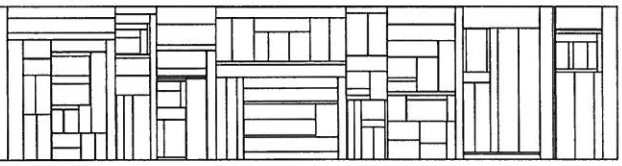
ISSN 0105-8517

What object-oriented programming may be – and what is does not have to be

Ole Lehrmann Madsen
Birger Møller-Pedersen

DAIMI PB – 273
February 1989

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 71 88 *Telex: 64767 aausci dk*



What object-oriented programming may be - and what it does not have to be

Ole Lehrmann Madsen
Dept. of Computer Science, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
email: olm@daimi.dk

Birger Møller-Pedersen
Norwegian Computing Center
P.O.Box 114 Blindern, N-0314 Oslo 3, Norway
ean: birger@vax.nr.uninett

Abstract

A conceptual framework for object-oriented programming is presented. The framework is independent of specific programming language constructs. It is illustrated how this framework is reflected in an object-oriented language and the language mechanisms are compared with the corresponding elements of other object-oriented languages. Main issues of object-oriented programming are considered on the basis of the framework presented here.

1. Introduction

Even though object-oriented programming has a long history, with roots going back to SIMULA [SIMULA 67] and with substantial contributions like Smalltalk [Smalltalk] and Flavors [Flavors], the field is still characterized by experiments, and there is no generally accepted definition of object-oriented programming.

Many properties are, however, associated with object-oriented programming, like: "Everything is an object with methods and all activity is expressed by message passing and method invocation", "Inheritance is the main structuring mechanism", "Object-oriented programming is inefficient, because so many (small) objects are generated and have to be removed by a garbage collector" and "Object-oriented programming is only for prototype programming, as late name binding and run-time checking of parameters to methods give slow and unreliable ("message not understood") systems".

Part of this work has been supported by NTNF, The Royal Norwegian Council for Scientific and Industrial Research, grant no. ED 0223.16641 (the Scala project) and by the Danish Natural Science Research Council, FTU Grant No. 5.17.5.1.25. Also appearing in Proceeding of European Conference on Object Oriented Programming, August 1988, Oslo, Norway.

There are as many definitions of object-oriented programming as there are papers and books on the topic. This paper is no exception. It contributes with yet another definition. According to this definition there is more to object-oriented programming than message passing and inheritance, just as there is more to structured programming than avoiding gotos.

While other programming perspectives are based on some mathematical theory or model, object-oriented programming is often defined by specific programming language constructs. Object-oriented programming is lacking a profound theoretical understanding. The purpose of this paper is to go beyond language mechanisms and contribute with a conceptual framework for object-oriented programming. Other important contributions to such a framework may be found in [Stefik & Bobrow 84], [Booch 84], [Knudsen & Thomsen 85], [Shriver & Wegner 87], [ECOOP 87] and [OOPSLA 87,88].

2. A conceptual framework for object-oriented programming

Background

The following definition of object-oriented programming is a result of the BETA Project and has formed the basis for the design of the object-oriented programming language BETA, [BETA87a].

Many object-oriented languages originate from SIMULA, either directly or indirectly via Smalltalk. Most of these languages represent a line of development characterized by everything being objects, and all activities being expressed by message passing. The definition and language are built directly on the philosophy behind SIMULA, but represent another line of development.

SIMULA was developed in order to describe complex systems consisting of objects, which in addition to being characterized by local operations also had their own individual sequences of actions. In SIMULA, this led to objects that may execute their actions as coroutines. This has disappeared in the Smalltalk line of development, while the line of development described here, has maintained this aspect of objects. While SIMULA simulates concurrency by coroutines, the model presented here incorporates real concurrency and a generalization of coroutines.

The following description of a conceptual framework for object-oriented programming is an introduction to the basic principles underlying the design of BETA. In section 3 the framework is illustrated by means of the BETA language. It is not attempted to give complete and detailed description of the basic principles nor to give a tutorial on BETA. Readers are referred to [DELTA 75], [Nygaard 86] and [BETA 87a] for further reading. This paper addresses the fundamental issues behind BETA, but it also has a practical side. The Mjølner [Mjølner] and Scala projects have produced a industrial prototype implementation of a BETA system, including compiler and support tools on SUN and Macintosh.

Short definition of object-oriented programming

In order to contrast the definition of object-oriented programming, we will briefly characterize some of the well-known perspectives on programming.

> *Procedural programming.* A program execution is regarded as a (partially ordered) sequence of procedure calls manipulating data structures.

This is the most common perspective on programming, and is supported by languages like Algol, Pascal, C and Ada.

> *Functional programming.* A program is regarded as a mathematical function, describing a relation between input and output.

The most prominent property of this perspective is that there are no variables and no notion of state. Lisp is an example of a language with excellent support for functional programming.

Constraint-oriented (logic) programming. A program is regarded as a set of equations describing relations between input and output.

This perspective is supported by e.g. Prolog.

The definitions above have the property that they can be understood by other than computer scientists. A definition of object-oriented programming should have the same property. We have arrived at the following definition:

Object-oriented programming. A program execution is regarded as a *physical model*, simulating the behavior of either a real or imaginary part of the world.

The notion of physical model shall be taken literally. Most people can imagine the construction of physical models by means of e.g. LEGO bricks. In the same way a program execution may be viewed as a physical model. Other perspectives on programming are made precise by some underlying model defining equations, relations, predicates, etc. For object-oriented programming the notion of physical models have to be elaborated.

Introduction to physical models

Physical models are based upon a conception of the reality in terms of *phenomena* and *concepts*, and as it will appear below, physical models will have elements which directly reflect phenomena and concepts.

Consider accounting systems and flight reservation systems as examples of parts of reality. The first step in making a physical model is to identify the relevant and interesting phenomena and concepts. In accounting systems there will be phenomena like invoices, while in flight reservation systems there will be phenomena like flights and reservations. In a model of an accounting system there will be elements that model specific invoices, and in a model of a flight reservation system there will be elements modelling specific flights and specific reservations.

The flight SK451 may belong to the general concept of flight. A specific reservation may belong to the general concept of reservation. These concepts will also be reflected in the physical models.

Figure 1. In the object-oriented perspective *physical* models of part of the real world are made by choosing which phenomena are relevant and which properties these phenomena have.

In order to make models based on the conception of the reality in terms of phenomena and concepts, we have to identify which aspects of these are relevant and which aspects are necessary and sufficient. This depends upon which class of physical models we want to make. The physical models, we are interested in, are models of those parts of reality we want to regard as information processes.

Aspects of phenomena

In information processes three aspects of phenomena have been identified: *substance*, *measurable properties* of substance and *transformations* on substance. These are general terms, and they may seem strange at a glance. In order to provide a feeling for what they capture, we have found a phenomenon (Garfield) from everyday life and illustrated the three aspects by figures 2- 4.

Substance is physical matter, characterized by a volume and a position in time and space. Examples of substances are specific persons, specific flights and specific computers. From the field of (programming) languages, variables, records and instances of various kinds are examples of substance.

Figure 2. An aspect of phenomena is substance. Substance is characterized by a volume and a position in time and space.

Substance may have *measurable properties*. Measurements may be compared and they may be described by types and values. Examples of measurable properties

are a person's weight, and the actual flying-time of a flight. The value of a variable is also an example of the result of the measurement of a measurable property.

Figure 3. Substance has measurable properties.

A *transformation on substance* is a partially ordered sequence of events that changes its measurable properties. Examples are eating (that will change the weight of a person) and pushing a button (changing the state of a vending machine). Reserving a seat on a flight is a transformation on (the property reserved of) a seat from being free to being reserved. Assignment is an example of a transformation of a variable.

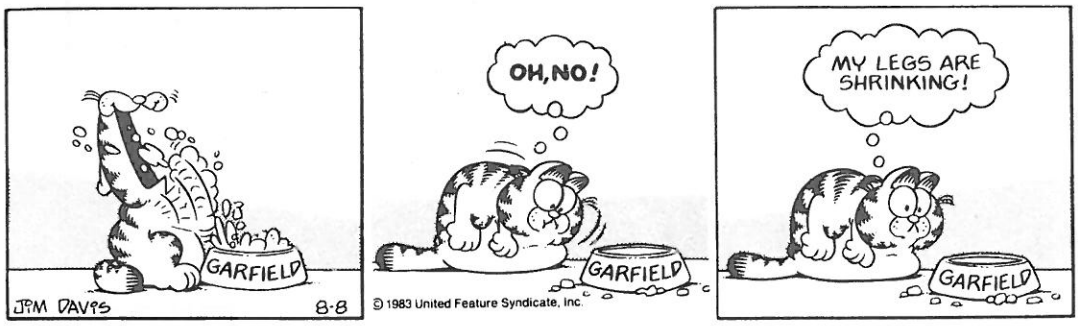


Figure 4. Actions may change the measurable properties of substance.

Aspects of concepts

Substance, measurable properties and transformations have been identified as the relevant aspects of phenomena in information processes. In order to capture the essential properties of phenomena being modelled it is necessary to develop abstractions or concepts.

The classical notion of a concept has the following elements: *name*: denoting the concept, *intension*: the properties characterizing the phenomena covered by the concept, and *extension*: the phenomena covered by the concept.

Concepts are created by *abstraction*, focussing on similar properties of phenomena and discarding differences. Three well-known sub-functions of abstraction have been identified. The most basic of these is *classification*. Classi-

fication is used to define which phenomena are covered by the concept. The reverse sub-function of classification is called *exemplification.*

Concepts are often defined by means of other concepts. The concept of a flight may be formed by using concepts like seat, flight identification, etc. This sub-function is called *aggregation.* The reverse sub-function is called *decomposition.*

Concepts may be organized in a *classification hierarchy.* A concept can be regarded as a *generalization* of a set of concepts. A well-known example from zoology is the taxonomy of animals: mammal is a generalization of predator and rodent, and predator is a generalization of lion and tiger. In addition, concepts may be regarded as *specializations* of other concepts. For example, predator is a specialization of mammal.

Elements of physical models: Modelling by objects with attributes and actions

Objects with properties and actions, and patterns of objects

Up till now we have only identified in which way the reality is viewed when a physical model is constructed, and which aspects of phenomena and concepts are essential. The next step is to define what a physical model itself consists of.

> A physical model consists of *objects,* each object characterized by *attributes* and a sequence of *actions.* Objects organize the substance aspect of phenomena, and transformations on substance are reflected by objects executing actions. Objects may have part-objects. An attribute may be a reference to a part object or to a separate object. Some attributes represent measurable properties of the object. The *state* of an object at a given moment is expressed by its substance, its measurable properties and the action going on then. The state of the whole model is the states of the objects in the model.

> In a physical model the elements reflecting concepts are called *patterns.* A pattern defines the common properties of a category of objects. Patterns may be organized in a classification hierarchy. Patterns may be attributes of objects.

Notice that a pattern is not a set, but an abstraction over objects. An implication of this is that patterns do not contribute to the state of the physical model. Patterns may be abstractions of substance, measurable properties and action sequences.

Consider the construction of a flight reservation system as a physical model. It will a.o. have objects representing flights, agents and reservations. A flight object will have a part object for each of the seats, while e.g. actual flying time will be a measurable property. When agents reserve seats they will get a display of the

flight. The seat objects will be displayed, so that a seat may be selected and reserved. An action will thus change the state of a seat object from being free to becoming reserved.

Reservations will be represented by objects with properties that identify the customer, the date of reservation and the flight/seat identification. The customer may simply be represented by name and address, while the flight/seat identification will be a reference to the separate flight object/seat object. If the agency has a customer database, the customer identification could also be a reference to a customer object.

As there will be several specific flights, a pattern Flight defining the properties of flight objects will be constructed. Each flight will be represented by an object generated according to the pattern Flight.

A travel agency will normally handle reservations of several kinds. A train trip reservation will also identify the customer and the date of reservation, but the seat reservation will differ from a flight reservation, as it will consist of (wagon, seat). A natural classification hierarchy will identify Reservation as a general reservation, with customer identification and date, and Flight Reservation and Train Reservation as two specializations of this.

Actions in a physical model

Many real world systems are characterized by consisting of objects that perform their sequences of actions *concurrently*. The flight reservation system will consist of several concurrent objects, e.g. flights and agents. Each agent performs its task concurrently with other agents. Flights will register the reservation of seats and ensure that no seats are reserved by two agents at the same time. Note that this kind of concurrency is an inherent property of the reality being modelled; it is not concurrency used in order to speed up computations.

Complex tasks, as those of the agents, are often considered to consist of several more or less independent activities. This is so even though they constitute only one sequence of actions and do not include concurrency. As an example consider the activities "tour planning", "customer service" and "invoicing". Each of these activities will consist of a sequence of actions.

A single agent will not have concurrent activities, but *alternate* between the different activities. The shifts will not only be determined by the agents themselves, but will be triggered by e.g. communication with other objects. An agent will e.g. shift from tour planning to customer service (by the telephone ringing), and resume the tour planning when the customer service is performed.

The action sequence of an agent may often be decomposed into *partial action sequences* that correspond to certain routines carried out several times as part of an activity. As an example, the invoicing activity may contain partial action sequences, each for writing a single invoice.

Actions in a physical model are performed by objects. The action sequence of an object may be executed *concurrently* with other action sequences, *alternating* (that is at most one at a time) with other action sequences, or as *part* of the action sequence of another object.

The definition of physical model given here is valid in general and not only for programming. A physical model of a railroad station may consist of objects like model train wagons, model locomotives, tracks, points and control posts. Some of the objects will perform actions: the locomotives will have an engine and the control posts may perform actions that imply e.g. shunting. Patterns will be reflected by the fact that these objects are made so that they have the same form and the same set of attributes and actions. In the process of designing large buildings, physical models are often used.

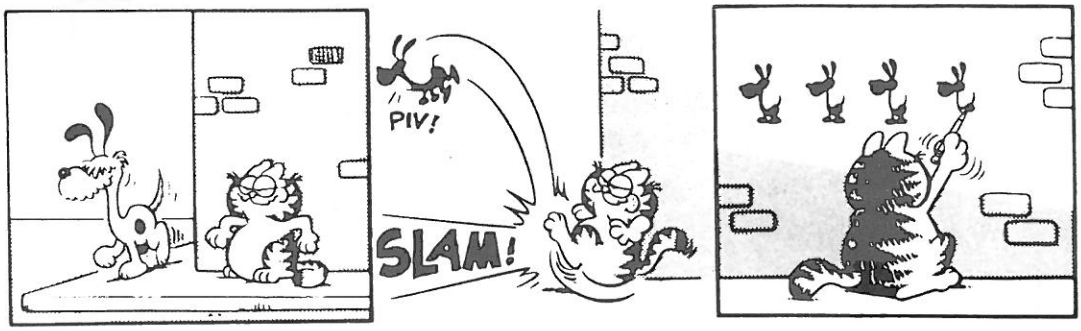


Figure 5. States are changed by objects performing actions that may involve other objects.

Object-oriented programming and language mechanisms supporting it

The notion of physical models may be applied to many fields of science and engineering. When applied to programming the implication is that the *program executions* are regarded as physical models, and we rephrase the definition of object-oriented programming:

Object-oriented programming. A program execution is regarded as a *physical model*, simulating the behavior of either a real or imaginary part of the world.

The ideal language supporting object-orientation should be able to prescribe models as defined above. Most elements of the framework presented above are represented in existing languages claimed to support object-orientation, but few cover them all. Most of them have a construct for describing objects. Constructs for describing patterns are in many languages represented in the form of classes, types, procedures/methods, and functions.

Classification is supported by most existing programming languages, by concepts like type, procedure, and class. Aggregation/decomposition is also supported by

most programming languages; a procedure may be defined by means of other procedures, and a type may be defined in terms of other types.

Language constructs for supporting generalization/specialization (often called sub-classing or inheritance) are often mentioned as the main characteristic of a programming language supporting object-orientation. It is true that inheritance was introduced in SIMULA and until recently inheritance was mainly associated with object-oriented programming. However, inheritance has started to appear in languages based on other perspectives as well.

Individual action sequences should be associated with objects, and concurrency should be supported. For many large applications support for persistent objects is needed.

Benefits of object-oriented programming

Physical models reflect reality in a natural way

One of the reasons that object-oriented programming has become so widely accepted and found to be convenient is that object orientation is close to the natural perception of the real world: viewed as consisting of object with properties and actions. Stein Krogdahl and Kai A. Olsen put it this way:

> "The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality, they are going to treat. It is then often easier to understand and get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs."
>
> (translated citation from "Modulær- og objekt orientert programming", DataTid Nr.9 sept 1986).

Physical model more stable than the functionality of a system

The principle behind the Jackson System Development method (JSD, [JSD]) also reflects the object-oriented perspective described above. Instead of focussing on the functionality of a system, the first step in the development of the system according to JSD is to make a physical model of the real world with which the system is concerned. This model then forms the basis for the different functions that the system may have. Functions may later be changed, and new functions may be added without changing the underlying model.

3. A language based on this model and comparisons with other languages

The definition above is directly reflected in the programming language BETA. The following gives a description of part of the transition from framework to language mechanisms. Emphasis is put on conveying an understanding of major language mechanisms, and of differences from other languages.

Objects and patterns

The BETA language is intended to describe program executions regarded as physical models. From the previous it follows that by physical model is meant a system of interacting objects. A BETA program is consequently a description of such a system. An object in BETA is characterized by a set of attributes and a sequence of actions. Attributes portray properties of objects. The syntactic element for describing an object is called an *object descriptor* and has the following form:

```
(#
  Decl1; Decl2; ...; Decln
do
  Imp
#)
```

where `Decl1; Decl2; ... Decln` are declarations of the attributes and `Imp` describes the actions of the objects in terms of imperatives.

In BETA a concept is modelled by a *pattern*. A pattern is defined by associating a name with an object descriptor:

```
P:(#  Decl1; Decl2; ...; Decln
  do
    Imp
  #)
```

The intension of the concept is given by the object descriptor, while the objects that are generated according to this descriptor, constitute the extension. So, while patterns model concepts, the objects model phenomena.

The fact that pattern and object are two very different things is reflected in their specification. An object according to the pattern `P` has the following specification

```
aP: @ P
```

where `aP` is the name of the object, and `P` identifies the pattern. The fact that some objects model singular phenomena is reflected in BETA: it is possible to describe objects that are not generated according to any pattern, but are singular. The object specification

```
S:@ (#  Decl1; Decl2; ...; Decln
    do
      Imp
    #)
```

describes a singular object `S`. The object `S` is not described as belonging to the extension of a concept, i.e. as an instance of a pattern. Singular objects are not just a convenient shorthand to avoid the invention of a pattern title. Often an application has only one phenomenon with a given descriptor, and it seems intuitively wrong to form a concept covering this single phenomenon. A search for a missing person in the radio includes a description of the person. This description is singular, since it is only intended to cover one specific phenomenon. From a description of a singular phenomenon it is, however, easy to form a concept covering all phenomena that match the description.

The framework presented here makes a distinction between phenomena and concepts, and this is reflected in the corresponding language: objects model phenomena and patterns model concepts. A pattern is not an object. In contrast to this distinction between objects and patterns, Smalltalk-like languages treat classes as objects. Concepts are thus both phenomena and used to classify phenomena. In the framework presented here, patterns may be treated as objects, but that is in the *programming process*. The objects manipulated in a programming environment will be descriptors in the program being developed.

Delegation based languages do not have a notion corresponding to patterns. They use objects as prototypes for other objects with the same properties as the prototype object.

References as attributes

An attribute of an object may be a *reference* that denotes another object. A referenced object may be either a part of the referent (the object containing the reference) or separate from the referent.

In the flight reservation system each flight is characterized by a number of seat objects. Each seat object will have a reference to a separate object representing the reservation, one reference to a part object representing the class of the seat and one reference to a part object representing whether the seat is a Smoking seat or not.

Given the pattern Seat,

```
Seat:(#
       Reserved:^ Reservation;
       Class: @ ClassType;
       Smoking: @ Boolean
     #)
```

a flight reservation system will contain the pattern `Flight` defined locally to pattern `FlightType`:

```
FlightType: (#
            source, destination: ...;

            Flight:
            (#
              Seats: [NoOfSeats] @ Seat
              ...
            #);

            DisplayTimeTableEntry: (# ... #);
            ...
          #)
```

For each entry in the time-table there will be a `FlightType`-object. `SK451` will be a `FlightType`-object. The actual flights on this route will be represented by `Flight`-objects. Scheduled departure time, flying time, and arrival time will be attributes of `FlightType`-objects, while actual departure time, flying time, and arrival time will be attributes of `Flight`-objects.

`Reserved, Class, Smoking` are references to objects, while `Seats` is a repetition of references to `Seat` objects.

Each `Flight`-object will consist of `NoOfSeats` objects of the pattern `Seat`. The lifetime of these `Seat`-objects will be the same as the lifetime of the `Flight`-object. Every `Seat`-object will consist of part objects that represent its class and whether it is a smoker's seat or not. In addition it will have a reference `Reserved`, to a `Reservation`-object representing the reservation of the seat (with customer identification, date, agent, etc.). The object referenced by `Reserved` will change if a reservation is cancelled or changed to another reservation. `Class` and `Smoking` are part objects as they represent properties that may be changed.

The fact that the substance of a phenomenon may consist of substances of part-phenomena is reflected in BETA by the possibility for objects to have part objects. Part objects are integral parts of the composite object, and they are generated as part of the generation of the composite object.

Most languages support part-objects of pre-defined types, or they model it by references to separate and often dynamically generated objects. One exception is composite objects of Loops [Stefik & Bobrow 82].

References in BETA are *qualified* (typed). A reference that may only denote `Reservation` objects, will thus be specified by

```
Reserved:^ Reservation
```

The reference `Reserved` may then not by accident be set to denote e.g. an `Invoice` object.

Patterns as attributes

Objects may be characterized by pattern attributes. A Smalltalk method is an example of a pattern attribute in the BETA terminology. In the example above `Flight` is a pattern attribute of `FlightType`. In addition `FlightType` instances have the attribute `destination` and the method `DisplayTimeTableEntry`. For the different instances `SK451` and `SK273` of the pattern `FlightType`, the attributes `SK451.destination` and `SK273.destination` are different attributes. In the same way `SK451.DisplayTimeTableEntry` and `SK273.DisplayTimeTableEntry` are different patterns, since they are attributes of different instances. In the same way the "classes" `SK451.Flight` and `SK273.Flight` are different. For further exploitation of "class attributes" see [Madsen 86].

Actions are executed by objects

Another consequence of the definition above is that every action performed during a program execution is performed by an object. For example, if `Seats` are to be displayed, then the display action must either be described as the actions of `Seat`-objects or as the actions of a local object.

In

```
Seat:(#...
       Display:
       (#   (* display Reserved, Class, and Smoking *)   #)
      #)
```

`Display` is a local pattern (here just described by a comment). In order to display a `Seat`-object (as part of the display of the Flight) a `Display`-object is generated and executed. In BETA this is specified by:

```
do ... ; Seats[inx].Display; ...
```

Most object-oriented languages has a construction like this. Lisp-based languages may use the form `Display(Seats[inx])`. From Smalltalk it has become known as "message passing", even though concurrent processes are not involved. It has the same semantics as a normal procedure call, the only difference is that the procedure is defined in a remote object and not globally. SIMULA introduced the notion of "remote procedure call" for this construction.

In BETA `Display` is a pattern attribute. While `Seat` is a pattern defining objects with attributes only, `Display` has an action-part, describing how `Seats` are displayed on the screen. The objects in BETA will thus have different functions (or missions) in a program execution, depending on their descriptor . No objects are a priori only "data objects with methods" and no objects are a priori only "methods".

Measurable properties

As an example of a measurable property, consider the percentage of occupied seats of a flight. A flight has parts like seats, but is has no part representing the percentage of occupied seats. This is a property that has to be measured. The value "85 %" is not an object, but is rather a denotation of the *value* of some measuring object.

Actions producing measurements

In BETA a measurable property is reflected by an object that produces a value as a result of executing the object. An object may therefore as part of its actions have an *exit*-part. This consists of a list of evaluations that represents the value of the object.

```
Flight:(#   Seats: [NoOfSeats] @ Seat;

            Occupied:
            (# ...
            do (* compute NoOfReservedSeats *)
            exit NoOfReservedSeats/NoOfSeats*100
            #)
        #)
```

While a `Flight` object will have `Seat` part objects, its `Occupied` will be a pattern attribute. In this example the specification of `Occupied` is just indicated.

Measurement of percentage of occupied seats is represented by execution of an object generated according to the `Occupied` pattern. This object will not be a part of the `Flight` object, but will temporarily exist when measuring the percentage of occupied seats.

In other languages this aspect is to some degree covered by function attributes.

Actions resulting in state changes

Change of state is usually associated with assignment to variables. In physical models there is a duality between observation of state (measurement) and change of state. A measurement is reflected in BETA by the execution of an object and production of a list of values that represents the value of the measurement. Correspondingly a change of state is reflected by reception of a list of values followed by execution of the actions of an object.

In order for an `OccupiedRecord` object (e.g. of a `Statistics` object) to receive the value of the percentage of occupied seats of a flight, the `OccupiedRecord` object will have an enter-part:

```
OccupiedRecord:@ (# Occ: @ Real enter Occ do ... #);
```

The percentage of occupied seats of a `Flight` object may then be measured and assigned to this object by

```
SK451.Occupied → Statistics.OccupiedRecord
```

The main actions of `SK451.Occupied` are executed (the actions described after **do**), its exit-part is transferred to the enter-part of `Statistics.OccupiedRecord` and the main actions of `Statistics.OccupiedRecord` are executed. As a side-effect the main action of `Statistics.OccupiedRecord` may e.g. count the number of assignments.

The association of reception of values with actions are also found in other languages (active, annotated values).

Classification hierarchies

As mentioned above a travel agency will normally handle reservations of several kinds. Classification of reservations into a general Reservation and two specializations Flight Reservation and Train Reservation will be reflected by corresponding patterns.

```
Reservation:
(#
  Date: (# ... #);
  Customer: (# ... #);
#)

FlightReservation: Reservation
                   (#
                      ReservedFligt: ^ Flight;
                      ReservedSeat: ^ Seat;
                   #)

TrainReservation:     Reservation
                   (#
                      ReservedTrain: ^ Train;
                      ReservedWagon: ^ Wagon;
                      ReservedSeat: ^ Seat;
                   #)
```

We will say that `FlightReservation` and `TrainReservation` are *sub-patterns* of `Reservation` and that `Reservation` is the *super-pattern* of `FlightReservation` and `TrainReservation`.

Besides supporting a natural way of classification, this mechanism contributes to making object-oriented programs compact. The general pattern only has to be described once. A revision of the pattern `Reservation` will have immediate effect on both sub-patterns. It also supports re-usability. All object-oriented languages, except delegation based languages, have this notion of class/sub-class.

Given this classification of reservations, the `Reserved` attribute (qualified by `Reservation`) of each `Flight` object may now denote `Reservation`,

`FlightReservation` and `TrainReservation` objects. In order to express that it may only denote `FlightReservation` objects, it is qualified by `FlightReservation`:

```
Reserved:^ FlightReservation
```

Virtuals

When making a classification hierarchy, some of the pattern attributes of the super-pattern are completely specified, and these specifications are valid for all possible specializations of the pattern. The printing of date and customer of a reservation will be the same for both kinds of reservation. Other attributes may only be partially specified, and first completely specified in specializations. The printing of reservations will depend upon whether a `FlightReservation` or a `TrainReservation` is to be printed.

The descriptor of `PrintDateAndCustomer` in `Reservation` will be valid for all kinds of reservations. The descriptor of `Print` will, however, depend upon which kind of reservation is to be printed. So this pattern may not be fully described in the pattern `Reservation`. It will perform `PrintDateAndCustomer`, but in addition it must print either flight/seat or train/wagon/seat. It is, however, important to be able to specify (as part of `Reservation`) that all `Reservation` objects have a `Print`, and that it may be specialized for the different kinds of Reservations. This is done by declaring `Print` as a *virtual pattern* in `Reservation`.

Declaring `Print` as a virtual in `Reservation` implies that

- in every sub-pattern of `Reservation`, `Print` can be specialized to what is appropriate for the actual sub-pattern, and

- execution of `Print` of some `Reservation` object, by

  ```
  do ...; SomeReservation.Print; ...
  ```

  where `SomeReservation` denotes some `Reservation` object, means execution of the `Print`, which is defined for the `Reservation` object *currently* denoted by `SomeReservation` is executed.

Execution of a virtual pattern implies late binding (to the `Print` pattern of the actual object denoted by `SomeReservation`), while qualification of `SomeReservation`, so that it may only denote objects of pattern `Reservation` or of sub-patterns of `Reservation`, assures that `SomeReservation.Print` will always be valid (`SomeReservation` will not be able to denote objects that do not have a `Print` attribute).

In languages like Smalltalk and Flavors all methods are virtuals, while in BETA, C++ [Stroustrup 86] and SIMULA it must be indicated explicitly. This means,

that message-passing in Smalltalk and Flavors always implies late binding, while non-virtuals in C++ , BETA and SIMULA may be bound earlier and thereby be executed faster. It also has the implication, that with non-virtual methods it is possible to state in a super-pattern, that some of the methods may *not* be specialized in sub-classes. This is useful when making packages of patterns. In order to ensure that these work as intended by the author, some of the methods should not be re-defined by users of the packages.

As methods in BETA are represented by pattern attributes, the ordinary pattern/sub-pattern mechanism is also valid for these. The virtual concept and specialization of methods are further exploited in [BETA 87b].

It is well-known that object-oriented design greatly improves the re-use of code. The main reason for this is sub-classing combined with virtuals [Meyer 87]. For many people this is the main issue of object-orientation. However, as pointed out above, modelling which reflects the real world is an equally important issue.

Individual action sequences

As mentioned above all actions in a BETA program execution are executed by objects. Each object has an individual sequence of actions. The model identifies three ways of organizing these sequences: as concurrent, alternating or partial sequences. In BETA, these are reflected by three different *kinds* of objects: *system* objects, *component* objects and *item* objects.

Concurrent action sequences

System objects are *concurrent* objects and they have means for synchronized communication: a system object may *request* another system object to execute one of its part-objects, and the requester will wait for the acceptor to do it. When the requested object *accepts* to execute this part-object, possible parameters will be transferred and the part-object executed. If parameters are to be returned, then the requesting object must wait until the part-object is executed.

In the BETA model of the flight reservation system mentioned above, agents and flights will be represented by concurrent objects, reflecting that there will be several agents, each of which at some points in time tries to reserve seats on the same flight. Seat reservations will take place by synchronized communication (the flight object will only perform one reservation at a time), so double reservation of the same seat is avoided.

For example, an Agent object may perform the following request

```
(date,3,window) → SK451 >? ReserveSeat
```

in order to reserve a window seat in 3th row.

The object SK451 may at this point in time be performing `ReserveSeat` for another Agent object, but when it performs the accept-imperative

```
<? ReserveSeat
```

then it will accept to perform `ReserveSeat`. The descriptor of `ReserveSeat` may contain a specification saying that it may be requested by *some* Agent object or only by one specific Agent object.

As each object may have their individual action sequence it may at different stages in this sequence accept different requests. When the flight is fully booked, it has come to a stage in its action sequence where it does not accept `ReserveSeat` requests.

The underlying model of a language determines to a certain degree which kind of concurrency is supported. While languages supporting objects as the main building blocks will have objects executing actions concurrently (even if this may only be accomplished by concurrent execution of methods as in Concurrent Smalltalk), Lisp-based languages will have concurrency based on concurrent evaluation of expressions (futures).

Alternating action sequences

Component objects in BETA are alternating objects, i.e. objects where at most one object is executing at a time and where the shift of control from object to object is non-deterministic.

In a BETA model of the system above the activities Tour Planning, Invoicing and Costumer Service will be represented by component part-objects of Agent objects, and each Agent will be an object, that executes these component objects alternately:

```
Agent:  (#  ...
        do...;
            (| TourPlanning | Invoicing | CostumerService |);
          ...
        #)
```

The activity TourPlanning may consist of planning a series of tours that are bought earlier, and just wait to be planned. Correspondingly, the activity Invoicing may consist of writing invoices for a series of tours. The activity CostumerService consists of waiting for customer requests and fulfilling them. A shift from TourPlanning or Invoicing to CostumerService will thus only take place, when there is a request for it, so it will not be part of the descriptor of neither TourPlanning nor Invoicing when this shall happen.

Partial action sequences

The example above with the execution of an object according to the Print attri-

bute of some Reservation object is an example of a partial action sequence, represented by an item object. The action sequence of Print is executed as part of the "calling" object.

All languages have a notion of partial action sequences. The notion of procedure and function covers this aspect of actions. Method invocation as a result of message passing is a special case of a partial action sequence, where the procedure to perform is defined in an object different from the invoking object.

One of the characteristics of most other object-oriented languages is that everything is an object with methods and all activity is expressed by message passing and method invocation. Objects in these languages do not have any individual action sequence; they are only "executing methods" on request. Thus, objects in these languages may not support alternating or concurrent action sequences. One exception is the Actor model of execution [Agha 86], where the objects are concurrent, but sub-classes are not supported in the common sense of the word. Work has been initiated to make concurrent Smalltalk, but this work does not include giving the objects their own sequence of actions.

4. What object-oriented programming does not have to be

As mentioned in the introduction many properties are associated with object-oriented programming. In this section we will comment on some of the misunderstandings (according to the definition given here) of object-oriented programming.

Everything is objects with methods, and all actions are message passing

A property common to most object-oriented programming languages is that everything has to be regarded as objects with methods and that every action performed is message passing. The implication of this is that even a typical functional expression such as

```
6+7
```

gets the unnatural interpretation

```
6.plus(7)
```

Even though 6 and 7 are objects (integer so), and they are also in the definition of object-oriented programming presented here, then there is no reason that + may not be regarded as an object that adds two integer objects:

```
plus(6,7)
```

Thinking object-oriented does not have to exclude functional expressions when that is more natural. Functions, types and values are in fact needed in order to describe measurable properties of objects.

Object-oriented programming and automatic storage management

According to the definition of object-oriented programming given here it has not necessarily anything to do with *dynamic generation of objects*. This is one of the properties often associated with object-oriented programming. In many object-oriented languages it is only possible to generate objects dynamically. But whether objects are parts of other objects or generated independently and dynamically, is not crucial for whether program executions are organized in objects or not. It is demonstrated above that in BETA it is possible to specify part-objects. Program executions are still organized in objects with attributes and actions, but some of the objects are allocated as part of other objects. As shown above a `Flight` object consists of `Seat` objects, and these are constituent parts of a `Flight` object. When objects are generated dynamically, as e.g. `Reservation` objects will be in the example above, it is, however, important that the implementation includes an automatic storage management system.

Late (and unsafe) binding of names gives slow execution (and unreliable systems)

Object-oriented programming does not necessarily imply *late and unsafe binding of names*. As mentioned above, pattern attributes of BETA objects and procedures in C++ objects may be specified as non-virtual, which means that late binding is not used when invoking them.

When Smalltalk or Flavors objects react on a message passed to it with "message not understood", it has nothing to do with Smalltalk or Flavors being object-oriented, but with the fact that they are untyped languages.

The combination of qualified (typed) references and virtuals in BETA implies that it may be checked at compile time that expressions like "aRef.aMethod" will be valid at run time, provided of course that aRef denotes an object and not **none**. And still a late binding determines which aMethod (of which sub-pattern) will be executed. Which aMethod to execute depends upon which object is currently denoted by aRef.

In the example above the reference `SomeReservation` will be qualified by `Reservation`. This means, that `SomeReservation` may denote objects generated according to the pattern `Reservation` or sub-patterns of `Reservation`. As Print is declared as a virtual in `Reservation`, it is assured that

```
SomeReservation.Print
```

is always valid and that it will lead to the execution of the appropriate `Print`. However, the use of untyped references in Smalltalk-like languages has the benefit, that recompilation of a class does not have to take the rest of the program into consideration .

What makes late binding slow is not only the method look-up. If a method in Smalltalk has parameters, then the correspondence between actual and formal parameters must be checked at the time of execution. `Print` will e.g. have a parameter telling how many copies to print. This will be the same for all specializations of `Print`, and should therefore be specified as part of the declaration of `Print` in `Reservation`.

In BETA this is obtained by *qualifying virtuals*. The fact that `Print` will have a parameter is described by a pattern `PrintParameter`:

```
PrintParameter:(# NoOfCopies: @ Integer; enter(NoOfCopies) do
... #)
```

Qualifying the virtual `Print` with `PrintParameter` implies that all specializations of `Print` in different sub-patterns of `Reservation` must be sub-patterns of `PrintParameter`, and thus have the properties described in `PrintParameter`. This implies that `Print` in all sub-patterns to `Reservation` will have an Integer `NoOfCopies` input-parameter.

If object-oriented programming is to be widely used in real application programming, then the provision of typed languages is a must. As Peter Wegner says in "Dimensions of Object-Based Language Design":

> "..., the accepted wisdom is that strongly typed object-oriented languages should be the norm for application programming and especially for programming in the large."

As demonstrated above it does not have to exclude flexibility in specialization of methods or late binding.

Inheritance/code sharing

Since inheritance has been introduced by object-oriented languages, object-oriented programming is often defined to be programming in languages that support inheritance. Inheritance may, however, also be supported by functional languages, where functions, types and values may be organized in a classification hierarchy.

In most object-oriented languages classes are special objects, and inheritance is defined by a message forwarding mechanism. Objects of subclasses send (forward) messages to the super-class "object" in order to have inherited methods performed. This approach stresses *code sharing*: there shall be only one copy of the super-class, common to all sub-classes. With this definition of inheritance it is not strange that "distribution is inconsistent with inheritance" [Wegner] and that "This explains why there are no languages with distributed processes that support inheritance" [Wegner].

In the model of object-oriented programming presented here, the main reason for sub-classing (specialization) is the classification of concepts. The way in which an object inherits a method from a super-class is - or rather should be - an implementation issue, and it should not be part of the language definition.

According to the definition of patterns and objects in BETA given above, patterns are not objects, and in principle every object of pattern P will have its own descriptor. It is left to the implementation to optimize by having different objects of P share the descriptor. Following this definition of patterns and objects there is no problem in having two objects of the same sub-pattern act concurrently and even be distributed. The implementation will in this case simply make as many copies of the pattern as needed, including a possible super-pattern. This does not exclude that a modification of the super-pattern will have effect on all sub-patterns.

Multiple inheritance

Multiple inheritance has come up as a generalization of single inheritance. With single inheritance a class may have at most one super-class, whereas multiple

inheritance allows a class to have several super-classes. Inheritance is used for many purposes including code sharing and hierarchical classification of concepts. In the BETA language inheritance is mainly intended for hierarchical classification. BETA does not have multiple inheritance, due to the lack of a profound theoretical understanding, and also because the current proposals seem technically very complicated.

In existing languages with multiple inheritance, the code-sharing part of the class/sub-class construct dominates. Flavors has a name that directly reflects what is going on: mixing some classes, so the resulting class has the desired flavor, that is the desired attributes. For the experience of eating an ice cone it is significant whether the vanilla ice is at the bottom and the chocolate ice on top, or the other way around. Correspondingly, a class that inherits from the classes (A, B) is not the same as a class that inherits from the classes (B, A).

If, however, multiple inheritance is to be regarded as a generalization of single inheritance and thereby as a model of multiple concept classification (and it should, in the model presented here), then the order of the super-classes should be insignificant. When classifying a concept as a specialization of several concepts, no order of the general concepts is implied, and that should be supported by the language.

Single inheritance is well suited for modelling a strict hierarchical classification of concepts, i.e. a hierarchy where the extensions of the specializations of a given concept are disjoint. Such hierarchies appear in many applications, and it is often useful to know that the extensions of say class predator and class rodent are disjoint.

By classifying objects by means of different and independent properties, several orthogonal strict hierarchies may be constructed. A group of people may be classified according to their profession leading to one hierarchy, and according to their nationality leading to another hierarchy. Multiple inheritance is often used for modelling the combination of such hierarchies. It may, however, be difficult to recognize if such a non-strict hierarchy is actually a combination of several strict hierarchies.

5. Conclusion

The programmer's perspective on programming is perhaps more important than programming language constructs. Object-oriented programming should not be defined only by specific language constructs. It is absolutely possible to think and program object-oriented even without a language that directly supports it.

It is a great advantage to use an object-oriented language that directly supports object-orientation. Such a language should support:

- Modelling of concepts and phenomena, i.e. the language must include constructs like class, type, procedure.
- Modelling classification hierarchies, i.e. sub-classing (inheritance) and virtuals.
- Modelling active objects, i.e. concurrency or coroutine sequencing, combined with persistency.

The benefits of object-oriented programming may be summarized as follows:

- Programs reflect reality.
- Model is more stable than functionality.
- Sub-classing and virtuals improve re-usability.

The above statements are of course not objective, in the sense that it is arguable what is natural, easy and stable. For a mathematician it may be more natural to construct a model using equations.

Finally we would like to stress that a programming language should support other perspectives than object-orientation. There are many problems that may be easier to formulate using procedural, functional or constraint-oriented programming. BETA supports procedural programming and has good facilities for functional programming. Work is going on to improve the support for functional programming and to include support for constraint-oriented programming. The overall perspective will still be object-oriented, but transitions may be described as functions without intermediate states. Support for constraint-oriented programming will allow for expressing constraints and for a more high level description of transitions. Loops is an example of a language supporting several perspectives.

Acknowledgement. The framework presented here is mainly a result of the authors' participation in the BETA project. In addition to the authors, Bent Bruun Kristensen, Aalborg University Centre, Denmark and Kristen Nygaard, University of Oslo, Norway, have been members of the BETA team. Jørgen Lindskov Knudsen, Kristine Stougård Thomsen, Jon Skretting and Einar Hodne have contributed with useful discussions and by commenting the paper. A very early version (in Danish) appeared in the special December 1987 issue of Nordisk Datanytt edited by Stein Gjessing.

References

[Agha 86] G. Agha: *An overview of Actor Languages.* Sigplan Notices Vol.21 No.10 October 1986.

[BETA 87a] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *The BETA Programming Language.* In: [Shriver & Wegner 87].

[BETA 87b] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Classification of Actions or Inheritance also for Methods.* Proceedings of the Second European Conference on Object Oriented Programming, Paris, June 1987.

[Booch 86] G. Booch: *Object-Oriented Development,* IEEE Trans. on Software Engineering, Vol. SE-12, No. 2, Feb. 1986.

[DELTA] E. Holbaek-Hanssen, P. Haandlykken, K. Nygaard: *System Description and the DELTA Language,* Publication no. 523, Norwegian Computing Center, 1975.

[ECOOP 87] *Proceedings of European Conference on Object-Oriented Programming.* BIGRE+GLOBULE No. 54, June 1987.

[Flavors] H. Cannon: Flavors, A Non-Hierarchical Approach to Object-oriented Programming. Draft 1982.

[JSD] M. Jackson: *System Development.* Prentice Hall 1983.

[Knudsen&Thomsen 85] J. Lindskov Knudsen and K. Stougård Thomsen: *A Conceptual Framework for Programming Languages.* DAIMI PB-192, Aarhus University, April 1985.

[Madsen 86] O.L. Madsen: *Block Structure and Object Oriented Languages.* In [Shriver & Wegner 87].

[Meyer 87] Reusability: *The Case for Object-Oriented Design.* IEEE Software, Vol.4, No.2, March 1987.

[Mjølner] *MJØLNER, A highly efficient Programming Environment for industrial use.* Mjølner Report No.1.

[Nygaard 86] K.Nygaard: *Basic Concepts in Object Oriented Programming.* Sigplan Notices Vol.21 No.10 October 1986.

[OOPSLA 87,88] *OOPSLA, Object oriented Programming Systems, Languages and Applications.* Conference Proceedings, 1986 and 1987.

[Shriver & Wegner 87] B. Shriver, P. Wegner: *Research Directions in Object-Oriented Languages*, MIT Press, 1987.

[SIMULA 67] O.J. Dahl, B. Myhrhaug & K. Nygaard: *SIMULA 67 Common Base Language*, Norwegian Computing Center, February 1968,1970,1972,1984.

[Smalltalk] A. Goldberg, D. Robson: *Smalltalk 80: The Language and its Implementation.* Addison Wesley 1983.

[Stefik & Bobrow 82] D.G. Bobrow and M. Stefik, : *Loops: An Object-Oriented Programming System for InterLisp,* Xerox PARC 1984.

[Stefik & Bobrow 84] M. Stefik, D.G. Bobrow: *Object-Oriented Programming: Themes and Variations,* The AI Magazine, 1984.

[Stroustrup 86] B. Stroustrup: *The C++ Programming Language.* Addison Wesley 1986

[Wegner] P. Wegner: *Dimensions of Object-Based Language Design.* Tech. Report No. CS-87-14, Brown University, July 1987.