# The object-oriented paradigm

*Document Version:*
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

*Please check the document version of this publication:*

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

Eindhoven University of Technology

Department of Mathematics and Computing Science

The object-oriented paradigm

by

P. America, M. van der Kammen, R.P. Nederpelt,
O.S. van Roosmalen and H.C.M. de Swart

# The object-oriented paradigm*

P. America[†]    M. van der Kammen    R.P. Nederpelt[**]    O.S. van Roosmalen [**]

H.C.M. de Swart[‡]

December 24, 1993

## Abstract

In this paper we discuss the fundamental concepts present in the object-oriented methodology.

First we concentrate on the notion of an object, the key concept in this approach. A (software) object is the abstract representation of a physical or conceptual object. It consists of a name, a specified set of data-elements and methods. Data-elements can have values attached to them.

Data-hiding is the feature that certain data and methods can be kept *invisible* (= hidden) for the outside of an object, thus facilitating its description. Only knowledge on the nature of the visible data-elements and methods is required to make proper use of the object. This is called data-abstraction. A related concept is encapsulation, a technique for achieving both data-hiding and data-abstraction.

A *class* is a template for a number of similar objects. Classes do not prescribe values for the data-elements nor fixed implementations for their methods. A class can be seen as a set of objects that satisfy the same specification for data-elements and method-behavior.

An alternative grouping of objects may take place by means of object types, as we will describe. A type is a set of objects that satisfy the same *external* specification, i.e., specification of the visible data-elements and methods. Thus, a classification via types differs from an ordering into classes, as we shall explain. The notion of type brings along a notion of subtyping.

We also discuss different forms of inheritance between classes. By means of inheritance a class can use data- and method-descriptions from another class. We describe, among other things, single inheritance, multiple inheritance and overriding. We also discuss multiple preferred inheritance and runtime inheritance.

Finally, we show how actual programming can take place in an object oriented approach. For that we need a description of inter-object communication by means of messages. Relevant aspects are: synchronous and asynchronous message passing, scheduling and delegation.

The paper concludes with an overview and a number of summarizing remarks.

---

1

# 1 Introduction

Recently, there has developed a growing interest in object-oriented programming, both in research and in the industry. It is more the name of a methodology than a collection of language features.

Languages can *support* this methodology, just like PASCAL *supports* structured programming. But, naturally, there is no language that *enforces* this approach to such an extend that one is guaranteed to develop 'proper' object-oriented systems.

Unfortunately, it is not easy to say what features are required in an object oriented language, although there is a growing consensus on the minimal set of features that must be present to call it object-oriented. This consensus causes a convergence in the features offered by popular object-oriented languages.

In this paper we shall discuss the most important aspect of object-orientation. The central concept is, of course, that of an *object*. We shall give an idea of what an object is and what can be its *values* (or *states*). By means of objects we can *encapsulate* data and code. Related concepts are *classes* and *types*.

When the concept of an object has been made sufficiently clear, we will take a look at inter-object relations. We will explain the idea of *inheritance*, as both an inter-class and an inter-object relation. Next we will discuss *message passing* between objects. Together with inheritance we will treat *subtyping*, which has been taken as a means of implementing and describing inheritance by some authors.

Finally, we will give an overview of our presentation and we draw a number of conclusions.

# 2 Objects and values

In the seventies, *structured programming* was one of the keywords used in the programming community ([Dahl et al. 72]). The rationale behind this approach was the growing conviction that the best solution for a realistic problem could be found in the use of a methodology that takes the logical structure into account. Structured programming is now generally accepted as a proper way of programming.

Object-oriented programming is based on the idea that the physical and conceptual objects in a problem domain can be used as a template to structure programs: First, reality is modelled as a set of objects, including object-properties and relations, that are relevant to the problem to be solved. Thus a problem is structured into small units that turn out to be relatively independent of each other. Second, the units thus obtained are directly mapped onto the program.

Naturally the modelling of reality is not an easy process. The actual partitioning that one obtains depends on many factors, the most important one being the way one perceives reality. Thus, on the one hand the approach offers intuitive means to support the analysis and design-process, on the other hand it puts a higher burden on the power of abstraction and creativity of the software-engineer.

The independent pieces of reality that one is looking for can be found by establishing the aspects that one considers of importance to the problem at hand. For example, for the modeling of an old-fashioned alarm-clock we could consider the following aspects as relevant: the time it is indicating (the current time); the point of time at which it is supposed to sound (the wake-up time); the little switches on the back with which the alarm time can be

changed; and the mechanisms it uses to operate. There are also things of lesser importance in this example, such as the substance the clock is made of. It follows that one usually considers only a small part of reality to be interesting. These interesting aspects can be further divided: the little wheels inside the alarm-clock are things that we do not take into consideration immediately, the direct concern is to be awoken at the proper time. Thus a distinction is made between externally observable aspects and internal ones.

| Reality | Abstract model |
|---------|----------------|
| my_alarm (the real thing) | **Object** |
| | *Name*<br>my_alarm |
| | *Data-elements*<br>current_time<br>wake_up_time |
| | *Methods*<br>set_current_time<br>set_wake_up_time |

Figure 1: The introduction of the abstract model.

The step to the abstract model is now very easy. We introduce for this purpose the notion of *objects*, that are tuples of *data-elements* and *methods* [1]. The data-elements of an object represent the modeled aspects, like the time the clock indicates, or the time it is set to sound. The methods of an object represent the transformations which can be performed on the data-elements. For example, we can change the time the alarm-clock is set to sound by turning the appropriate switch on the back.

An object can be seen ([America and Rutten 89]) as a black box which can store some data and act upon it. An object can only change its own data, and not that of another object. Other objects are involved in realizing an object's behavior, e.g. the wheels of the clock that implement its mechanism. The required cooperation is made possible through inter-object communication. This communication can be done in several ways as will be explained later. Objects can be created and destroyed. This creates a dynamic structure: the part of reality that the model is supposed to describe can change in the process.

An object-oriented program in execution will be called a *system*. Such a system can be thought of as being a varying set of communicating objects.

In figure 1, we show an abstract model of an alarm-clock in the form of an object. Note that this object is indeed an abstraction of reality, i.e. we can do more with a real alarm-clock than only setting the current time and the wake_up_time (even if the clock is not intended

---

[1] Different authors use different terminology. In e.g. [Madsen and Møller-Pedersen 88] these are called *attributes* and *actions*, respectively. One also often reads about *variables* and *procedures*.

to be used for other purposes). It is our personal choice (in accordance with our aims) to disregard all features of the alarm-clock that are not modeled.

The data-elements in an object have, apart from their name, a certain *value*. If it is nine o'clock PM, the value of the data-element `current_time` is supposed to reflect this, e.g. as the number `21:00`.

In [MacLennan 82] the values of the data-elements in an abstract model have the following four properties:
– abstraction: Values are abstractions from real values.
– changeability: Values do not change, they are constant and static. We cannot change the value `21:00`; the only thing that can change is the current time.
– state: Values do not have a state; because they are constant, they represent only one real value.
– referential transparency: If a data-element has a value $a$, and the value $a$ equals the value $b$, then the data-element has also value $b$.

About the last-mentioned property, we note the following. In general, a model has referential transparency if and only if equal values can be substituted for each other, without affecting the model. This is clearly the case with values: values cannot be duplicated, they are unique. Therefore there is referential transparency in the abstract model at value-level.
This is not so trivial as it seems; suppose we know that "The number of inhabitants of Amsterdam decreases" and that "The number of inhabitants of Amsterdam equals the number of inhabitants of Rotterdam". We do not necessarily also know then that "The number of inhabitants of Rotterdam decreases".
Here we encounter the difference between the *intension* and the *extension* of a notion. The number of inhabitants of Amsterdam is only equal to the number of inhabitants of Rotterdam in an extensional sense. It is in fact the extension, viz. the *value* of the number of inhabitants of Amsterdam, that equals the *value* of the number of inhabitants of Rotterdam. Now the number of inhabitants (the intension) can decrease, but the value of that number (the extension) cannot. (See [Dowty et al. 81].)

Values of data-elements can be used to characterize an object. The *state* of an object in the abstract model, at a particular moment, consists of the values of the data-elements (at that moment). We do not take methods as entries in the state. The reason is that the set of methods of an object does not change in time. Hence, the state of an object at any moment consists exclusively of values of data-elements. It changes in discrete steps, and not continuously. Moreover, an object is (in our abstraction) fully characterized by its state.

Of course, methods have their effect on the data-elements of an object. In order to guarantee that methods do not interfere, we assume that at any one moment, in any object, only one method can be active.

Suppose that an object contains other data-elements than the ones that actually occur in the methods of the object. Then these data-elements do not contribute to the intuitive meaning of the object, since they will never change, nor do they have an effect on any method. Therefore, we may consider not to include them in the state.
In the example depicted in figure 2, where $x$ and $y$ are the only data-elements occurring in the methods $m_0$ and $m_1$, the usefulness of $z$ could be questioned. As there is no method which makes use of or changes the value of $z$, it is inaccessible.

4

| Object |
| --- |
| *Name* |
| a |
| *Data-elements* |
| x |
| y |
| z |
| *Methods* |
| $m_0$ = ... x ... |
| $m_1$ = ... x ... y ... |

Figure 2: Too many data-elements.

Slight variations on these general principles for objects can be found in the literature. For example, in the language *POOL* ([America and Rutten 89]), objects do not only have data-elements and methods acting on these data-elements, but also a *body*, which is a process that starts executing upon creation of the object. In this fashion concurrency is introduced in *POOL*.

Another variation is proposed by [Goguen and Meseguer 87]. Objects are there seen as descriptions of reality using only equations. There is no distinction between data-elements and methods. Every object is meant to embody a relation between properties.

There is a clear distinction between objects and values (which is not always made in literature). Just like in real life, where the perception of a certain entity will change, objects — which represent this entity in the abstract model — can change too. If for example time changes (which continually happens), the current_time of my_alarm will get a different value. (We are not interested here in the question whether we have a continuum of values for the current_time, or that we only have a discrete set of values.)

Its original value, of course, does not change (as values are unchangeable), and neither does its new value. Only the data-element changes (see above; recall that a data-element is a pair of a name and a value).

It is further possible to have two alarm-clocks which are similar. If two alarm-clocks look very much alike, we are very soon inclined to say that they are equal. In our abstract model, it is therefore possible that two identical objects exist apart from each other: they form two instances of the same kind (see also section 4). We are then inclined to say that the two objects have the same state.

Of course, this equality can change. When one of the alarm-clocks is set to sound at a different time, its state changes and with that its equality to the other alarm-clock, which will still sound at the original time.

Concluding, we list some properties of an object as follows:
- abstraction: Objects are abstractions from pieces of reality.
- changeability: Objects are subject to change in time: they can be created, change their state and disappear.
- state: At any one moment in time, objects have exactly one state; this state is composed of the momentary value of all data-elements the object contains.

5

– referential transparency: Objects can have duplicates, which can act independently; two objects that have the same state (are "equal") at a certain moment can be different the next moment.

From our discussion above it is clear that there is no referential transparency at object level. This causes problems, for example with aliasing[2].

Objects can, as is clear from the above, be used to represent (sets of) values. The interested reader is encouraged to read [America and Rutten 89], where this is actually done.

# 3  Data-hiding, data-abstraction and encapsulation

The description of reality by means of objects gives a uniform view. Objects are seen as modules which consist of data and methods. For the object itself, it is very important *how* the methods change the data, but for a user of the object, this is of no interest. The mere fact *that* the data change is enough for the user, and no more.

Let's take the alarm-clock as an example again. The fact that its current time changes is something an observer sees. The observer notices that the hands of the alarm-clock move. But what the observer does not need (or want!) to know, is which wheels inside the alarm-clock do work. However, if the alarm-clock wouldn't know how to change the position of its hands, it would be useless. Therefore, the method turn_hands should be known to the object my_alarm, but need not be visible to any other object.

| **Reality** | **Abstract model** |
|---|---|
| my_alarm (the real thing) | **Object** |
| | *Name* <br> my_alarm |
| | *Data-elements* <br> current_time <br> wake_up_time |
| | *Hidden data-elements* <br> tick_count |
| | *Methods* <br> set_current_time <br> set_wake_up_time |
| | *Hidden methods* <br> turn_hands |

Figure 3: Hidden object features in the abstract model.

---

[2]We talk of *aliasing* whenever we have two names for the same thing, and use one of these names to change it; one of the mentioned problems is that one often forgets that it has also changed when we use this thing with the other name.

As we may infer from this example, the concept of *data-hiding* appears to be very important in object oriented approaches. Data-hiding is a way to keep irrelevant information away from the observer.

This idea is closely related to the modular approach of programming. A natural consequence of a modular approach is that sets of related methods and the data they manipulate are put together in one module; this facilitates the hiding of unimportant information (see [Stroustrup 87]). Actually, the term 'data-hiding' does not fully cover its load; we had better talk about *data- and method-hiding* (or *information-hiding*), as from the example given above it is already clear that methods are also in the same module. In figure 3 we have inserted examples of a hidden data-element and a hidden method in the abstract model.

In order to use a certain module, we need to have some kind of description of the module. There are two aspects of interest in this regard.

First, the description needs to tell which data-elements and methods are visible to the outside of the module and which are not. Second, it needs to give us a description of the possible values of visible data-elements and behavior of visible methods. In order to use a module, we need to know what it does and how we should use it in order to achieve a certain effect. This manner of describing a module as outlined above, is called *data-abstraction*.

The above description of a module is often called an *external interface*. The part of the description that tells us the behavior of the visible parts of the module is called the *external specification* of the module. Naturally, also an *internal specification* exists.

The concepts of data-hiding and data-abstraction are important aspects of the *encapsulation technique* (see [Snyder 86]):

> "**Encapsulation** *is a technique for minimizing interdependencies among separately written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers. If clients depend only on the external interface, the module can be reimplemented without affecting any clients, as long as the new implementation supports the same (or an upward compatible) external interface. Thus, the effects of compatible changes can be confined.*"

Generally, encapsulation is considered one of the main features of object-oriented programming. It provides the designer of programs with an easy way of *re-using* previously developed modules, and therefore offers an efficient (and clean) way of program development. However, reuse is stimulated by equally important other mechanisms in object-oriented programming languages, such as inheritance and genericity. These features distinguish object-oriented languages from other languages that support modularity (e.g. Ada, Modula-2). According to some estimations regarding the development of large software projects, up to 80% of the code can be re-used ([van Ginderen 90]).

An object has data-abstraction if it has an external interface which gives a certain interpretation of the externally visible data-elements and accompanying methods (or in other words, if it has an external specification)[3]. We are not interested in exactly what is inside the module, but we only want to know what it means and how we can use it.

---

[3]The objects in our abstract model can be seen as the abstract data objects of [Snyder 86]. The external behaviour of an object is fully defined by a set of abstract operations on the data of the object.

If an object has data-abstraction, we have the freedom to change the internal structure of the object, as long as we do not change the external interface that forms the interpretation of the data and methods. In [Stefik and Bobrow 86], data-abstraction is explained as the principle that modules should not make assumptions about implementations and internal representations of the modules they are using.

# 4 Classes

We are sometimes inclined to say that certain pieces of reality are very similar (of the "same kind"). Although the alarm-clock we have in mind may differ a lot from the one you have in mind, we all say that it is *an* alarm-clock.

To formalize the intuitive notion of this type of similarity in reality, we need some classification. Two pieces are of the same kind if and only if they have the same relevant aspects. Using this notion, anything that ticks, that has hands indicating the current time, on which the current time and the alarm time can be set, will be known as an *alarm-clock*.

Similarity is reflected in the abstract model by means of the notion of *class*. A class is meant to serve as a specification for objects. Any object which matches the specification belongs to the class. No distinction is made here between the visible and hidden features. A class gives internal specifications for methods (see below) and a pattern of data for objects of the same kind. It is not an object itself. However, given a class, objects of this class can be created. In figure 4 we illustrate the notion of class with our example of the alarm-clock.

A class is a set of method- and data-descriptions. The difference between classes and objects lies in this word *description*. Objects have values for their data-elements. Classes have descriptions for their data-elements (think of information on type, etc.). Also, objects-methods are fully detailed, whereas classes have possibly partial descriptions for their methods: objects belonging to the same class may have diferent implementation of their methods (see the discussion on inheritance and polymorphism further on).

Classes are descriptions of objects, and therefore consist of descriptions of data-elements and methods.

A description of a data-element consists of a name and a value-domain. There may also be restrictions on the combined values of data-elements, as expressed in so called class-invariants. A description of a method consists of its name, pre- and post-conditions, and often a default implementation which we will call the *internal specification* of the method.

To point out the difference between description and definition, we use the symbol ':' for the descriptions in classes, and '=' for the definitions in objects. See figure 5 for an example. We give the method-description by means of a Hoare-triple ($\{$time=T$\}$, set_time(t),$\{$time=t$\}$). Its meaning is: starting in a state in which the pre-condition time=T is satisfied, the execution of method set_time(t), if terminating, will lead to a state in which the post-condition time=t is satisfied[4]. Note that a description is given for the visible as well as the hidden data and methods of the class. Although a user of the corresponding objects requires only information on the visible aspects, the class description limits the way in which the externally observed behavior can internally be realized.

---

[4]In this simple case — contrary to the general situation — the pre-condition has no connection with the method or the post-condition: the original value of the time (T) has no consequences for the method set_time(t) or the new time t.

8

| Reality | Abstract model |
|---|---|
| my_alarm (the real thing) | **Object**<br><br>*Name*<br>my_alarm<br><br>*Class-Name*<br>alarm<br><br>*Data-elements*<br>current_time<br>wake_up_time<br><br>*Hidden data-elements*<br>tick_count<br><br>*Methods*<br>set_current_time<br>set_wake_up_time<br><br>*Hidden methods*<br>turn_hands |
| alarm (the real kind) | **Class**<br><br>*Name*<br>alarm<br><br>*Data descriptions*<br>current_time<br>wake_up_time<br><br>*Hidden-data descriptions*<br>tick_count<br><br>*Method descriptions*<br>set_current_time<br>set_wake_up_time<br><br>*Hidden-method descriptions*<br>turn_hands |

Figure 4: Classes and objects in the abstract model for alarm-clocks.

9

| Class |
| --- |
| *Name* |
| `alarm` |
| *Data descriptions* |
| `time:Int*Int` |
| *Hidden-data descriptions* |
| `ticks:Int` |
| *Method descriptions* |
| `{time=T }` |
| `set_time(t:Int*Int):[time:=t]` |
| `{time=t }` |
| *Hidden-method descriptions* |
| `{time=T }` |
| `turn_hands` |
| `{time=..}` |

| Object |
| --- |
| *Name* |
| `my_alarm` |
| *Class-Name* |
| `alarm` |
| *Data-elements* |
| `time = 11:10` |
| *Hidden data-elements* |
| `ticks= ..` |
| *Methods* |
| `set_time(t) = [time:=t]` |
| *Hidden methods* |
| `turn_hands=[..]` |

Figure 5: Classes and objects.

A class is often seen as a set of objects (see [Halbert and O'Brien 87]), where every object represents a different "value". This idea of value is of course not the same as the previous one. Here the "value" of an object is completely characterized by its state.

# 5 Types

In the previous section, we have grouped objects on the basis of the description of their data-elements and the internal specification of their methods. This gave rise to the notion of a class.

In this section, we will group objects in an alternative way, thereby creating *types*. In this approach, objects are grouped on the basis of their external behaviour; i.e., what is visible from the outside (cf. [America and Rutten 89]).

A *type* is determined by the external specification of an object, i.e. the specification of the names and types of the visible data-elements, types of method-arguments and names and returned results of methods, and the specification of the behavior of the methods. Two objects have the same type if their external specifications coincide.

The essence of the difference between class and type can be phrased as follows: a class groups together objects that are built in the same way while a type is a collection of objects that can be used in the same way.

In programming languages we do not always have types. Looking at pure PROLOG, there is no typing on the domain of the terms. But looking at PASCAL, we have a very strict notion of typing. For C things are different again. There, automatic type conversion plays an important role.

The use of types has various advantages, like the possibility of static type checking, resulting in a larger efficiency and a larger chance of correctness for programs, because there is less need for run-time checks. Another advantage is that domains for functions can be given as

types; for all function applications one then may check beforehand whether the argument of the function has the proper type, i.e., whether it fits in the domain.

Type compatibility is one of the main issues of typing. It is based upon an ordering on types, thus introducing notions like sub- and supertype. An assignment $x := E$ is allowed only if the type of $E$ is a subtype of the type of $x$. In some languages that support types, we can instruct the compiler to check the types, thus preventing execution of the code if the types are not compatible.

Porting this idea of subtyping to our abstract model, we arrive at what is described in the following.

By the very definition of class it is not possible that objects of the same class have different types (or external specification). This fact enables us to talk about the type of a class, instead of about the type of an object[5]. With any class, exactly one type can be associated.

We say that a type $A$ is a *subtype* of a type $B$ (and write $A \ll_{type} B$) iff adherence to the external specification $A$ implies adherence to the external specification $B$. This means, that any visible behavior of an object with type $A$ is in accordance with the specification $B$.

| Class |
|:---:|
| *Name* |
| STACK |
| *Hidden data descriptions* |
| n:integer |
| s:array of integer |
| *Method descriptions* |
| {s=W ∧ n=N} |
| push(x) |
| {s=W\|s[N]=x ∧ n=N+1} |
| |
| {s=W ∧ n=N} |
| pop |
| {s=W ∧ pop=s[N-1] ∧ n=N-1} |

Figure 6: The class STACK.

In figure 6 we can see the class STACK and the specification of methods **push, pop** and the data-elements s and n. The array s is used to contain the elements of the stack. Pushing is done from the bottom-up in this array. An object of class STACK represents a stack with the operations **push** and **pop**, specified as given in figure 6. We can see that all data-elements of objects of class STACK are hidden. Only the methods can be seen from the outside. The external behavior as produced by the methods push and pop is completely determined by the external specification: **pop(push(stack,x))** = x, stating that a pop delivers the last element pushed onto the stack.

In order to create a subtype of STACK, we need to look at its external specification. Important

---

[5] We are aware of the fact that we introduce some limitations, which may restrict the usefulness.

11

is that a subtype can at least "do" everything that STACK can, and possibly more. Consider the following class XSTACK (see figure 7).

| Class |
| --- |
| *Name*<br>XSTACK |
| *Hidden data descriptions*<br>m:integer<br>t:array of integer |
| *Method descriptions*<br>{t=W ∧ m=N}<br>push(x)<br>{t=W\|t[N]=x ∧ m=N-1}<br><br>{t=W ∧ m=N}<br>pop<br>{t=W ∧ pop=t[N+1] ∧ m=N+1}<br><br>{t=W ∧ m=N}<br>empty<br>{t=W ∧ m=0} |

Figure 7: The class XSTACK.

The specification of class XSTACK differs from the one of STACK. There is not only an additional method empty in XSTACK, but also a different internal representation: the array is built from the top down. This contrasts with STACK where the array is built bottom-up[6]. Also, the data-elements have different names. Nevertheless, XSTACK $\ll_{type}$ STACK since if we restrict ourselves to the use of the operations offered by both classes the same behavior is externally observed for objects of either class.

# 6  Inheritance

Typing is something extra, something to ensure correctness, to improve efficiency, which is not necessarily present in a programming language. *Inheritance* is something typical for object oriented programming and therefore is essential for any programming language which claims to support object-oriented design.

Many authors do not distinguish between the notions of subtyping and inheritance, e.g. in [Bruce and Wegner 86] we can find a very nice theory which describes inheritance using a subtyping relation. However, our point of view is that typing should not be used for other purposes than the ones given above. Inheritance is something typical for object-oriented approaches and if we use typing to describe it, its power is somewhat limited ([Cook et al. 90]).

---

[6]Note that m is of type integer and therefore can be negative. The method empty permits to start with an empty stack, initializing $m$ to zero.

In the following, we will describe inheritance, starting in a very simple form, and then extending it to a general form. Many other forms exist, but we will limit ourselves to the most important ones.

Inheritance can be described as a mechanism through which classes obtain data- and method-descriptions from other classes. Of course, our abstract model is supposed to be able to express this property. Therefore, we have a *linking function* between two classes.

We will use $\mathcal{D}_A$ and $\mathcal{M}_A$ for the set of data-element-names respectively the set of method-names for any class $A$. It will be the case that $\mathcal{D}_A \cap \mathcal{M}_A = \emptyset$ for any class $A$. Moreover, we use $\mathcal{E}_A$ for all entries in $\mathcal{D}_A \cup \mathcal{M}_A$. Hence, $\mathcal{E}_A$ contains all data-element-names and all method-names of class $A$.

**Definition 1** *Linking function.*
Given two classes $A$ and $B$, we call $\tau$ a linking function from $B$ to $A$ iff

$$\tau \in \mathcal{E}_B \xrightarrow{p} \mathcal{E}_A$$

such that[7] $\tau(\mathcal{D}_B) \subseteq \mathcal{D}_A$ and $\tau(\mathcal{M}_B) \subseteq \mathcal{M}_A$.
With $\xrightarrow{p}$ we denote a partial function.

We use a linking function to express the way we inherit a data-description or method-description from another class (note that the linking function implicitly consists of two distinct parts, one for data-descriptions and one for method-descriptions). The idea is that the linking function transfers some data- or method-descriptions of class $A$ to another data- or method-description of class $B$. If $\tau(b) = a$, then $a$ is inherited from class $A$ in class $B$ under the name $b$.

In the most simple form of inheritance, there are basically only two classes $A$ and $B$ involved, and $B$ inherits everything that $A$ has. This means, that every data- and method-description from $A$ is also in $B$. The linking function will there be used in order to find the origin of the description of a method or data-element.
This most basic form of inheritance we will call *complete inheritance*.

**Definition 2** *Complete inheritance.*
Given two classes $A$ and $B$ and a linking function $\tau$ from $B$ to $A$, we say that
$B \ll_{cpl}^{\tau} A$ ($B$ inherits complete from $A$) iff
(**for all** $a \in \mathcal{E}_A$ : (**there is a** $b \in \mathcal{E}_B$ : $(\tau(b) = a)$))
We say that the elements of $\mathcal{E}_A$ are inherited from $A$ by $B$.

For an example, see figure 8. The linking function from ALARM to CLOCK can be expressed as the following set of pairs: {(current_time,time),(set_current_time,set_time)}. In the following, we will use this notation in order to express the linking function.
All data- and method-descriptions from CLOCK are inherited by ALARM under a different name. This will complicate our discussion further on and therefore we will simplify this by giving the inherited items their original name. This means, that instead of having current_time and set_current_time in ALARM, we now have time and set_time, which have the same description. The linking function now becomes trivial and our class is slightly changed (see figure 9).

---

[7]We do not consider the option available in e.g. Eiffel ([Meyer 88]) to redefine a method as a data-element.

| Class |
|---|
| *Name* <br> CLOCK |
| *Data descriptions* <br> time :   ... |
| *Method descriptions* <br> set_time :   ... |

| Class |
|---|
| *Name* <br> ALARM |
| *Data descriptions* <br> wake_up_time :   ... <br> current_time :   time(from CLOCK) |
| *Method descriptions* <br> set_wake_up_time :  ... <br> set_current_time :   set_time (from CLOCK) |

Figure 8: Complete inheritance.

Note that it is not necessary to explicitly specify in the list of data-elements and methods of ALARM that we also have time and set_time. This information can be extracted from the complete inheritance of the class ALARM from the class CLOCK. Moreover, the method set_wake_up_time described in ALARM may use both set_time and time.

Class CLOCK itself can of course inherit from another class. This way a chain is formed along which complete inheritance takes place. For an example of such an inheritance chain, see figure 10. This possibility of chaining of inheritance is called *linearity*.

In order to find the description of set_time of class ALARM in the situation of figure 10, we use functional composition of the linking functions along the chain. In this case, we have a linking function $\tau_{\text{CLOCK\_METHODS}\to\text{CLOCK\_DATA}}$ and a linking function $\tau_{\text{ALARM}\to\text{CLOCK\_METHODS}}$, which are both quite trivial. Functional composition gives a linking function $\tau_{\text{ALARM}\to\text{CLOCK\_DATA}}$. We can compute the values of these functions by starting at the end of the chain, at the point where the data-elements and methods are actually described.

Therefore, we must require that these chains do indeed end. E.g., we must prevent to have that $A$ inherits from $B$ and $B$ inherits from $C$ and ... inherits from $A$. Checking on the presence of circular inheritance relations between classes is obviously the task of a language compiler.

Another problem that requires attention is the following. Suppose that the class ALARM in figure 9 has a method-description set_time in it. As class CLOCK, which is completely inherited by ALARM, also has a method-description set_time in it, it is not clear which of the methods (set_time) is meant when one talks of set_time in $\mathcal{M}_{\text{ALARM}}$. This problem of so called *name-clashes* will be discussed later.

Some generalizations of the notion of complete inheritance have been introduced. The first

14

| Class |
|---|
| *Name* |
| CLOCK |
| *Inheritance* |
| none |
| *Data-descriptions* |
| time : ... |
| *Method-descriptions* |
| set_time : ... |

| Class |
|---|
| *Name* |
| ALARM |
| *Inheritance* |
| complete from CLOCK |
| *Data-descriptions* |
| wake_up_time : ... |
| *Method-descriptions* |
| set_wake_up_time : ... |

Figure 9: Complete inheritance (simplified).

extension we will consider here is the one towards *incomplete inheritance*. The idea behind incomplete inheritance is that not all data-elements or methods are inherited from another class.

**Definition 3** *Incomplete inheritance.*
Given two classes $A$ and $B$ and linking function $\tau$ from $B$ to $A$, we say that
$B \ll^\tau_{\text{incpl}} A$ ($B$ inherits incomplete from $A$) iff
**(there is a $a \in \mathcal{E}_A$ : (there is a $b \in \mathcal{E}_B$ : ($\tau(b) = a$)))**
We say that the elements of $\mathcal{E}_A \cap \mathcal{E}_B$[8] are inherited from $A$ by $B$.

Note that from this definition it follows that

$$B \ll^\tau_{\text{cpl}} A \quad \text{implies} \quad B \ll^\tau_{\text{incpl}} A, \quad \text{if } \mathcal{E}_A \neq \emptyset.$$

In the case of incomplete inheritance, the linking function is not necessarily surjective, as not all data-element- or method-descriptions of class $A$ need to be inherited from $A$ by $B$. An example of this can be found in figure 11, where we have retained the name of the inherited items from CLOCK in ALARM. The linking function is obvious from the figure.

Similarly to the previous case, in the situation of figure 11 we can make the observation that $\mathcal{D}_{\text{ALARM}} = \{\text{wake\_up\_time,time}\}$ and $\mathcal{M}_{\text{ALARM}} = \{\text{set\_wake\_up\_time,set\_time}\}$. Of course it is possible that class CLOCK, in its turn, inherits the description time from another class. This

---

[8]which are data-descriptions in $\mathcal{D}_A \cap \mathcal{D}_B$ or method-descriptions in $\mathcal{M}_A \cap \mathcal{M}_B$
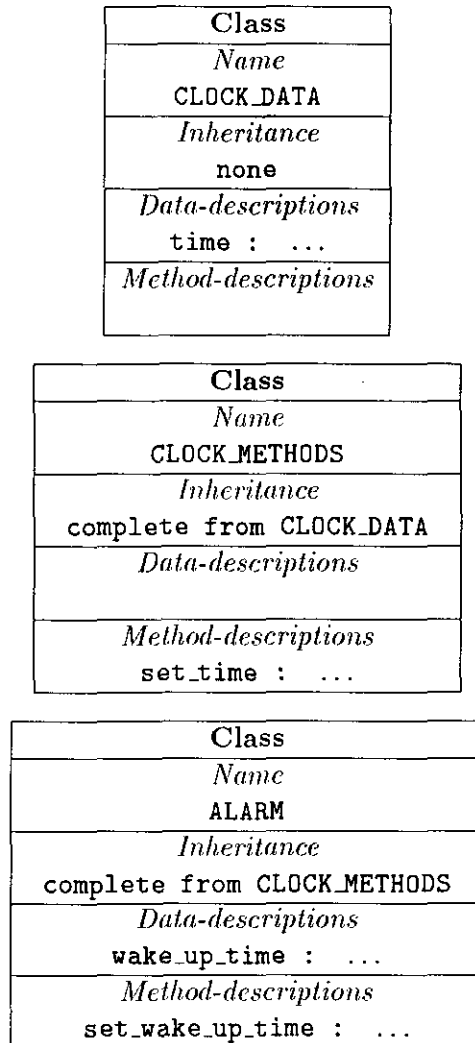
| Class | |
|---|---|
| *Name* | |
| CLOCK_DATA | |
| *Inheritance* | |
| none | |
| *Data-descriptions* | |
| time :   ... | |
| *Method-descriptions* | |

| Class | |
|---|---|
| *Name* | |
| CLOCK_METHODS | |
| *Inheritance* | |
| complete from CLOCK_DATA | |
| *Data-descriptions* | |
| | |
| *Method-descriptions* | |
| set_time :   ... | |

| Class | |
|---|---|
| *Name* | |
| ALARM | |
| *Inheritance* | |
| complete from CLOCK_METHODS | |
| *Data-descriptions* | |
| wake_up_time :   ... | |
| *Method-descriptions* | |
| set_wake_up_time :   ... | |

Figure 10: Complete inheritance (linear).

could even be the description wake_up_time from class ALARM!. But in the last-mentioned case, one would not allow that wake_up_time inherits from time. Hence, also incomplete inheritance must obey some form of linearity.

The use of incomplete inheritance as described above, is problematic. The reason is that set_time might use the data-element dual_time. Not inheriting this data-element renders set_time in ALARM useless. The programmer or compiler must check on the occurence of such inconsistent incomplete inheritance chains. Also from a more formal standpoint there is a drawback: incomplete inheritance no longer implies subtyping, e.g. if not all visible methods and data-elements are inherited. Therefore it will not be a surprise that incomplete inheritance hardly ever occurs.

Another generalization of complete inheritance is *multiple complete inheritance*. Data-element-descriptions and method-descriptions may be inherited from more than one class. In this case,

16

| Class |
|---|
| *Name* |
| CLOCK |
| *Inheritance* |
| none |
| *Data-descriptions* |
| time :  ... |
| dual_time :  ... |
| *Method-descriptions* |
| set_time :  ... |

| Class |
|---|
| *Name* |
| ALARM |
| *Inheritance* |
| time, set_time from CLOCK |
| *Data-descriptions* |
| wake_up_time :  ... |
| *Method-descriptions* |
| set_wake_up_time :  ... |

Figure 11: Incomplete inheritance (simplified).

our linking function should not only express which name is mapped to which name, but also from which class it stems. We create the *extended linking function*:

**Definition 4** *Extended linking function.*
Given a class $B$ and a set $S$ of classes with the property that $B \notin S$, we call $\tau$ an extended linking function from $B$ to $S$ iff

$$\tau \in ( \mathcal{E}_B \xrightarrow{p} \bigcup_{A \in S} [\{A\} * \mathcal{E}_A] )$$

such that $\tau(\mathcal{D}_B) \subseteq \bigcup_{A \in S}[\{A\} * \mathcal{D}_A]$ and $\tau(\mathcal{M}_B) \subseteq \bigcup_{A \in S}[\{A\} * \mathcal{M}_A]$.

The extended linking function gives us for each of the inherited data-elements and methods a tuple which contains the class-name (as a label) and the data-element- or method-name to which it is mapped. Multiple complete inheritance can now be defined as follows:

**Definition 5** *Multiple complete inheritance.*
Given a class $B$, a set $S$ of classes with $B \notin S$, and an extended linking function $\tau$ from B to S, we say that
$B \ll^\tau_{\text{mltpl cpl}} S$ ($B$ inherits multiple complete from $S$) iff
(**for all** $A \in S : B \ll^{\tau_A}_{\text{cpl}} A$), where $\tau_A$ is the projection of $\tau$ on $A$ (with the class-label $A$ omitted).
We say that the elements of $\bigcup_{A \in S} \mathcal{E}_A$ are inherited by $B$ from $S$.

| Class |
|---|
| *Name* |
| SILVER_BELL |
| *Data descriptions* |
| time : ... |
| *Method descriptions* |
| set : ... |
| reset : ... |

| Class |
|---|
| *Name* |
| CLOCK |
| *Data descriptions* |
| time : ... |
| color : ... |
| *Method descriptions* |
| set_time : ... |

| Class |
|---|
| *Name* |
| GOLDEN_BELL |
| *Data descriptions* |
| weight : ... |
| *Method descriptions* |
| set : ... |
| reset : ... |

| Class |
|---|
| *Name* |
| ALARM |
| *Data descriptions* |
| time : ... |
| dual_time : time (from SILVER_BELL) |
| wake_up_time : time (from CLOCK) |
| background_color : color (from CLOCK) |
| gross_weight : weight (from GOLDEN_BELL) |
| *Method descriptions* |
| set : set (from SILVER_BELL) |
| reset : reset (from SILVER_BELL) |
| set_wake_up_time : set_time (from CLOCK) |
| set_gross_weight : set (from GOLDEN_BELL) |
| reset_gross_weight : reset (from GOLDEN_BELL) |
| ring : ... |

Figure 12: Multiple complete inheritance.

In this definition, the surjective property for the extended linking function means that for every class $A \in S$ we have that for every data-element and method of $A$, there is a data-element resp. method in $B$ that is mapped to that one. An example that illustrates multiple complete inheritance is given in figure 12. The extended linking function belonging to this example is:

{(dual_time,(SILVER_BELL,time)), (wake_up_time,(CLOCK,time)),
(background_color,(CLOCK,color)), (gross_weight,(GOLDEN_BELL,weight)),
(set,(SILVER_BELL,set)), (reset,(SILVER_BELL,reset)),
(set_wake_up_time,(CLOCK,set_time)), (set_gross_weight,(GOLDEN_BELL,set)),
(reset_gross_weight,(GOLDEN_BELL,reset)) }.

As before one can run into the problem of name-clashes: refering to two methods or two data-elements with the same name. For example: in the above inheritance scheme, we cannot inherit the description of time under this name from both class SILVER_BELL and class CLOCK without introducing an ambiguity. The solution used in the figure is to perform an appropriate renaming [9]. We will discuss some alternative solutions.

---

[9]This, however, does not solve the problem entirely. Consider, e.g., the case where dual_time and

18

| Class |
| :---: |
| *Name* <br> ALARM |
| *Inheritance* <br> multiple preferred: <br> SILVER_BELL $\prec_{prf}$ CLOCK $\prec_{prf}$ GOLDEN_BELL |
| *Data-descriptions* <br> time : ... |
| *Method-descriptions* <br> ring : ... |

Figure 13: The preference relation for multiple incomplete inheritance.

The most obvious solution to name-clashes is to demand that all names be different:

$$[\textbf{for all } A_1, A_2 : A_1 \in S \wedge A_2 \in S \wedge A_1 \neq A_2 : \mathcal{E}_{A_1} \cap \mathcal{E}_{A_2} = \emptyset].$$

However, this places quite a burden upon the designer of the classes and it violates the principle of modularity. E.g., the designer of a new class $A$ should not be concerned about names of possibly even hidden data-elements of another class $B$, just because at some later point in time someone may decide to introduce a class that inherits form both $A$ and $B$. Therefore, this solution is inappropriate. It is remarkable, however, that this solution is nevertheless chosen in some existing object-oriented languages.

A second alternative takes us from multiple complete inheritance to a special form of *multiple incomplete inheritance*. We add a linear ordering (a so-called *preference relation*) to the set $\{B\} \cup S$. In this manner we create a chain, from which the first class is the most preferred one, and the last class the least preferred. The purpose of this chain is to introduce a priority: in case of an ambiguity for a reference of a data-element or method, we take the most preferred class.

This obviously requires that $B$ is always the first class in the chain (note that if we find the description there, it is actually not a case of inheritance, but just description-lookup). In the class under consideration, we only list the chain from the second element onwards, as we know that the class itself is always the first element.

In the above example, we could have the following ordering (from most to least preferred): SILVER_BELL $\prec_{prf}$ CLOCK $\prec_{prf}$ GOLDEN_BELL. Class ALARM (a more simple one than the one in figure 12) could look like the one in figure 13.

The inheritance relation in that example implies that $\mathcal{D}_{\text{ALARM}} = \{\text{time}, \text{color}, \text{weight}\}$ and that $\mathcal{M}_{\text{ALARM}} = \{\text{set}, \text{reset}, \text{set\_time}, \text{ring}\}$. The description of data-element time is obviously in class ALARM (as being the most preferred), for color in class CLOCK and for weight it can be found in class GOLDEN_BELL. For the methods set and reset the description is in

---

wake_up_time are derived from a single data-element of a common ancestor class. If in this so called *repeated-inheritance* situation an object is an instance of the class ALARM the ambiguity in the selection of implementation remains if the data-element is addressed as an element of the common ancestor. We refer here to the mechanism of dynamic binding which is not further discussed in this paper. A more extensive treatment of this issue can be found in [Meyer 88].

| Class |
|---|
| *Name* |
| GOLDEN_BELL |
| *Inheritance* |
| none |
| *Data-descriptions* |
| time :   ... |
| *Method-descriptions* |
| set :   ... |

| Class |
|---|
| *Name* |
| BELL |
| *Inheritance* |
| multiple preferred: |
| GOLDEN_BELL |
| *Data-descriptions* |
| color :   ... |
| *Method-descriptions* |
| set :   ... |

| Class |
|---|
| *Name* |
| CLOCK |
| *Inheritance* |
| none |
| *Data-descriptions* |
| time :   ... |
| *Method-descriptions* |
| set_time :   ... |

| Class |
|---|
| *Name* |
| ALARM |
| *Inheritance* |
| multiple preferred: |
| BELL $\prec_{prf}$ CLOCK |
| *Data-descriptions* |
| weight:   ... |
| *Method-descriptions* |
| reset :   ... |

Figure 14: Linearity in multiple preferred inheritance.

SILVER_BELL (being preferred above GOLDEN_BELL), for set_time in class CLOCK, and ring is described in class ALARM itself.

There are some drawbacks in this solution, however, which can be seen in the example. (1) Suppose we would want to use set of class SILVER_BELL and reset of class GOLDEN_BELL. There is no preference relation that lets ALARM inherit both of these methods. (2) The preference relation has an effect which is known as *overriding*. A data-element or method $\sigma$ from a class $C$ overrides the data-element or method with the same name from another class $D$ only if $C$ is more preferred than $D$. The preference mechanism therefore destroys the relation between inheritance and subtyping: if $A$ inherits from $B$ this no longer implies that $A$ is a subtype of $B$.

The multiple inheritance as described, with preference relation, we call *multiple preferred inheritance*. We will denote it by $\ll^\tau_{mltpl\ prf}$. An exact definition we consider outside the scope of this paper.

Of course, also in the case of multiple preferred inheritance, we desire linearity. Any class

```
                    weight; reset (ALARM)
                   _____/  _____

        color; set (BELL)    time; set_time (CLOCK)
                   |

        time; set (GOLDEN_BELL)
```
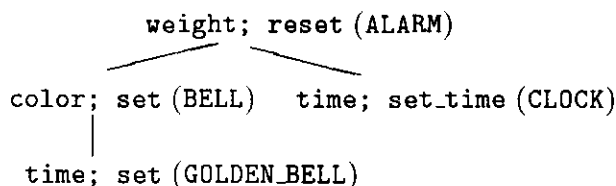
Figure 15: The tree of the preference relation.

$A_i \in S$ can inherit multiply preferred from a set of other classes. Therefore preference can be no longer interpreted as a chain. It is more like a tree, or even a graph, in which case we should be very careful with our inheritance. An example of this is given in figure 14. In this example, class ALARM inherits color from BELL, set from BELL and set_time from CLOCK. But what about time? Where does that come from? There are two basically different ways to resolve the conflict, corresponding to a *breadth-first* resp. a *depth-first* search strategy in the tree of the preference relation. Using the breadth-first approach, we get time from CLOCK, with depth-first we get it from GOLDEN_BELL, via BELL[10].

Most approaches choose this depth-first strategy, and therefore we will also do so. The tree for the example looks like the one in figure 15.

From this example it will be clear that it is quite hard for a programmer to keep track of the inheritance structure. Therefore, it should be used with lots of care.

One could consider other forms of inheritance. Most of these, however, can be classified among the above. As we mentioned before, incomplete inheritance is not frequently used. Complete and multiple preferred inheritance are the most popular forms.

The way we have described inheritance thusfar does not give us a flexible mechanism at runtime. During execution of a program, objects have a fixed set of data-elements and methods. In recent research (see [Shriver and Wegner 87]), however, there is more emphasis on having a flexible set. Especially with methods it seems useful to be able to change, add or subtract some definitions from this set. Following this approach, inheritance is taken from class-level to object-level. No descriptions are inherited, but definitions (actual code). Therefore, we call this form *runtime inheritance*. (See [Hailpern and Nguyen 87].)

A major problem that arises in this respect is that of consistency with class-level inheritance. The restriction that all objects of the same class inherit data-elements and methods from one or more classes is no longer a requirement. Objects of the same class can therefore have different sets of data-elements and methods. Or, seen otherwise, new classes can be created at runtime and objects can change their class!

A second problem is that inheritance is not from classes, but from objects. As objects of the same class can now have different sets of data-elements and methods, it is not possible to choose an arbitrary object of a class to inherit from. A solution often encountered is the use of *prototypes* (see [Lieberman 86]). For every class, one object is designated as the object

---

[10]With data-hiding in mind, we note the following. In the lastmentioned case, the class ALARM does not know that it inherits time from GOLDEN_BELL. It only knows that it inherits the description from BELL (which in turn gets it from GOLDEN_BELL).

21

from which inheritance takes place. A mechanism known as *delegation* (see further on) is then frequently used to implement the inheritance. Prolog++ (see [Prolog++ 90]) is an example of a language which uses this solution.

# 7 Inter-object communication: messages

Up to now, we have only discussed individual objects, classes and relations between them. We have intentionally left out the discussion on programming. In this section we will show how object-orientation is related to programming.

Thus far, a program consisted of a set of classes, with which we were able to create objects in the system. But this is not enough, as objects are unable to perform any action without being triggered (remember that our objects do not have bodies, i.e. processes that get executed upon creation of the object; cf. Section 2). A static set of objects merely forms a description, but is unable of performing any action. Objects should be able to communicate in order to create a dynamic structure.

Communication in object-oriented languages is performed by the aid of *messages*. Messages can be sent from any object to any other object to which it holds a reference. The receipt of a message triggers that object to perform an action, provided that the external interface of the object "recognizes" the message. A message therefore can be seen as a package containing the names of a sender-object and a receiver-object, plus a method to be invoked in the receiver, with a non-negative number of arguments for the method.

Objects can be *active* or *inactive*. In a system, the only active objects at any one moment are those that have been triggered by the receipt of a message, but have not yet finished the execution of the method invoked. All other objects are inactive.

There are two distinct ways of communication between objects. We can have *synchronous* or *asynchronous* message passing. The difference is described in [America and Rutten 89], and can be explained in short as follows.

With synchronous communication, the sender waits for a return value once a message has been sent (a kind of hand-shaking mechanism). This signals the completion of the method. During this wait the sending object is inactive. It becomes active again (it was active before sending the message) after the receipt of the return value.

It can easily be seen that using this way of communication, we always have exactly one object which is active (we assume that initially we have one active object).

Actually, we can speak of two different forms of message-passing, namely *implicit* and *explicit* message passing.

Implicit messages are messages which are sent by the communication protocol — like the inheritance mechanism — or completion messages. No object has code for these messages; there is no such thing as a send command for implicit messages.

With explicit messages this is different. These are the messages that are evoked on the request of the user of the program. Therefore these messages must be explicitly coded.

The return value of synchronous communication is sent as a so-called *completion message*; this is an example of an implicit message. It evokes a special code which handles "completion".

With asynchronous communication, we do not require the sender to wait for the completion of a method. Upon sending a message, the sender stays active and proceeds with the next

operation. This way, we can have multiple objects active at the same time. Obviously, we introduce some parallelism.

Moreover, we introduce a problem known as *scheduling*. Suppose we have an object which at the same moment receives two messages invoking the same method. Which message should be given priority, or should both messages be granted access at the same time, and what is then the effect? Or suppose that the object which receives the messages is already executing the requested method, because it received an earlier message for it. What happens then?

This scheduling problem is not related to asynchronicity, but more to parallelism. Therefore, we will not attempt to solve all the above-mentioned problems. On the other hand, we do feel that objects, independent as they are, should be allowed to execute their methods in parallel whenever possible.

In reality, it is impossible to have a mechanism which schedules messages and guarantees that a message sent will eventually be processed unless the method invoked aborts or ends up in an infinite loop. Individual starvation can be prevented by guarantees on the language level, but the prevention of deadlocks is a task for the programmer.

It is clear that an object can send a message which contains a reference to itself as an argument. In most languages, for this purpose the reserved word SELF is used in the description of the method. Especially when used with delegation (see below), a lot of attention should be paid to the question which objects are supposed to receive a completion message.

Data-elements are global to all methods of the object. Therefore, if we have a message sent to the object itself, the activated method acts upon the same data-elements as the sending method does. In fact, we have some kind of *in/out parameter passing*. When we use asynchronous communication, this leads to problems. We will demonstrate this by means of an example; see figure 16. The sending of messages is executed there by the (built-in) method send.

| Object |
|:---:|
| *Name* |
| alpha |
| *Class-Name* |
| A |
| *Data-elements* |
| n |
| *Methods* |

```
run = {   if n≤10   →   n:=n+1;
                         send(SELF,run);
                         n:=n-1;
                         print(n)
          □ n=11    →   skip
          fi
      }
```

Figure 16: The printing of some natural numbers.

The idea of the example is to print the numbers 0...10 in decreasing order. Method

**run** is used for this purpose. Suppose that n has initial value 0. If we use synchronous communication, the following is happening. Upon activation, the object increases the value of n, creates a new invocation of the method **run** and makes the old invocation inactive. This process repeats itself for n = 0 up to 10. That is to say: the object keeps on sending messages to itself until n reaches the value of 11. This is the "first round".

In the second round, each invocation becomes reactivated by a completion message from the invocation it created. This (re-)activation takes place in the reverse order. In each step, starting with n, the method does nothing but sending a completion message to the object, in response to which the object decreases n by 1 and prints the number obtained. In a stack-like fashion we now will get the numbers 0 ... 10 in decreasing order.

However, if we use asynchronous communication in this example, we can get completely different results. After the first message from object **alpha** to itself, it is not guaranteed that the next event in time will be the sending of another message. As the sender of the first message does not wait for the completion of a method, it can very well be that the first **print** statement is executed before the first message is even received. In that case, the first number printed can be 0.

An interesting application of message passing is *delegation*, which is a manner to simulate inheritance. With delegation, one supposes that every object tries first to answer a message itself. But if it fails to do so, it should forward the message to another object[11] that is of the class from which it was originally intended to inherit. Most languages allow objects to be created dynamically (at runtime). In order to determine the set of data-elements and methods of such a created object, one may use the prototype of the class to which the new object will belong. If no prototypes are available, objects can be created as copies of others.

Some approaches even allow more flexibility in object creation, and allow the programmer to define (parts of) the set of data-elements and methods. This way, class-less objects can be created. Or equivalently, new classes can be created at runtime. We feel that too much flexibility in object creation, just like too much flexibility in inheritance structure, is harmful to a language, because it places too high a burden on the programmer to ensure that the correct combination of data elements and methods is present.

# 8 Overview

In this paper, we have tried to build a model for the object-oriented approach. Not all the details of the model have been worked out. On the other hand, we sketched several directions in which the model can be elaborated.

In the introduction of the model we described how to abstract from reality into an abstract model. The basic modules in the model are called objects. A program in the model consists of a number of objects. We added the concepts of data-hiding and data-abstraction to our model, thereby creating classes. These classes were meant to group objects together, based on internal specification. An alternative way of grouping objects, based on external specification, was introduced in the form of the concept of type.

Next, we introduced inheritance in several forms, in order to show the possibility for modular development of programs. Inheritance is generally considered to be one of the main features

---

[11]The prototype mentioned before.

24

of object-oriented programming. Therefore we treated this notion extensively. Inheritance in our view is a relation between classes. There are alternatives like runtime inheritance, which has also been discussed. We mentioned that too much flexibility in inheritance structure can be harmful to a language, as programmers cannot easily keep the inheritance tree in mind.

Finally, we noted that a static model is not very interesting, nor representative for reality. Therefore we provided programs in our model with a message-passing mechanism. Messages can be sent from any object to any other object. Two distinct ways were discussed, namely synchronous and asynchronous message passing. We mentioned the problem of scheduling and how to use message-passing in implementing inheritance, a technique known as delegation. The message passing mechanism itself was not treated extensively. In fact, this is an implementation issue that we are not dealing with here.

In short, we have given a model for object-oriented programming, based on several notions and interpretations that can be found in the literature. Part of this literature is mentioned in our list of references. We have tried to synthesize the best parts and create a useful reference for anyone looking for a basic and general introduction to the main concepts in object-oriented programming. Our discussion might also be useful if one tries to establish what object-oriented features one desires to be part of a language that is to be used in a software project.

# 9 Conclusions

This paper discusses the main problems concerning the nature of object oriented programming. What does it mean if a program is called object oriented, what features should then be present?

In the literature, several aspects of object-oriented programming are treated in different versions. There is no complete agreement, not even with respect to the most fundamental concepts.
We have tried to compose an overview of what are considered the important aspects of object-oriented programming. We have tried to point out the strength and weakness of each aspect. Existing object-oriented languages usually offer a subset of these possibilities.

We summarize the results of our investigations:

- **Inheritance should be used with care**

A complicated inheritance tree makes it very hard for a designer to determine the data-element- and method-descriptions present in a class, and also makes it hard to re-use them, as the insertion of a new class in the tree may change a lot. Incomplete inheritance is even more dangerous, as the inheritance of a method does not guarantee that it can be executed (suppose it uses another method which is not inherited). Further, we feel that inheritance is a compile-time issue, at description level. However, some languages promote it to runtime, and treat classes as objects. Here it becomes very hard to reason about a program. Herewith one introduces a kind of self modifying code, as objects can change their set of data-elements and methods.

- **Object-orientedness is not a programming language feature but a design feature**

We have modeled reality into an abstract model. This abstract model has the object-oriented features like objects and classes. A programming language can offer or enforce object-orientedness by offering ways to implement these objects and classes. But as a program is the solution of a problem existing in reality, the process of designing the program should be object-oriented, not (only) the programming language.

● **Object-oriented programming is inherently imperative**

In object-oriented approaches, some of the main aspects are messages and states. Eliminating the states abolishes classes, leaving out messages makes computation impossible. Both are basic aspects, and the result of a message depends on the state of the object that receives it. As states change in time, due to transformations of the object, and since the system is fully determined by the states of all objects, object-oriented programming is imperative.

# References

[America and Rutten 89] P.H.M. America, J.J.M.M. Rutten, *A parallel object-oriented language: design and semantic foundations*, in J.W. de Bakker (ed.): *Languages for Parallel Architectures*, Wiley, 1989, pp. 1 - 49.

[America 90] P.H.M. America, *A behavioural approach to subtyping in object-oriented programming languages*, in *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Wiley, 1991, pp. 173 - 190.

[Bruce and Wegner 86] K.B. Bruce, P. Wegner, *An algebraic model of subtypes in object-oriented languages*, in *SIGPLAN Notices*, Vol. 21, No. 10, Oct 1986.

[Cook et al. 90] W.R. Cook, W.L. Hill, P.S. Canning, *Inheritance is not subtyping*, in *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, ACM-Press, January 1990.

[Dahl et al. 72] O-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press, New York, 1972.

[Dowty et al. 81] D.R. Dowty, R.E. Wall, S. Peters, *Introduction to Montague Semantics*, Reidel, 1981.

[Goguen and Meseguer 87] J.A. Goguen, J. Meseguer, *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, Report CSLI-87-93, Center for the Study of Language and Information, Stanford University, Palo Alto, 1987.

[Hailpern and Nguyen 87] Hailpern, B. and Nguyen, V., A model for object-based inheritance, in: B. Shriver, P. Wegner, eds., *Research Directions in Object-oriented Programming*, MIT-Press, 1987.

[Halbert and O'Brien 87] D.C. Halbert, P.D. O'Brien, *Using types and Inheritance in Object-Oriented Languages*, in *Proceedings of the 1st European Conference on Object-Oriented Programming, ECOOP87*, Springer Lecture Notes in Computer Science 276, 1987.

[Lieberman 86] H. Lieberman, *Using prototypical objects to implement shared behavior in object-oriented systems*, in *OOPSLA '86, Conference Proceedings*, ACM-SIGPLAN, Vol. 21, No. 11, Nov. 1986.

[MacLennan 82] B.J. MacLennan, *Values and objects in programming languages*, in *SIG-PLAN Notices*, Dec. 1982.

[Madsen and Møller-Pedersen 88] L.L. Madsen, B. Møller-Pedersen, *What object-oriented programming may be — and what it does not have to be*, in *Proceedings of the 2$^{nd}$ European Conference on Object-Oriented Programming, ECOOP88*, Springer Lecture Notes in Computer Science 322, 1988.

[Meyer 88] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[Nygaard 86] K. Nygaard, *Basic Concepts in Object Oriented Programming*, in *SIGPLAN Notices*, Vol. 21, No. 10, Oct 1986.

[Prolog++ 90] *Prolog++, Programming Reference Manual*, Version 1.0, Logic Programming Associates Ltd., London 1990.

[Shriver and Wegner 87] B. Shriver, P. Wegner, eds., *Research Directions in Object-oriented Programming*, MIT-Press, 1987.

[Snyder 86] A. Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, in *OOPSLA '86, Conference Proceedings*, ACM-SIGPLAN, Vol. 21, No. 11, Nov. 1986.

[Stefik and Bobrow 86] M. Stefik, D.G. Bobrow, *Object-Oriented Programming: Themes and Variations*, in *The AI Magazine*, 6, no. 4, 1986.

[Stroustrup 87] B. Stroustrup, *What is "Object-Oriented Programming"?*, in *Proceedings of the 1$^{st}$ European Conference on Object-Oriented Programming, ECOOP87*, Springer Lecture Notes in Computer Science 276, 1987.

[van der Kammen 91] M. van der Kammen, *The logic of objects; object-oriented programming in a logical perspective*, Master's Thesis, Eindhoven University of Technology, Dept. of Math. and Comp. Science, Eindhoven, 1991.

[van Ginderen 90] B. van Ginderen, *An object oriented hypertext system for learning*, Master's thesis, Eindhoven University of Technology, Open Universiteit Heerlen, 1990.

# Index

| 92/22 | R. Nederpelt<br>F.Kamareddine | A useful lambda notation, p. 17. |
|---|---|---|
| 92/23 | F.Kamareddine<br>E.Klein | Nominalization, Predication and Type Containment, p. 40. |
| 92/24 | M.Codish<br>D.Dams<br>Eyal Yardeni | Bottum-up Abstract Interpretation of Logic Programs,<br>p. 33. |
| 92/25 | E.Poll | A Programming Logic for Fω, p. 15. |
| 92/26 | T.H.W.Beelen<br>W.J.J.Stut<br>P.A.C.Verkoulen | A modelling method using MOVIE and SimCon/ExSpect,<br>p. 15. |
| 92/27 | B. Watson<br>G. Zwaan | A taxonomy of keyword pattern matching algorithms,<br>p. 50. |
| 93/01 | R. van Geldrop | Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36. |
| 93/02 | T. Verhoeff | A continuous version of the Prisoner's Dilemma, p. 17 |
| 93/03 | T. Verhoeff | Quicksort for linked lists, p. 8. |
| 93/04 | E.H.L. Aarts<br>J.H.M. Korst<br>P.J. Zwietering | Deterministic and randomized local search, p. 78. |
| 93/05 | J.C.M. Baeten<br>C. Verhoef | A congruence theorem for structured operational semantics with predicates, p. 18. |
| 93/06 | J.P. Veltkamp | On the unavoidability of metastable behaviour, p. 29 |
| 93/07 | P.D. Moerland | Exercises in Multiprogramming, p. 97 |
| 93/08 | J. Verhoosel | A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32. |
| 93/09 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part I: System Concepts, p. 72. |
| 93/10 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part II: Frameworks, p. 44. |
| 93/11 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part III: Modeling Methods, p. 101. |
| 93/12 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part IV: Analysis Methods, p. 63. |
| 93/13 | K.M. van Hee | Systems Engineering: a Formal Approach<br>Part V: Specification Language, p. 89. |
| 93/14 | J.C.M. Baeten<br>J.A. Bergstra | On Sequential Composition, Action Prefixes and Process Prefix, p. 21. |

93/34  J.C.M. Baeten and
       J.A. Bergstra            Real Time Process Algebra with Infinitesimals, p.39.

93/35  W. Ferrer and
       P. Severi               Abstract Reduction and Topology, p. 28.

93/36  J.C.M. Baeten and
       J.A. Bergstra            Non Interleaving Process Algebra, p. 17.

93/37  J. Brunekreef            Design and Analysis of
       J-P. Katoen             Dynamic Leader Election Protocols
       R. Koymans              in Broadcast Networks, p. 73.
       S. Mauw

93/38  C. Verhoef              A general conservative extension theorem in process
                               algebra, p. 17.

93/39  W.P.M. Nuijten          Job Shop Scheduling by Constraint Satisfaction, p. 22.
       E.H.L. Aarts
       D.A.A. van Erp Taalman Kip
       K.M. van Hee

93/40  P.D.V. van der Stok     A Hierarchical Membership Protocol for Synchronous
       M.M.M.P.J. Claessen     Distributed Systems, p. 43.
       D. Alstein

93/41  A. Bijlsma              Temporal operators viewed as predicate transformers,
                               p. 11.

93/42  P.M.P. Rambags          Automatic Verification of Regular Protocols in P/T Nets,
                               p. 23.

93/43  B.W. Watson             A taxomomy of finite automata construction algorithms,
                               p. 87.

93/44  B.W. Watson             A taxonomy of finite automata minimization algorithms,
                               p. 23.

93/45  E.J. Luit               A precise clock synchronization protocol,p.
       J.M.M. Martin

93/46  T. Kloks                Treewidth and Patwidth of Cocomparability graphs of
       D. Kratsch              Bounded Dimension, p. 14.
       J. Spinrad

93/47  W. v.d. Aalst           Browsing Semantics in the "Tower" Model, p. 19.
       P. De Bra
       G.J. Houben
       Y. Kornatzky

93/48  R. Gerth                Verifying Sequentially Consistent Memory using Interface
                               Refinement, p. 20.