Part Objects and their Location

Ole Lehrmann Madsen[a] and Birger Møller-Pedersen[b]

[a] Computer Science Dept., Aarhus University, Ny Munkegade, DK-8000 Aarhus C, Denmark

[b] Norwegian Computing Center, P.O. Box 114, Blindern, N-0314 Oslo 3, Norway

Abstract

The notion of location of part objects is introduced, yielding a reference to the containing object. Combined with locally defined objects and classes (block structure), singularly defined part objects, and references to part objects, it is a powerful language mechanism for defining objects with differentaspects or roles. The use of part objects for inheritance of codeis also explored.

1. Introduction

When modeling real world phenomena and concepts, classification and composition are fundamental methods of organization for apprehending the real world. Classification is the means by which we form and distinguish between different classes of phenomena and concepts. Composition is the means by which we understand phenomena and concepts as a composition of other phenomena and concepts.

A language to be used for modeling should have direct support for classification and composition. Object-oriented languages have support for classification by means of the class/subclass mechanism which may be used to describe instance-of and generalization/specialization relationships. Some languages only support tree structured classifications whereas others support non-tree structured hierarchies by means of so-called multiple inheritance.

There are several forms of composition that are useful in organizing composite objects, including whole/part relationships, reference composition and localization. In most object-oriented languages there is little direct support for composition. This is usually supported indirectly through instance variables as in e.g. Smalltalk. Instance variables (and thereby composition) are often considered implementation details and are not part of the public interface of an object. Alternatively, composition is often simulated using multiple inheritance. A consequence of this is that most object- oriented languages have good support for classification, but poor support for composition. This most clearly appears to be a problem when using a language for modeling. In [Booch90] and [Coad91], however, composition (often called aggregation or restricted to whole/part composition) is just as fundamental as classification.

In the Scandinavian tradition for object-orientation, modeling and design have been just as important as implementation when designing languages. BETA is designed to be used for modeling and design as well as implementation.

The purpose of this paper is to introduce a new language mechanism associated with part objects - the location of part objects. The mechanism of part- objects is not a new idea. Within the fields of semantic modeling and databases it has been recognized as a useful mechanism in order to model the is-part-of relation, see e.g. [Kim89]. Part-objects have also been proposed in object-oriented languages, e.g. [Cook87], [Sakkinen89].

The location of a part object is a reference to the object containing the part object. The paper will demonstrate that this mechanism extends the expressiveness of part objects.

Part objects are normally instances of globally defined classes. In this paper it is illustrated that locally defined part objects and objects of locally defined classes are natural extensions of the part object language concept.

In addition to the obvious purpose of modeling that wholes consist of parts, part objects may also be used to model that the containing object are characterized by various aspects, where these aspects are defined by other classes. The location mechanism and locally defined objects enhance this possibility considerably.

As proposed in e.g. [Raj89] code reuse does not have to be obtained solely by inheritance, but may also be obtained by part objects.

Multiple inheritance is often used to combine unrelated classes for the purpose of reuse of code. This often leads to complicated inheritance relationships. This (mis-)use of multiple inheritance may often be avoided and expressed in a cleaner way by using part objects as presented in this paper.

2. Basic Language Mechanisms

The language notation used in this paper is BETA [Beta93]. The ideas may, however, be applied to other object-oriented languages. The paper uses only a few of the constructs of the language, and only these are introduced. Readers familiar with BETA may skip this section. Objects, patterns, and subpatterns BETA program executions consists of objects. Patterns are used to represent categories of objects with the same properties. The example below illustrates most of the language mechanisms used in the paper; it will be used throughout the paper. It gives a definition of a pattern Address:

```
Address:
 (# Street:@ Text;
    StreetNo:@ Integer
    Town:@ Text;
    theCountry: ^ Country;
    whichCountry: (# do theCountry.Display #);
    printLabel:< (# do inner;
                       {print Street, StreetNo, Town};
                 #)
 #)
```

Address objects have the attributes Street, StreetNo, Town, theCountry, whichCountry, and printLabel.

2.1. Object attributes

The attributes Street, StreetNo, Town and theCountry are *object attributes*. The attribute theCountry is a *reference* to a *separate* object since this application keep information on countries in separate objects. Other object references may need to refer to the same country object. References to separate objects may be assigned new objects dynamically. In this sense they are similar to instance variables in Smalltalk.

Street, StreetNo and Town are *part* objects of an Address object in the sense that they denote the same objects during the lifetime of the containing Address object. Any instance of Address will have two Text objects and an Integer object as fixed parts. These objects are generated as part of the generation of an Address object.

2.2. Pattern attributes

The attributes whichCountry and printLabel are *pattern attributes* that are used as procedures. While whichCountry is a non-virtual procedure pattern, printLabel is a *virtual* procedure pattern. Note that BETA has only one single pattern mechanism for defining categories of objects. The notion of pattern is a generalization of mechanisms like classes, procedures, functions, and types. The definiton and the use of a pattern determines whether it is a class or procedure pattern. In the example above, the pattern Address is a class pattern, while e.g. printLabel is a procedure pattern. In general any object may have attributes (references to objects, and patterns) and an action sequence (do...).

Virtual patterns In BETA a subpattern is supposed to be behavioural and structural compatible with its superpattern, according to the definitions in [Wegner88]. The language does not support behavioral compatibility, but it has more support for structural compatibility than most other languages. A virtual pattern attribute like printLabel of Address may not be completely redefined, but only further *extended* in subclass patterns of Address. Extending a virtual pattern implies to define it as a subpattern of the definition given as part of the virtual pattern specification. In subpatterns attributes may be added, and execution of the special imperative inner implies the execution of the actions of the subpattern. We shall see later how this is used in an extension of printLabel. In general any pattern may be specified to be virtual, thus also class patterns. For a more detailed description of virtual classes see [Madsen89].

3. Introducing Part Objects

The patterns of the part objects (Text, Integer) in the example above are not so interesting for the discussion in this paper. They correspond more or less to what in some languages would be standard, predefined simple (type) classes.

The example has, however, indicated that the combination of predefined simple types (e.g. Integer) to be regarded as class patterns and the wish to be able to specify attributes of predefined simple types as part objects (and not only as references to e.g. separate Integer objects) may be resolved in two ways: either treat predefined simple (type)) classes in a special way, or provide part objects in general for objects of any class.

In BETA the last alternative has been chosen. It is possible to specify part objects of any pattern, that is also of user- defined patterns. Consider e.g. a pattern Person defined as below. Here the pattern Address is used to define a part object of Person.

```
Person:
(# Name:@ Text;
   Addr:@ Address;
   ...
#)
```

For the definition of Person it is here assumed important to model the name and address attributes as part objects. Other applications could have the Name as a part object and Addr as a reference to a separate Address object.

A much more interesting example on part object is the following from [Booch90]. In order to model apartments consisting of a kitchen, a bathroom, a bedroom, and a family room it should be possible to model that the various room objects are permanent part objects of the apartment object. In BETA this is directly modelled by:

```
Apartment:
  (# theKitchen   :@ Kitchen;
     theBathroom  :@ Bathroom;
     theBedroom   :@ Bedroom;
     theFamilyRoom:@ FamilyRoom;

     theOwner  :^ Person;
     theAddress:@ Address;
     ...
  #)
```

where Kitchen, Bathroom, Bedroom and FamilyRoom are names on classes. As long as the Apartment object exist it will have the same four part-objects.

Note that the owner of an apartment is represented by a *reference to a separate* Person object (so that the owner may change), while the address of an apartment is represented by a part object, see figure 1.

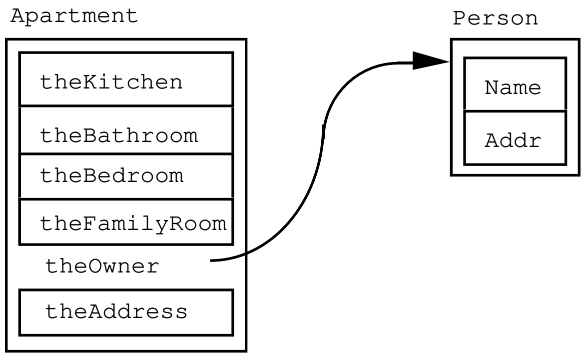


Figure 1. Separate and Part Objects

Representing parts by references to separate objects is of course possible in languages that do not support part objects, but it will not ensure that the references are not later changed to denote other objects. In languages where object references are not typed, the theKitchen reference may even change to denote a Bathroom object, while in typed languages only changing an old kitchen to a new kitchen may be specified. Note that BETA provides both part and separate objects, and that object references are typed. In the above example the theOwner is a reference to a separate object, and it may only denote objects of class Person or of subclasses to this.

4. Singular Objects and locally defined Objects and Classes

Even though the notion of part-objects is not a new idea, then the combination of this and the notion of *locally defined classes and objects* gives a more powerful notion of part objects. In addition to express the is-part relation, it may then also express that the description of the classes of the part-objects or the descriptions of the part object are only meaningful in the context of the containing object.

If an object is described directly without referring to a pattern, the object is called *singular*. Singular objects resembles objects in Self [Ungar87] in the sense that they are not described as instances of classes. Both separate objects and part objects may be singular. Applied to part objects it introduces also the notion of *locally defined part-objects*.

As the Apartment class is described above, the kitchen is modeled by a general Kitchen class. This is a general class also used in other classes than Apartment, e.g House. Suppose that kitchens of apartments have special properties in additions to those of kitchens in general, and that these properties are only meaningfully defined in the context of Apartments. This will in BETA be represented by theKitchen being a part object that is singularly defined locally to class Apartment:

```
Apartment:
(#   theKitchen:@ Kitchen
      (# {additional attributes specific for a kitchen of an apartment} #);
    ...
 #)
```

The descriptor of theKitchen describes a singular object and not a category of objects. The superclass pattern mechanism applies to object-descriptors in general, and not only to pattern definitions. The object-descriptor of the theKitchen object has Kitchen as superclass pattern.

Classes may also be described *locally* to other classes. If bathrooms of apartments also have properties in addition to those of bathrooms in general, and if the apartment had two bathrooms, then this calls for a locally defined subclass of Bathroom and two objects of this class:

```
Apartment:
 (# ...
    ApartmentBathroom:  Bathroom
      (# {additional attributes} #);

    Bathroom1:@ ApartmentBathroom;
    Bathroom2:@ ApartmentBathroom;
    ...
 #)
```

If the two bathrooms are not quite the like, then they may also be defined as two singular objects, both with the common properties of ApartmentBathroom but also with specific attributes in addition:

```
...
Bathroom1:@ ApartmentBathroom(# ... #);
Bathroom2:@ ApartmentBathroom(# ... #);
...
```

Note that in some cases the class ApartmentBathroom could as well have been defined outside class Apartment. In some cases it will, however, be important that ApartmentBathroom is defined locally to Apartment. One reason can be that it is desirable to express that ApartmentBathroom objects are only meaningful within Apartment objects (and HouseBathroom within House). Another reason can be that the additional attributes of ApartmentBathroom are defined directly in terms of attributes of Apartment: within the description of ApartmentBathroom the attributes of Apartment are directly visible. The same holds for the additional part of the description of theKitchen.

For further examples on using locally defined patterns see [Madsen87].

5. References to Part Objects

From a modeling point of view it seems obvious that supporting part objects as demonstrated above also implies that the part objects are visible to other objects. Given a reference

```
  myApartment: ^Apartment
```

then the attribute theKitchen is accessible by:

```
  myApartment.theKitchen
```

This is used in order to get access the attributes of theKitchen, e.g.:

```
 pink -> myApartment.theKitchen.Paint
```

The issue of encapsulation is in most object-oriented languages closely linked to specific language constructs, e.g. only method/function attributes of objects are accessible, and not object attributes (instance variables). Instance variable are encapsulated in these languages, because they are regarded as implementation specific. BETA has a separate mechanism for distinguishing between interface and implementation. This mechanism is not covered in this paper, but it allows both object attributes and pattern attributes to be specified visible or not.

The difference between attributes like theKitchen and e.g. instance variables is that the theKitchen object attribute is an important part of the modeling of a real world apartment, while instance variables are only intended as auxiliary variables for representation purposes.

The part objects are not only visible for remote access to their attributes. The object reference anAddress (qualified by Address) below may denote Address objects that are part objects, in this case of an Apartment object. Given the reference:

```
   anAddress: ^ Address
```

the following *reference assignment* to anAddress are legal:

```
   myApartment.theAddress[]->anAddress[]
```

The object reference anAddress does then not refer to the whole Apartment object. It refers to the Address part of the containing object, see figure 2.

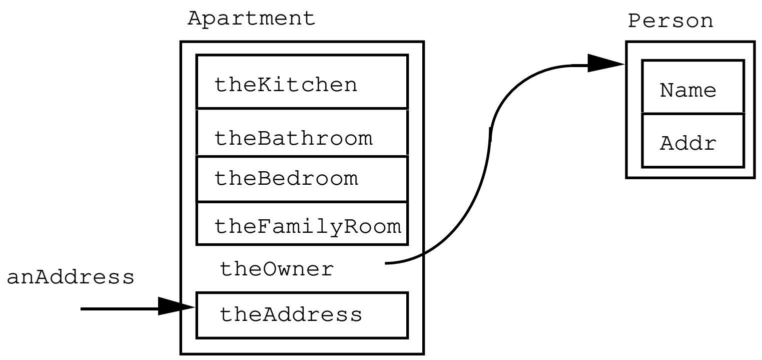


Figure 2. Reference to Part Objects

6. Location of Part Objects

All BETA objects have a predefined reference attribute loc denoting a possibly containing object. In the case of a separate, whole object, loc has the value none. In the example above, the loc attribute of the theKitchen part object of an Apartment object denotes the Apartment object. The loc attribute of the theAddress part object of an Apartment object denotes the Apartment object, while the loc attribute of the Addr part object of a Person object denotes the Person object, see figure 3.

Since the object reference anAddress above denotes a part of a whole Apartment object (after the reference assignment above) we may obtain a reference to the whole Apartment object (having the Address denoted anAddress as a part object) by means of the object reference expression

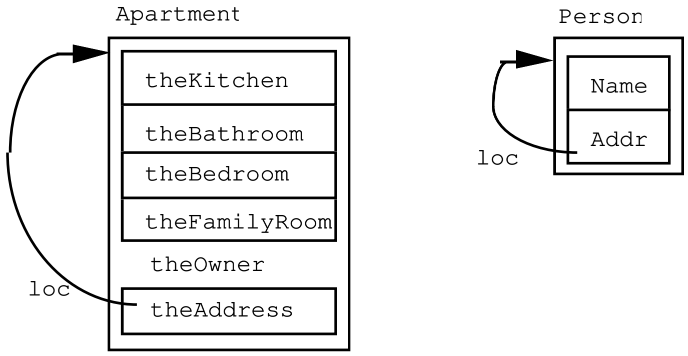
```
anAddress.loc
```

and we may assign this object reference to the object reference myApartment by:

```
anAddress.loc[]->myApartment[]
```

anAddress.loc denotes the Apartment object of which anAddress is a part. Note that the above assignment is only legal if anAddress denotes a part object of an Apartment object, and not if it is part of an Person object. This can in general only be detected at run-time.

The reference loc to the containing object is *not* intended to be used in order to access the attributes of the containing object. Different objects of a given class (e.g. Address) may be parts of objects of different classes, so in general the loc reference will not be qualified by a specific class. Not even loc of objects of locally defined classes will necessarily be qualified by the (textually) enclosing class. The loc for a specific part object is, however, well-defined.



Firgure 3. Location of Part Objects

What may then the notion of location of part objects be used for ?

The situation addressed by location of part objects is the situation where objects of different classes must be handled similarly, where this similarity is represented by a class, and where it is not adequate to have this similarity class as a superclass for all these classes - the reason being e.g. that they are parts of other class hierarchies.

Assume as a simple example that both Person and Company objects have to be treated similarly according to their addresses.

The properties associated with an address is assumed to be represented by the class Address (as above). The Person class is defined by:

```
Person: (#  Addr:@ Address; ... #);
```

and Company by

```
Company: (#  Addr:@ Address; ... #);
```

The similar handling of the objects used here will be the simple one to have them in an Address list. Assume that the concept of a list is defined by the abstract class pattern:

```
list: (#  elmType:< ; ... #)
```

where the type of the elements is represented by the virtual class pattern elmType. The type of a specific list is specified by extending (::) the virtual elmType to the appropriate class:

```
AddressList:@ list(# elmType:: Address #);
```

Person objects and Company objects may now be parts of an Address list. The references used in the implementation of an Address list are qualified by Address and they reference the Address part object of the objects, and not the whole Person or Company objects.

Given a reference to a Person object it is e.g. inserted in an Address list by a reference assignment of the form:

```
  aPerson.Addr[] -> AddressList.Insert
```

But what about the other way ? How are Person objects in an address list extracted and treated as Person objects? The address list will only know of Address objects being linked together and will not know whether these are parts of Person or Company objects.

The solution is the loc attribute of the Address part objects, as illustrated in the following.

Assume that the AList may have a getNextNational procedure pattern attribute, that delivers a reference to the next national (according to the address) element in the set.

Assume that there is also a list of persons:

```
PersonList:@ list(# elmType:: Person #);
```

This list will be implemented with references qualified by Person and they will denote whole Person objects. The Insert procedure pattern attribute of Plist objects will require a Person qualified object reference as input parameter.

Obtaining e.g. the next national person object from the address list and inserting it into the Plist is simply obtained by using the location property of the Address part object: nextAddr: ^Address;

```
do
  ...
  Alist.getNextNational->nextAddr;
  ...
  nextAddr.loc[]-> PersonList.Insert;
  ...
```

The expression nextAddr.loc denotes the whole Person object of which the Address object denoted by nextAddr is a part object.

6.1. Run-time type check

The assignment "nextAddr.loc[]-> Plist.Insert" will involve a run-time type check to ensure that the Address object denoted by nextAddr in fact is a part object. Assigning the containing object to Plist.Insert will generate a run-time type check to ensure that it is a Person object (and e.g. not a Company object). Most likely the last type check will be specified explicitly, e.g. by

```
(if nextAddr.loc[] is-a Person then
    nextAddr.loc[]-> PersonList.Insert
if)
```

In general the use of loc in assignments of the form:

```
A: ^ Address; P: ^ Person
...
A.loc[] -> P[]
```

implies a run-time check since it cannot be known if A in fact refers to an Address object which is part of a Person object. This form of assignment is analogous to assignments of the form:

```
Vehicle: (# ... #);
Bus: Vehicle (# ... #);

V: ^ Vehicle; B: ^ Bus
...
V[] -> B[]
```

which also requires a run-time check since it cannot be known at compile-time whether or not V in fact refers to a Bus object, see [betat.

In both cases it is the responsibility of the programmer to guarantee/prove the type correctness. A traditional assignment from a less qualified reference to a more qualified reference, like "V[]->B[]", is only made if the programmer is sure that V in fact refers to a Bus object. Similarly an assignment using loc, like "A.loc[]->P[]", is only made if the programmer is sure that the object is in fact a part of a Person object. If theses conditions cannot be verified, it is possible to test the qualification of V and A.loc, respectively, before making the assignment.

It could be argued that it would be simpler to define the attributes of Address as direct attributes in pattern Person. Then the reference to the next element in the list would be qualified by Person, and it would not be necessary with the loc attribute of any part objects. The reason in this case is that several class of objects - and not only Person objects - may be characterized by an address.

It could also be argued that a better solution would be to define *class Person as a subclass pattern of Address*. This would be a solution that would work technically, but it would be rather strange to define the class of persons as a subclass of class address.

Note the fact that even though both Person and Company objects have an Address attribute, this does not necessarily qualify them to be subpatterns of a common superpattern. It has not been included here, but Person could be a subpattern of one pattern, and Company could be a subpattern of quite another pattern.

Person and Company are not subpatterns of a common superpattern, but they may still be treated similarly because of their part objects. The Alist may both have Person objects and Company objects as members, indirectly through the Address part objects.

As mentioned earlier, it may appear that the visibility of part objects breaks the rules of encapsulation. Part objects, like Address, are, however, considered part of the model (public interface) of Person and Company and should thus be visible. Choosing the alternative with Address being a superclass of Person and Company, the attributes of Address would also be visible as attributes of Person and Company objects.

7. Part Objects representing Aspects of Objects

The simple example above has just indicated the possibilities of the notion of part objects beyond the simple modeling of real world parts. While theKitchen attribute represented a real world part of a larger real world object, then the Address part object of e.g. Person objects rather represent an (though simple) *aspect* of a Person object.

Representing different aspects of objects are sometimes the rationale for introducing multiple superclasses. It could be argued that representing different aspects of an object by means of multiple superclasses is more powerful than representing them by part objects. The subclass will have direct access to the attributes of the superclasses, and virtuals of the superclasses may be redefined in the subclass in order to express adaptation to special needs. The following will show, that this is also supported by part objects.

7.1. Access to Attributes - simple Renaming of Attributes

The use of part objects for modeling aspects of objects will imply indirect access to the attributes of the part objects. If it is desirable to regard the attributes of part-objects as attributes of the containing object, then this is obtained by a simple, general renaming scheme.

In BETA an attribute may in general be renamed using the following construct:

```
<alias> : <attributeDenotation>
```

This facility can be used to ``lift'' the attributes of part objects to become attributes of the containing object.

As an example consider the Person class, with an Address part object specified. If Person objects should have an attribute theTown being the Town attribute of the

Address part object, then this is specified by:

```
Person:
 (#  Name:@ Text;
     Addr:@ Address;
     ...
     theTown: Addr.Town;
     ...
 #)
```

7.2. Redefinition of Virtuals

By use of part objects, a Person object will get the attributes specified in Address (although indirectly contained in a part-object), and by renaming it may access them just as if Address had been a superclass pattern, but what about the redefinition of virtual pattern attributes in Address? This possibility is closely associated with subclassing and seems to be lost when representing aspects by means of part objects. The following will show that this is *not* the case.

The very first description of Address had a virtual procedure pattern attribute printLabel. If this virtual pattern should be redefined when using Address to represent an aspect of Person, should Person then not be a subpattern of Address ?

This is not the case. Recall that in the Apartment example the theKitchen part object were described as an object with the attributes of Kitchen and some attributes in addition. This was obtained by an object descriptor with Kitchen as superpattern. Similarly the local class ApartmentBathroom were defined as a subclass pattern of Bathroom.

The same may be done for the Addr part object of Person. Defined as a singular part object, with Address as super pattern, it is possible to redefine the virtual procedure pattern printLabel of the superpattern.

This redefinition may even manipulate attributes defined in the containing pattern. This is exemplified by the use of the attribute Name below:

```
Address:
(# Street:@ Text;
   StreetNo:@ Integer
   Town:@ Text;
```

```
    printLabel:< (# do inner;
                       {print Street, StreetNo, Town};
     #)
 #);
Person:
 (# Name:@ Text;
    Addr:@ Address
      (# printLabel:: (# do ...;{print Name}...#);
      #)
#)
```

The Addr part object is now a singular object, defined with the pattern Address as superpattern. The printLabel of Address has been extended to print also the Name attribute. Recall that redefinition of a virtual in BETA implies extension: the redefinition is a *subpattern* of the virtual definition. Execution of the special imperative inner implies the execution of the actions of the subpattern. The action print Name is therefore executed in the place of inner and therefore printed before Street, StreetNo, Town.

If Address had been a superclass pattern of Person, then the redefinition of the virtual printLabel would have access to attributes of Person. Note that even though Address is not a superclass pattern of Person, then the redefinition of the virtual printLabel still has access to attributes of Person: the Name is visible in the extension because the redefinition is defined in the scope of Person. This is due to the notion of locally defined objects: the redefinition of printLabel is specified in a locally defined object Addr.

7.3. Aspects belonging to different Classification Hierarchies

The notion of different aspects of objects is closely associated with subclassing used to model classification hierarchies, and often there are more than one classification hierarchy for a class of objects. Consider two possible classification hierarchies for Persons: One classifies them according to what kind of (Sportsman they are, (e.g. TennisPlayer, GolfPlayer, ...), another classifies them into how they are Students: (e.g. FullTimeStudent, PartTimeStudent). Each of these are represented by subclasses (of e.g Sportsman and Student, respectively), see figure 4. This example is an extension of the example found in [Carre90], but in the following it is just indicated how part objects may be used instead of multiple superclasses. The discussion in [Carre90] is much more detailed.

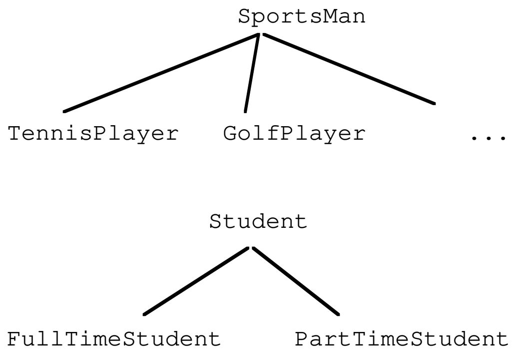


Figure 4. Independent classification Hierarchies

The class of persons being both a sportsman and being a student, SportyStudent, may obviously be defined as a class with Sportsman and Student as superclasses. This will, however, exclude this class from being specialized to e.g. a person that is a specific kind of sportsman and a specific kind of student. The superclasses are fixed and may not be redefined in subclasses of SportyStudent.

By the use of part objects of locally defined, virtual classes, this flexibility is achieved:

```
SportyStudent:
 (# TypeOfSportsman:<  Sportsman;
    TypeOfStudent:<  Student;

    theSportsman:@ TypeOfSportsman;
    theStudent:@   TypeOfStudent;
    ...
 #)
```

The constraint of the virtual classes implies that TypeOfSportsman may only be extended to one of the subclasses of Sportsman , while TypeOfStudent may only be extended to one of the subclasses of Student.

A sporty student being a tennis player and part time student will then be defined by the following subclass:

```
TennisPlayingPartTimeStudent:
 SportyStudent
 (# TypeOfSportsman:: TennisPlayer;
    TypeOfStudent:: PartTimeStudent;
 #)
```

7.4. More than one Aspect of the same Class

The part object approach makes it straight forward to define persons that are students at two universities, while this is normally avoided using multiple superclasses. Renaming of attributes may here be used to e.g. introduce special names on some of the attributes of the parts:

```
 DoubleStudent:  SportyStudent
  (# ExtraTypeOfStudent: <  Student;
     theExtraStudent:@ ExtraTypeOfStudent;
     ...
     mainUniversity: theStudent.University
  #)
```

Another example on this is the use of multiple inheritance to express that windows may have both a title and a border: Two subclasses WindowWithTitle and WindowWithBorder (of Window) are defined, and a class WindowWithTitleAndBorder is defined with WindowWithTitle and WindowWithBorder as superclasses. As pointed out in [kasper there is no reason that window may not have e.g. two titles

```
 DoubledTitledWindow:  Window
  (# theBorder: @ Border;
     title1:@ Title;
     title2:@ Title
  ...
 #)
```

The notion of e.g. sub-windows also calls for being defined as several part objects.

8. Inheritance of Code

Inheritance (or subclassing) have turned out to be useful for many purposes. This has resulted in a number of papers discussing different views on inheritance [sakkinen, [pun, [wz, and [snyder. These discussions may briefly be summarized as follows:

Subclassing may be used to model classification hierarchies in the application domain. In this situation there are intuitive guidelines on when and how to use subclassing. A subclass hierarchy should be meaningful in the application domain.

There are of course also objects and classes that are used purely for implementation purposes and which may not represent phenomena and concepts from the application domain. In this situation it may seem more difficult to decide whether or not to use subclassing. Some authors distinguish between *inheritance of specification* and *inheritance of code*. When inheriting specification, there is a subtype relation between a subclass and its superclass. If a subclass is viewed as a subtype of its superclass, then the subclass should be applicable whenever the superclass is applicable. When inheriting code, a subtype relation does not have to exist. It may be difficult (and it has turned out to be difficult) to design one language mechanism that supports inheritance of specification and inheritance of code equally well. See [hp for a discussion of this.

In BETA [beta subclassing is intended for classification hierarchies and subtyping. Using subclassing for representing classification hierarchies or ``type hierarchies'' correspond to using subclassing as specialization.

In this section we shall describe how part objects may be used to obtain inheritance. This form of inheritance is mainly useful for inheritance of code. In [sakkinen it has been proposed to use aggregation to simulate incidental inheritance. This section shows how this proposal is supported in BETA

In the examples above the part objects are used to model directly real parts or to model aspects. It is, however, also possible to construct objects with part objects with the only purpose of providing useful properties for the further definition of the objects in question.

Remote identifiers to part objects may be used to get access to desirable code represented by patterns. For pattern Person this means that the pattern attributes of Text and Address are available in the further definition of class Person. The attributes of the Text object are available through remote identifiers of the form

```
Name.getChar    Name.putChar
```

and the attributes of the Address object are available through remote identifiers of the form

```
 Addr.whichCountry   Addr.printLabel
```

In this special case these accessible patterns are procedure patterns that act upon the part objects, but in general the part objects may just contain a set of related patterns. These will then be accessible by remote identifiers.

This observation forms the basis for the use of part objects for code-sharing as an alternative to inheritance.

Assume that we want to define a pattern Deque that happens to be quite similar to the class Queue. In this case all the attributes of Queue may be reused. The standard way of defining Deque using inheritance by means of subclassing is:

```
 Deque:  Queue(# enterInFront: ...;
                 removeFromBack:
                 ...
               #)
```

Here Deque is defined as a subpattern of Queue and Deque inherits all attributes of Queue. If we try the same with the definition of Stack as a subpattern of Deque, then it would work only by excluding some of the operations of Deque. There will therefore not be a sub-type relation between Deque and Stack, so subclassing does not seem to be the right thing to do in a language where subclassing is supposed to be subtyping. Using the terminology of [sakkinen, it is quite incidental that Stack can inherit most of the description of Deque.

We may alternatively describe Stack using inheritance by means of part objects and renaming the desirable properties:

```
Stack: (# d:@ Deque;
          push: d.enterInFront;
          pop: d.removeInFront
       #)
```

In general in BETA the d attribute is remotely accessible. If S refers to an instance of Stack, the following remote attribute is available:

```
S.d.removeFromBack
```

Let us now assume that the description of Stack is identical to the description of Deque (for the non-excluded attributes) except for the attribute enterInFront. The standard way of defining Stack is to define a subpattern of Deque and redefine enterInFront as shown below:

```
Stack:  Deque(# enterInFront: (#...#);  #)
```

Instead part objects may be used as shown in the following example:

```
Stack: (# d: @ Deque;
          push:  (#...#);
          pop: d.removeInFront
        #)
```

Here the push is defined as a new attribute of Stack, while pop is just a renaming of d.removeInFront.

If push shall not be a complete redefinition, but a specialization of the enterInFront of Deque, then this may still be expressed, even though Stack is not a subpattern of Deque and enterInFront not a virtual pattern. The solution is to define the new push as a subpattern of d.interInFront:

```
Stack: (# d:@ Deque;
          push:
          d.enterInFront(#...#);
          pop: d.removeInFront
       #)
```

Often multiple inheritance is used for code sharing. The reuse-of-code-by-part-objects approach above may easily by generalized to also cover the need for multiple inheritance.

Suppose that we have a pattern A with attributes x, y and z, a pattern B with attributes s and z and that T may be described by inheriting from A and B. Using part objects only will require that the attributes of the part objects must be accessed by the full names:

```
T: (# a:@ A;
      b:@ B;
   #)
```

The attributes will then be accessed by

a.x a.y a.z b.s b.z;

Note that this form of multiple inheritance resembles the one which has been implemented for C++ [Stroustrup86].

In order to resolve name conflicts and in order to avoid the compound identifiers, renaming of attributes (as introduced above) may be used:

```
   T: (# a:@ A;
         b:@ B;

         x: a.x;
         y: a.y;
         az:a.z;
         s: b.s
         bz:b.z
      #)
```

Another consequence of renaming is that T objects now have the following attributes:

x y az s bz

In case of no name collisions, the following (shorter) solution may be used:

```
  T: (# :@ A;
        :@ B;
     #)
```

T is here specified to have two anonymous part objects. The implication of this is that the attributes of A and B become attributes of T without renaming.

In case there are only a few name conflicts, then anonymous singular part objects with renaming specified will be the way to do it:

```
  T: (# :@ A;
        :@ B(# bz: z #);
     #)
```

T objects will now have the following attributes:

```
    x    y    z    s    bz
```

where the z attribute is the one from A.

9. Conclusion

The notion of location of part objects has been introduced. Examples have been given combining part objects with locally defined objects and classes, especially for representing different aspects of objects.

It has been demonstrated that some of the features of inheritance of code may just as well be provided by part objects. The introduction of the notion of the location of a part object greatly improves the use of parts-objects.

The provision of real multiple specialization (that is multiple subtyping) is not covered by part objects. By specifying singular part objects that are specializations of some general classes, it is possible to add attributes and to redefine virtual procedures, so for each of the part objects a specialization may be obtained.

The location and renaming construct described in this paper has been tested in an experimental version of the Mjølner BETA System, but is not part part of the official release.

10. References

[Booch90] G. Booch: *Object Oriented Design with Applications.* Benjamin/Cunnings Pub. Co. Sept. 90.

[Cook87] E. Blake, S. Cook: *On including Part Hierarchies in Object-Oriented Languages, with an implementation in Smalltalk.* In Proceedings ECOOP'87.

[Canning89] P.S. Canning, W.R. Cook, W.L. Hill, W.G. Olthoff: *Interfaces for Strongly-Typed Object-Oriented Programming.* In OOPSLA'89, Sigplan Notices Vol. 24, No. 10, Oct. 1989

[Carre90] Carre, B., Geib, J-M.: *The Point of View Notion for Multiple Inheritance.* In OOPSLA'90, Sigplan Notices Vol. 25, No. 10, Oct. 1990

[Carroll91] Carroll, M.: *Using Multiple Inheritance to Implement Abstract Data Types.* The C++ Report, 3(4), April 1991.

[Coad91] P. Coad, E. Yordon: *Object Oriented Analysis.* Yuordon Press Computing Series 1991.

[Guimaraes91] Guimaraes, N.: *Building Generic User Interface Tools: an Experience with Multiple Inheritance.* In OOPSLA'91, Sigplan Notices Vol. 26, No. 11, Nov. 1991

[Kim89] Kim, W, et al.: *Composite Object Support in an Object-Oriented Database System* In OOPSLA'89, Sigplan Notices Vol. 24, No. 10, Oct. 1989

[Kristensen87] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the The BETA Programming Language.* Addison Wesley, 1993.

[Madsen87] 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

[Madsen90] O.L. Madsen, B. Magnusson, B. Møller-Pedersen: *Strong Typing of Object-Oriented Languages Revisited.* In OOPSLA'90, Sigplan Notices Vol. 25, No. 10, Oct.1990

[Madsen89] O.L. Madsen, B. Møller-Pedersen: *Virtual Classes --- A Powerful Mechanism in Object- Oriented Programming.* In OOPSLA'89, Sigplan Notices Vol. 24, No. 10, Oct. 1989

[Meyer88] B. Meyer: *Object-oriented Software Construction.* Prentice Hall, 1988.

[Pedersen89] C.H. Pedersen: *Extending Ordinary Inheritance Schemes to include Generalization.* In OOPSLA'89, Sigplan Notices Vol. 24, No. 10, Oct. 1989

[Pun89] W. Pun and R.L. Winder: *A Design Method for Object-Oriented Programming.* In Proceedings of the 1989 European Conference on Object-Oriented Programming, ECOOP'89, British Computer Society Workshop Series, Cambridge University Press 1989.

[Raj89] R.K.Raj, H.M.Levy: *A compositional Model for Software Reuse.* In In Proceedings of the 1989 European Conference on Object-Oriented

Programming, ECOOP'89, British Computer Society Workshop Series, Cambridge University Press 1989.

[Sakkinen89] M. Sakkinen: *Disciplined Inheritance.* In Proceedings of the 1989 European Conference on Object-Oriented Programming, ECOOP'89, British Computer Society Workshop Series, Cambridge University Press 1989.

[Snyder86] A. Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages.* In OOPSLA'86, Sigplan Notices Vol. 21, No. 11, Nov. 1986

[Stroustrup86] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley, 1986.

[Ungar87] D. Ungar, R. B. Smith.: *Self: The Power of Simplicity.* In OOPSLA'87, Sigplan Notices Vol. 22, No. 12, Dec. 1987

[Wegner88] P. Wegner, S. Zdonick: *Inheritance as an Incremental Modification Mechanism or What Like is and Isn´t Like.* In ECOOP'88, European Conference on Object-Oriented Programming, Lecture Notes In Computer Science, Vol. 322, Springer Verlag, 1988.

[Østerbye90] Østerbye, K.: *Parts, Wholes and Sub-Classes.* In Proceedings of the 1990 European Simulation Multiconference, edited by Bernd Schmidt. ISBN 0-911801-73-1.