all publications. This may be obtained by adding this attribute to class `Publication`. Another alternative is to create new subclasses of `Publication`, `Book` and `Article`. Often none of these alternatives are attractive. It may not be feasible to modify an existing class. Likewise, creating several new subclasses is clumsy.

By using a virtual class as a superclass it is possible to describe an extension of all `Publication` classes. Consider Figure 17. The class `PubGen` acts as a gen-

```

PubGen: class
  (# GenType: virtual class Publication;
   ArcPub: class GenType
     (# ArcNo: @integer #);
   New: proc
     (# RN: @integer; R: ^ ArcPub
      enter RN
      do new ArcPub[] -> R[];
        RN -> R.ArcNo
      exit R[]
     #);
  #);
BookGen: class PubGen
  (# GenType: extended class Book #);
ArticleGen: class PubGen
  (# GenType: extended class Article #);
BG: ^ BookGen; B: ^ Book; S: ^ Set
...
111 -> BG.New -> B[]; '...' -> B.Author;
... B[] -> S.Insert

```

Figure 17: Example of virtual superclass

erator for instances of `Publication` and its subclasses. The procedure `New` generates these instances. The actual instances generated are of class `ArcPub` which is a subclass of `GenType`. `GenType` is a virtual class qualified by `Publication`.

The subclass `BookGen` extends the definition of `GenType` to `Book`. The class `ArcPub` of an object denoted by `BG` will then have `Book` as its superclass. This means that the instances of `ArcPub` created by `BG.New` will have all attributes of a `Book` in addition to the attribute `ArcNo`.

Objects created by a `PubGen` object can be used as ordinary `Publication` objects without knowledge of `ArcNo`. In the context of a `PubGen` object, they may be used as `ArcPub` objects, i.e. the attribute `ArcNo` may be accessed.

## 6 Conclusion

The notion of virtual class has been introduced as a generalization of the corresponding notion for procedures. In addition the qualification of a virtual and the mechanism for extension of a virtual have been introduced. For both kinds of virtual attributes two different syntactic forms have been introduced: locally and globally qualified virtual classes/procedures. The qualification and extension of a virtual ensures that the attributes of a virtual are maintained in any subclasses where the virtual is extended.

The global form of specifying a virtual class makes it possible to bind a virtual class to a non-local class. This makes it possible to use virtual classes as formal type

parameters.

Classes and procedures are completely analogous with respect to declaring virtual attributes, recall Figure 10. In BETA class and procedure are unified into one abstraction mechanism: *the pattern*. In all examples the keywords `class` and `proc` may be dropped. By replacing `virtual`, `extended` and `fixed` by `<`, `:<`, and `:` respectively the results are BETA programs. The common template for patterns has the form shown in Figure 3 without the keyword `proc`. From this it follows, that classes in BETA also have an action-part (`enter In do Imp exit Out`) like procedures. For a further discussion of this the reader is referred to [8].

As it may be seen, the notion of virtual pattern (class and procedure) captures the concept of virtual procedure as known from Simula, C++ and Eiffel. In addition it extends the possibilities for simulating genericity by means of subclassing. Also the global form of virtual patterns may be used to simulate higher order procedures and classes. I.e. procedures and classes parameterized by procedures and classes. Finally the notion of virtual class, in addition, gives a number of new possibilities.

The usefulness of virtual procedures are well known. Class attributes and especially virtual class attributes are less well know. These concepts have been used for several years by BETA programmers and they have clearly demonstrated their usefulness in practice. For other examples see [11] and [15].

**Acknowledgement.** The notion of virtual class was developed as part of the BETA project, in which Bent Bruun Kristensen, Kristen Nygaard and the authors were involved. Part of this work has been supported by the Danish Natural Science Research Council, FTU Grant No. 5.17.5.1.25.

## References

- [1] Ada Reference Manual: Proposed Standard Document. *United States Department of Defense, July 1980.*
- [2] P. America: Inheritance and Subtyping in a Parallel Object-Oriented Language. *ECOOP'87, European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 276, Springer Verlag, 1987.*
- [3] A.H. Borning, D.H. Ingalls: A Type Declaration and Inference System for Smalltalk. *University of Washington, August 1981.*
- [4] O.J. Dahl, B. Myrhaug, K. Nygaard: SIMULA 67 Common Base. *Norwegian Computing Center, Oslo, 1968.*
- [5] H.P. Dahle, M. Løfgren, B. Magnusson, O.L. Madsen: The Mjølner Project. *Software Tools 1987, Wembley, June 1987.*
- [6] A. Goldberg, D. Robson: Smalltalk-80: The Language and its Implementation. *Addison Wesley, 1984.*
- [7] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Abstraction Mechanisms in the BETA Programming Language. *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages, January 24-26 1983, Austin, Texas.*
- [8] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language. *In: B.D. Shriver, P. Wegner (ed.), Research Directions in Object Oriented Programming, MIT Press, 1987.*
- [9] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Classification of Actions or Inheritance also for Methods. *ECOOP'87, European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 276, Springer Verlag, 1987.*
- [10] O.L. Madsen: Block Structure and Object Oriented Languages. *In: B.D. Shriver, P. Wegner (ed.): Research Directions in Object Oriented Programming, MIT Press, 1987.*
- [11] O.L. Madsen, C. Nørgaard: An Object-Oriented Metaprogramming System. *Hawaii International Conference on System Sciences - 21, January 5-8, 1988.*
- [12] B. Meyer: Genericity versus Inheritance. *Languages and Applications, Sigplan Notices, September 1986.*
- [13] B. Meyer: Object-oriented Software Construction. *Prentice Hall, 1988.*
- [14] P. Naur (ed.): Revised Report on The Algorithmic Language ALGOL 60. *Regnecentralen. Copenhagen, 1962.*
- [15] C. Nørgaard, E. Sandvad: Reusability and Tailorability in the Mjølner BETA System. *Computer Science Department, Aarhus University, Draft March 1989.*
- [16] B. Stroustrup: The C++ Programming Language. *Addison-Wesley, 1986.*
- [17] B. Stroustrup: Possible Directions for C++. *Proc. USENIX C++ Workshop Nov 1987.*
- [18] P. Wegner, S. Zdonik: Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *ECOOP'88, European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, Vol. 322, Springer Verlag, 1988.*