Sharing Properties in a Uniform Object Space

Heiko Kießling and Uwe Krüger

University of Karlsruhe, Department of Informatics,
Operating Systems Research Group
Am Fasanengarten 5, D-76128 Karlsruhe, Germany

Abstract. We consider a system with a uniform object space. In a uniform object space, objects are the primary structural elements of the system. Such a system requires a different way of looking at the problem of sharing properties among objects. We describe the requirements on a suitable model of sharing and linguistic elements in the form of *views* and *patterns* to meet these requirements. With this model, we show that delegation both on the level of objects and as inheritance on the level of classes should only serve as a basis for implementing some of the linguistic elements of the model but not as such an element by itself. The model described in this paper finds itself in the language ARISTARCH/L which supports the object-based operating system ARISTARCH/OS that is under development in SAMOS, a project undertaken by the Operating Systems Research Group at the University of Karlsruhe.

1 Introduction

We consider a system with a uniform object space. The properties of an object in this object space reveal themselves in its *behaviour* observed by other objects. Abstractly speaking, an object shows a changing of behaviour over time. This changing of behaviour is determined by the (abstract) *state* of the object. Enduring structure of behaviour might be called the *behaviour pattern* of the object. In most object-based systems the behaviour pattern of an object is described by its operations. The execution of an operation depends upon the values of local variables. Effectively, these values are interpreted by the operations to represent the abstract state of the object.

In a uniform object space, objects are the primary structural elements of the system. In consequence the notion of an application is only latent as a set of cooperating objects rather than to manifest itself as structural elements of the system, that is a process created out of an object-based program. The elements describing the application, such as classes, do not come to life by calling a program but rather are represented as objects as well. From the uniformity of the object space directly follows the longevity of objects, in the sense that objects may survive the applications that created them. This is because an object may survive the execution of the operation that created it and the execution of an application is only latent as executing an operation of another object. Longevity thus lends itself to preserving existing information even in the face of adaptations by preserving pointers to adapted objects.

2 Requirements

A system with a uniform object space requires a different way of looking at the problem of sharing common behaviour among objects. Based on the preliminary considerations given in the introduction, we shall now put forward the requirements to be met by a suitable model by way of examples.

2.1 Direct Description of Objects

One of these requirements is the ability to directly describe objects instead of classes, as in general is necessary if a class has only one instance. Typical examples are value objects representing numbers, characters or truth values. Classes with only one instance are also common, whenever modules would be chosen in imperative languages. Szyperski [26], for instance, complains about the lack of support of the notion of module by many object-based languages. As examples for its usefulness he gives modules with global constants or mathematical libraries. But since in the kind of system considered here the notions of program and linking are pushed to the background as a consequence of applications being only latent as sets of objects, a module needs a runtime identity. One may now consider the main difference between modules and objects the fact that modules unlike objects lack such a runtime identity. Therefore the requirement of modules turns out to be the requirement of directly describing single objects without classes.

2.2 Direct Description of Aspects of Objects

Objects are often looked at under different aspects by different kinds of applications of the system. For instance, an object representing a person may be regarded as representing a taxpayer by the tax office but an employee by his or her employers. Under different aspects, the same object may show different behaviour.

In most object-based systems, objects owned by different applications work in separate environments. Different aspects of an object modelling the same entity are then represented by different objects in these separate environments. For example, in such a system an application in the tax office would model each person as an object merely representing the taxpayer aspect whereas an application in a company would only allow for the employee aspect. Nevertheless, often cooperation is necessary among different applications. For instance, though taxpayer and employee are represented by different objects, if the income, as an attribute of the employee, changes, the rate of taxation, an attribute of the taxpayer, should change as well. In general, it is necessary for different objects in separate environments to share common state. Most often, this is done by means of shared files, data bases and comparable mechanisms. The uniformity of the object space, however, allows different applications to work at least partially on the same set of objects. The same object now models both the taxpayer and the employee aspect.

Therefore, a dedicated linguistic element to describe aspects seems appropriate. Such an element should allow to describe different aspects separately to the greatest possible extent, but nonetheless in a single object, and to select the aspect of interest at a given time.

2.3 Categorization of Objects

To describe common behaviour of objects, it is necessary to divide them into different *categories* determining this common behaviour. The process of doing so is called *categorization*. It happens for the first time when an object is brought into the system, and in particular when it is created. A category may be defined by the concrete behaviour of its objects, or alternatively based on an abstract notion that separates it from other categories. The former kind of category is called *concrete*, whereas the latter is said to be *abstract*. While a concrete category statically determines behaviour, the behaviour specified by an abstract category may change over time according to the requirements on its objects. The differences between these two kinds of categorization shall now be elucidated by an example.

- By *concrete categorization* an object is assigned to a concrete category based on its behaviour to be modelled. We shall illustrate this by the example of two elephants introduced first by Lieberman [11], Clyde and Fred. Initially, the only elephant in a zoo is Clyde. In a visitor information system Clyde is represented by an object. When the zoo gets Fred, we recognize the similarities between the two of them, therefore putting Fred into the same concrete category as Clyde, which is given by the behaviour shown by the Clyde object at the time of creation of the object representing Fred. Creating an object by concrete categorization therefore means to copy the concrete behaviour of one or several other objects. Now, if Clyde changes his behaviour, for example by losing one of his legs due to an accident, this change should not be passed on to Fred. Because of the accident the object representing Clyde implicitly is assigned to another concrete category, that is it is *recategorized*. Concretely categorizing an object has a *momentary* nature since its membership of a concrete category changes whenever its behaviour changes.
- *Abstract categorization* is the result of an achievement of abstraction by the inventor of the abstract category, thus distinguishing it from other abstract categories. An abstract category therefore is independent of the concrete behaviour of its objects at any time. While a concrete category is determined by the modelled behaviour of its members, an abstract category determines part of the behaviour of its members.
  For example, in a zoo elephants are just one species among others. Therefore, elephants may form an abstract category. Common behaviour among all elephants are their habits. If there is scientific evidence for recently unknown habits, this part of behaviour is changed for all elephants at the same time, whereas their membership of the abstract category of elephants is preserved. Abstractly categorizing an object has a *permanent* nature since membership

is not affected by behaviour changes, but behaviour rather changes for all members at the same time.
Additionally, abstract categorization, in contrast to concrete categorization, in general is not related to the entire behaviour of its members. Parts of the behaviour may not be determined by the abstract category, or only determined in form of a behaviour pattern. For example, every elephant may possess a trunk, but the length of this trunk may be different for each specimen. The length of the trunk is not relevant to the category of elephants, whereas the property to possess one is essential. Altogether, we distinguish three different levels of abstract categorization, which we now shall explain in more conventional terms.

- *Categorization of Behaviour* strongly relates to class variables, as a part of state common to all instances of the class. As operations together with variables determine behaviour, this is synonymous with describing behaviour essential to the abstract category modelled by the class.
- *Categorization of Behaviour Pattern* in general is given by having the same operations for all instances of a class operating on local variables of each instance.
- *No Categorization* to our knowledge is not supported by existing class-based systems. For example, assume now that one of the elephants in the zoo is capable of walking upright. This may or may not be the first step of evolution towards upright walking elephants but, for the moment being, she is really unique. She remains in the same category as her fellows but she still shows a behaviour that distinguishes her from all the others. We may conclude that an important requirement on an object model for a system with a uniform object space is the extensibility of abstractly categorized objects.

In Sect. 1 we have argued that longevity is a natural property of objects in a uniform object space. The different forms of categorization together with longevity lend themselves to preserving existing information in the face of adaptations by systematically adapting single or entire sets of objects while preserving pointers.

Abstract categorization of behaviour may lead to different categories which determine different behaviour nevertheless based on a *common* behaviour pattern. Therefore, an important requirement on languages in a uniform object space is to enable programmers to categorize abstract categories by this common behaviour pattern. We shall elucidate this by the example of a class of character strings. To permit the efficient representation and comparison of strings, often a table is used which holds the different character sequences that occur in strings. Strings themselves merely possess indices into this table, strings containing the same sequence possessing the same index. The table may be thought of as a part of the state of each string that is common to all strings. It finds its natural place in the class of strings which therefore are categorized by using a certain table besides by possessing a common behaviour pattern. If it is not necessary to pass a string out of a certain context, for instance an application, it seems reasonable to use a separate table in this context to achieve higher efficiency. This may be

seen even more clearly if we assume a multiple user object space. Such an object space enables several users represented by proxy objects to cooperate with each other via commonly known objects. In this case it may not be just inefficient but plainly wrong to assign to all users the same string class, and thus the same string table, for depending on the functionality supported by strings a user may find out which character sequences are defined by others. This again necessitates multiple string classes for which access may be restricted separately. Put in conventional terms, the requirement is to enable programmers to describe classes with different states but instances which share the same component structure.

Some of the requirements described above seem to be met by delegation both on the level of objects and as inheritance on the level of classes. Lieberman argues for delegation as a means of abstract categorization while object-oriented languages to this end use classes and inheritance. In the following sections we will present a model for describing common behaviour of objects that we believe meets the requirements given above very well. With it we will show that delegation both on the level of objects and as inheritance on the level of classes should only serve as a basis for implementing some of the linguistic elements of the model but not as such an element by itself. The implementation of these elements apart from its relationship to delegation is out of the scope of this paper.

In the following section we will first clarify our terminology. In Sect. 4, we will then present our approach to the description of different aspects of objects. After that, we will deal with the problem of categorization in Sect. 5. Both sections close with a critique of the notion of delegation. In the closing section we will compare our model to related work. The model described here finds itself in the language ARISTARCH/L which supports the object-based operating system ARISTARCH/OS that is under development in SAMOS, a project undertaken by the Operating Systems Research Group at the University of Karlsruhe. Therefore examples are shown in this language but are also explained in the text.

3 Objects

In this section we clarify terminology in a concise way. In the following, we will have to strongly distinguish between the *definition* and the *implementation* of the components of an object. For now it is sufficient to assume *attributes* and *commands* as components. An attribute consists of two commands, one for reading it and one for writing it. These are the terms used when talking about the definition of an object. The implementation of the object describes how attributes and commands are realized. Every command is implemented by a *method*. Accordingly, every attribute is implemented by two methods, one for reading and one for writing, or by a *variable* which implicitly defines these two methods. This allows customizations of attribute implementations similar to, for instance, those in TRELLIS/OWL designed by Schaffert and Cooper [22]. In ARISTARCH/L, definition and implementation of an object are described by means of an *object expression* which may be understood as evaluating to a constant reference to

the object. To this end, each object expression has an **IMPLEMENTATION** section which describes both the definition and the implementation of the object and is divided into an **ATTRIBUTES** part and a **COMMANDS** part. Object expressions meet the requirement for direct description of objects stated in Sect. 2.1.

4 Views

4.1 Introduction

Objects are looked at from different aspects by different kinds of applications of the system. For instance, an object representing a person may be regarded as representing a taxpayer by the tax office but representing an employee by his or her employer. To this end, in ARISTARCH/L each object may possess one or more *views*. A view describes an aspect of an object by declaring a logically closed subset of its components, the *interface* of this view. These interfaces are included in the **INTERFACES** section of the object. Besides representing different aspects of an object interfaces of views also serve information hiding.

An object is always considered under one of its views. To achieve this, each reference to an object not only determines its identity but also the view it is currently used under. It is possible, starting from a reference, to get a reference to the same object under another view.

Example 4.1 describes three different aspects of a person in the left code fragment. A general view **Person** is restricted to basic information such as the name (lines 3 to 9). The view **Employee** additionally consists of the personnel number and income (lines 11 to 19), while the view **Taxpayer** (lines 21 to 29) includes the rate of taxation and the tax number[1]. If an object expression describes several different views the **AS** clause may be used to determine the initial view to use the object under (line 31). An application in a tax office may look at the person from its most pleasant aspect that is, a taxpayer, by switching to the appropriate view with the operation **VIEW person AS Taxpayer** (line 12 of the right fragment). The reference obtained by this operation allows access via the interface of the view **Taxpayer** and thus to the attribute **taxrate**. Please note that, as description of an aspect of an object, each view is given exactly once.

Each component may have different *versions of implementation* to cater to the object's different aspects. Hence each attribute and each command may be implemented by several different variables and methods. For example, assume a command **State** giving information about a person. This information for a **Taxpayer** is something different from what it is for an **Employee**. Therefore, a programmer will create at least two versions of the method implementing the command **State**.

4.2 Selecting Versions of Implementation

To execute the correct version of implementation corresponding to a certain view each view possesses a *mapping* which assigns a version to every component. Each

[1] *Derivation* of views allows for such constellations but is deferred to Sect. 4.4.

```
1 OBJECT
    INTERFACES
      VIEW Person
        ATTRIBUTES
5         IMMUTABLE name: tString
        COMMANDS
          State ();
          Action ()
      END VIEW Person;
10
      VIEW Employee
        ATTRIBUTES
          IMMUTABLE name: tString;
          personnelNumber: INTEGER;
15        income: INTEGER
        COMMANDS
          State ();
          Action ()
      END VIEW Employee;
20
      VIEW Taxpayer
        ATTRIBUTES
          IMMUTABLE name: tString;
          taxNumber: INTEGER;
25        IMMUTABLE taxrate: INTEGER
        COMMANDS
          State ();
          Action ()
      END VIEW Taxpayer;
30  ...
  END OBJECT AS Person -> person
```

```
1 OBJECT
    ...
    IMPLEMENTATION
      COMMANDS
5       TaxReturn (person: tPerson)
          BY METHOD
            DECLARE
              ATTRIBUTES
                t: tTaxpayer BY VARIABLE;
10            ...
            DO
              VIEW person AS Taxpayer -> t;
              ... t.taxrate ...
        END TaxReturn;
15  ...
  END OBJECT -> taxoffice
```

Ex. 4.1. Aspects of a **person**

version must be used in the mapping of at least one view. Since different views may use the same version it is not sensible to describe mappings by syntactically nesting versions into views. Instead, different versions of a component get identifiers which the mapping may refer to. These identifiers are only unambiguous in connection with the identifier of the component. The reason for this is the basic notion of *identifiers as specifications* of components in ARISTARCH/L. The type system of ARISTARCH/L is also based on this notion, and in this respect is comparable to that of EMERALD [3], but out of the scope of this paper.

While in Ex. 4.1 views were merely used to describe different subsets of the components of **person**, Ex. 4.2 employs the command **State** to show the use of different versions specific to views. In the **IMPLEMENTATION** section three versions of the method for **State** are defined (lines 21 to 29), printing the information appropriate to the aspects represented by different views. In the **MAPPINGS** section (lines 4 to 13), the versions are assigned to the views **Person**, **Employee**, and **Taxpayer**, respectively. Components like **State** are marked with the keyword **ASSERTED** (lines 16, 17, 20, and 31). This mark must be carried by all components given in the interface of any view of an object. It has the advantage of hidden components to be immediately recognizable in their implementation.

```
OBJECT
  ...
  IMPLEMENTATION
    MAPPINGS
      VIEW Person ...
        State BY BasicVersion; ...
      END VIEW Person;
      VIEW Employee ...
        State BY EmployeeVersion; ...
      END VIEW Employee;
      VIEW Taxpayer ...
        State BY TaxpayerVersion; ...
      END VIEW Taxpayer

    ATTRIBUTES
      ASSERTED name: tString BY VARIABLE;
      ASSERTED personnelNumber: INTEGER BY VARIABLE;
      ...
    COMMANDS
      ASSERTED State () BY
        VERSION BasicVersion BY METHOD
          Output Personal Data
        END VERSION BasicVersion,
        VERSION EmployeeVersion BY METHOD
          Output Personal and Employee Data
        END VERSION EmployeeVersion
        VERSION TaxpayerVersion BY METHOD
          Output Personal and Taxpayer Data
        END VERSION TaxpayerVersion
      END State;
      ASSERTED Action () BY METHOD
          ... State ();  ..
      END Action;
      ...
END OBJECT AS Person -> person
```

Ex. 4.2. Views and Versions

An object, that is its implementation, may use all of its own components. It does not use them under a specific view, however. Therefore, a generalized way to select versions of implementation for components is needed. For instance, assume that the command **State** giving information about a person now is to be used also for the purpose of debugging other parts of the implementation. To this end, it is called in the methods of the other commands of the object **person**. The problem, however, is to select the right version of the method for **State**. In the given case, it suffices to create several different versions of each method corresponding to the views provided by the object which differ merely in which output command they call. This approach is costly to program, but insufficient as well if the commands to be debugged sometimes are also called directly by the object. In general, if a command was issued by another object under a certain view the versions of implementation for all components used directly or indirectly by the method of this command should also be determined by the view. For this reason, the mapping of a view always has to assign a version

to all components of the object, not just to those given in its interface. If for a component there is only one version, it need not be explicitly handled in the mapping section of a view, however.

In Ex. 4.2, the command **Action** is accessible via all views (lines 8, 18, and 28 of the left code fragment of Ex. 4.1) but there is only one method version (lines 31 to 33), this being the reason why there is no entry for **Action** into the **MAPPINGS** section. The method of **Action** calls **State** to print debugging information (line 32). The version for **State** is selected by the view **Action** was called under. So, if **Action** was called under the view **Employee**, the version **EmployeeVersion** of **State** is called. On the other hand, if **Action** was called under the view **Taxpayer**, the version **TaxpayerVersion** of **State** is called.

4.3 Cooperation among Aspects

Despite having several different aspects, an object still possesses a single identity. Different versions for components operate on the same state, at least in part. For instance, a user who thinks of a person as a taxpayer should be able to recognize the adjusted rate of taxation caused by the income being increased by the person's employer. To achieve this, we could describe a version of the write method of **income** that adjusts the rate of taxation as well. The disadvantage of this approach is that if the view **Taxpayer** had been added at some time later we would have had to adapt existing method code requiring understanding it.

It is desirable, therefore, to separate the statements necessary for adjustment of the rate of taxation from those for changing **income**. Generally speaking, it may be necessary in the mapping of a view to assign to a component not just the version of implementation corresponding to the aspect represented by the view, but versions for cooperating aspects as well. With this aim, ARISTARCH/L enables programmers to assign several different versions of implementation to a component in a mapping which are executed in an undetermined order with the same arguments. This is called *combination of versions*. It is possible now to separate the statements necessary for **Taxpayer** from those necessary for **Employee** while at the same time by combination of both versions the rate of taxation gets adjusted whenever an increase of income occurs.

The attribute **income** therefore has two different versions, as shown in Ex. 4.3 (lines 17 to 24). The basic version implements **income** by using a variable which may be changed under the view **Employee**, for instance by the employer. The aspect represented by the view **Taxpayer** cooperates with **Employee** since the rate of taxation depends on the income of the employee. To this end, a method version for writing **income** is defined for **Taxpayer**. By assigning both versions to **income** in the mappings of all views (lines 6, 9, and 12), it is guaranteed that the rate of taxation is adjusted even if the object itself increases income in order to fulfill another task. Nonetheless, as before, other objects may change **income** only via **Employee**. The example also shows how attributes and commands interact. The implementation of **income** by a variable for the aspect represented by **Employee** implicitly defines both method versions for reading and writing, where reading delivers the value most recently written. This makes it possible to cater

```
1  OBJECT
     ...
   IMPLEMENTATION
     MAPPINGS
5      VIEW Person
         income BY BasicVersion, TaxpayerVersion; ...
       END VIEW Person;
       VIEW Employee
         income BY BasicVersion, TaxpayerVersion; ...
10     END VIEW Employee;
       VIEW Taxpayer
         income BY BasicVersion, TaxpayerVersion; ...
       END VIEW Taxpayer

15   ATTRIBUTES
       ...
       ASSERTED income: INTEGER BY
         VERSION BasicVersion BY VARIABLE END VERSION BasicVersion,
         VERSION TaxpayerVersion BY
20         PUT METHOD
             Adjust Rate of Taxation -> taxrate
           END PUT
         END VERSION TaxpayerVersion
       END income;
25     ASSERTED taxrate: INTEGER BY VARIABLE

     COMMANDS
       ...
29 END OBJECT AS Person -> person
```

Ex. 4.3. Cooperation among Aspects

to the aspect of `Taxpayer` by writing a `PUT METHOD` that intercepts any increase of `income` (lines 20 to 22).

One might believe that the undetermined order of execution poses a serious problem. For example, consider a command which changes an employee's benefits, payment policy, and the like at the same time. Each of these modifications necessitates an adjustment of the rate of taxation which may be done by intercepting the command. Unfortunately, the adjustment must be done *after* the changes, a requirement that apparently is in conflict with the undetermined order of execution. The problem disappears by defining versions not intercepting the command itself but rather those commands it calls to do the changes. This is always possible because the methods for both reading and writing an attribute may be customized.

4.4 Derivation of Views

Often an aspect of an object includes another one, that is it constitutes a specialization of another aspect. For instance, we would like to consider an employee a kind of person. To allow the description and exploitation of such relationships, a view may be *derived* from other views. In the course of this, in addition to the components listed in the derived view itself, in general all components from

the interfaces of these basic views are inherited by the derived view, with the exception of those given in its **OMITTING** clause.

Dealing with mappings is more demanding than uniting the interfaces. To the fore comes again the idea of identifiers as specifications of components. The versions of implementation for a component given by the mappings of the basic views, the *basic versions*, are combined. This is reasonable because, in a way, the aspects represented by the basic views cooperate with each other in the aspect represented by the derived view. If the mapping of the derived view gives still other versions for a component, they are regarded as *extensions* of the basic versions specific to the corresponding aspect.

This necessitates a linguistic element to describe a version as an extension of its basic versions[2]. It is sufficient to present such an element for methods since attributes are considered pairs of commands and thus are implemented by pairs of methods. The linguistic element sees a method version merely as a sequence of statements. The extended version is produced by combining the sequences of the basic versions with the sequences of the extending versions. The side effect of the extended version on the state of the object therefore consists of the contributions of both the basic and the extending versions. To control the order of these contributions, it is necessary to specify the point at which to execute the statements of the basic versions with regard to those of the extending versions. On the other hand, the linguistic element has to guarantee that basic versions are executed exactly once, for all paths of the method independent of object state and command arguments. In ARISTARCH/L we achieve this by syntactically grouping two sequences of statements around the point of execution of the basic versions, marked by the keyword **BASE METHOD**. All paths through the first sequence thus converge in this point, diverging again in the second sequence. To indicate the point is mandatory, as the programmer of a version does not know *a priori* if it will serve as an extension later. Regarding the extension of variables, we recall that a variable implicitly defines method versions for reading and writing. We establish that the version for reading ignores the results of the basic versions and returns its own value, whereas the version for writing calls the basic versions and then changes its own value.

Example 4.4 makes use of the fact that both **Employee** and **Taxpayer** are specializations of **Person**. Thus, for instance, **Employee** is derived from **Person** and inherits its general interface (line 11). At the same time the mapping is inherited, with the consequence of the version for the **State** command given by **Employee** implicitly being an extension of the version given by **Person**. While the basic version takes care of printing personal data such as the name, the version for **Employee** confines itself to information such as the personnel number. The body of the extension consists of two sequences of statements surrounding the execution of the basic version, **BASE METHOD**, the first one being empty for this example (lines 47 to 48). Accordingly, the method first prints the personal data and then the employee data. The same holds for the view **Taxpayer**.

[2] In ARISTARCH/L, it is possible to directly describe a version as an extension of other versions without the aid of view derivation but this feature is not described in this paper.

```
OBJECT
  INTERFACES
    VIEW Person
      ATTRIBUTES
        IMMUTABLE name: tString
      COMMANDS
        State ();
        Action ()
    END VIEW Person;

    VIEW Employee DERIVED FROM Person
      ATTRIBUTES
        personnelNumber: INTEGER;
        income: INTEGER
    END VIEW Employee;

    VIEW Taxpayer DERIVED FROM Person
      ATTRIBUTES
        taxNumber: INTEGER;
        IMMUTABLE taxrate: INTEGER
    END VIEW Taxpayer;

    VIEW TaxpayingEmployee DERIVED FROM Employee, Taxpayer
    END VIEW TaxpayingEmployee

  IMPLEMENTATION
    MAPPINGS
      VIEW Person
        income BY BasicVersion, TaxpayerVersion;
        State BY BasicVersion; ...
      END VIEW Person;
      VIEW Employee
        State BY EmployeeVersion; ...
      END VIEW Employee;
      VIEW Taxpayer
        State BY TaxpayerVersion; ...
      END VIEW Taxpayer

    ATTRIBUTES ...
    COMMANDS
      ASSERTED State () BY
        VERSION BasicVersion BY METHOD
          BASE METHOD
          DO Output Personal Data
        END VERSION BasicVersion,
        VERSION EmployeeVersion BY METHOD
          BASE METHOD
          DO Output Employee Data
        END VERSION EmployeeVersion,
        VERSION TaxpayerVersion BY METHOD
          BASE METHOD
          DO Output Taxpayer Data
        END VERSION TaxpayerVersion
      END State;
      ...
END OBJECT AS Person -> person
```

Ex. 4.4. Derivation of Views

So the same basic version of **State** may yield different methods by the use of different extensions without replicating code. The other way around, it is also possible to use the same extension with different basic versions. This may be accomplished by assigning this extension to the corresponding component in different views which are derived from yet other views that employ those different basic versions, respectively.

Please note that it suffices to assign the versions of implementation to the attribute **income** in the mapping of **Person** since it is inherited by the derived views. It is also easy to define a view **TaxpayingEmployee** which combines both **Employee** and **Taxpayer** (lines 23 to 24). The implementations for the components of this view are given by combining and extending the versions of all basic views. This especially is true for the implementation of **State** which now consists of all versions including the basic version, and thus now prints *all* information available for a person.

4.5 Views and Delegation

Delegation is regarded as a means for dealing with very different problems. This polyvalence is the main point of our criticism. Some of these problems are mirrored in the requirements given above, and we shall now consider them in turn in the context of our model.

One of those problems is to describe objects with different aspects. Delegation may be used to represent these aspects by different objects. They operate on a common state, at least in part, by using common prototypes, and actually describe a *single* object. Consistently, Dony, Malenfant, and Cointe [6] call such an object a *split object*. The objects representing different aspects may be compared to views in ARISTARCH/L. Delegation takes on the task of mappings. For instance, **Taxpayer** and **Employee** are given by two objects delegating to a third object which represents the view **Person**. Together they form the split object **person**. All three objects possess their own method versions for **State**. Using the *extended self* (Lieberman [11]) the version corresponding to the current aspect is called independent of the site of the call in one of these objects. For example, the version of **Taxpayer** is used even if **State** was called by **Person** if **Taxpayer** indirectly caused the call by delegating an instruction by another object to **Person**. If several views assign the same version of implementation to a component this may be modelled by an object which holds this common version and is delegated to by the objects representing those views. A serious disadvantage of delegation when used for describing different aspects of an object is that it does not support the notion of split objects as logical entities. It also contradicts the notion of prototypes because not different objects with common behaviour are described but aspects of a single split object. In our model, a reference always denotes the whole object while still indicating the aspect currently used via this reference.

Another disadvantage is that delegation also permits the dynamic extension of split objects with other aspects whereas dynamic changeability of an object in general is completely orthogonal to supporting different aspects of objects. It therefore should be realized by a linguistic element that is orthogonal to

views. Thus delegation in the case of split objects basically meets two different demands. This gives further evidence for the correctness of our opinion that delegation should only serve as a basis for implementing some of the elements of a linguistic model but not as such an element by itself. Supporting objects with different aspects on the one hand and dynamic changeability of objects on the other hand are separate problems and therefore should be dealt with by separate concepts. Szyperski [26] puts it the way that

> "It is a good rule of thumb to keep the number of concepts in a language small. However, it is always possible to reduce everything to (almost) a single concept. For example, in some functional languages everything is expressed using the single concept of higher-order functions. This tends to make programs hard to grasp. Keeping orthogonality and completeness in mind, it is often preferable to provide *separate concepts for separate problems*. This leads to more natural ways of expressing solutions in a given language."

The *single concept* in languages based on delegation is the delegating object.

5 Categorization of Objects

The linguistic elements introduced hitherto support the description of objects one by one. According to the requirements given above an object-based language should permit both concrete and abstract categorization of objects.

In ARISTARCH/L, concrete categorization of an object is handled by *cloning*. For the following the idea of identifiers as specifications of components comes to the fore again. The clone inherits the views of the cloned prototypes, a view of the clone with a given identifier being implicitly derived from all views with the same identifier of these prototypes. With the components of its prototypes, it also inherits the current states. After cloning has taken place both objects are independent of each other.

In class-based languages abstract categorization of objects is dealt with by each object being an *instance* of a *class*. In ARISTARCH/L, objects may have any number of *instance patterns*, or *patterns* for short, which may be instantiated to objects.

5.1 Patterns

Patterns describe sets of objects of the same kind. To this end, they may possess the same elements as do object expressions, that is components and their versions of implementation as well as various views. An object with several different patterns may now be used to create different kinds of objects according to these patterns. Objects with patterns may be actually regarded as classes with several kinds of instances. Although classes are objects there is no infinite meta regress since objects may as well be described directly by using object expressions.

```
1 OBJECT
    INTERFACES
      VIEW
        PATTERNS
5         PersonPattern INTERFACES
            ...
          END PersonPattern
      END VIEW

10  IMPLEMENTATION
      PATTERNS
        ASSERTED PersonPattern IMPLEMENTATION
          ...
        END PersonPattern
15 END OBJECT -> personclass
```

Ex. 5.1. Description of Patterns

```
NEW PersonPattern OF personclass AS Employee -> employee
```

Ex. 5.2. Creation of Instances

The object **person** described in the previous examples becomes a pattern in Ex. 5.1. The **INTERFACES** section of **person** reappears in the **PATTERNS** section of the **INTERFACES** section of the newly created object **personclass** (lines 5 to 7). This is because the definition of a pattern is required in the interface of a view of a class as to instantiate it under this view. The same holds for the **IMPLEMENTATION** section (lines 12 to 14). A pattern may also possess, as other components may, different versions which are used by different views of the class. This may be used to support several versions tuned to different marginal conditions. For example, we may have two different versions of lists which are optimized for fast random access to elements or for fast insertion of elements, respectively.

An arbitrary number of instances of **PersonPattern** may be created now by using the **NEW** expression. If a pattern describes several different views the **AS** clause may be used to determine an initial view. This is shown in Ex. 5.2.

Besides using a **NEW** expression, an instance of a class may be created by using a pattern when describing an object by an object expression. This allows for abstractly categorized objects with additional behaviour not determined by the category. The procedure is the same as for deriving patterns.

5.2 Derivation of Patterns

To describe common properties of patterns, a pattern may be *derived* from other patterns in much the same way views may. In the course of this, in addition to the views listed in the derived pattern itself, in general all views from the basic patterns are inherited by the derived pattern, with the exception of those given in

its **OMITTING** clause. As with cloning, a view of the derived pattern with a given identifier is always implicitly derived from all views with the same identifier of the basic patterns. Therefore, the derivation of patterns is reduced to the derivation of views. Any other views have to be explicitly derived, directly or indirectly, from at least one view of each basic pattern in order to guarantee that a version of implementation is assigned to each inherited component.

The rules given in Sect. 4, for implicitly assigning versions of implementation to components, together with the rule of the above paragraph, regarding views with matching identifiers, allow describing with a minor amount of work constellations in which each pattern has only one anonymous view and for each component there is only one version of implementation. In this case, the rules cause explicit mappings to be superfluous and the views of derived patterns to be implicitly derived from those of their basic patterns. Such constellations are typical of conventional object-oriented languages such as C++. The additional amount of work to be done in ARISTARCH/L to describe them is thus very low.

Different versions of a pattern in connection with derivation yield a controlled form of *or-inheritance* (LaLonde, Thomas, and Pugh [10]). Additionally, similar to attributes and commands, these versions may be extended by the same version used in derived views, respectively. Extension of versions for patterns is reduced to the extension of the components of the pattern according to the mappings of the views. It is not to be mixed up with the derivation of patterns.

The hierarchy of views in Ex. 4.4 may now be used as a starting point for a hierarchy of patterns, as is shown in Ex. 5.3. The different versions of implementation for the components are then distributed among the corresponding patterns so that we may do without explicitly defining them in one pattern. The resulting class possesses several different patterns which can be instantiated separately if they are all listed in the interface of at least one view. **TaxpayingEmployeePattern** describes the complete object **person** also known from the previous examples whereas the other patterns are restricted.

The example also shows the command **Check** in **PersonPattern** (line 23) which is not marked with the keyword **ASSERTED**. A pattern only guarantees the existence of **ASSERTED** components so that the identifier **Check** may not be used in derived patterns such as **TaxpayingEmployeePattern**. But in agreement with the general idea of identifiers as specifications, if a derived pattern defines a version of implementation for **Check** it is considered an extension of the version of **PersonPattern**. Nonetheless, the basic version is hidden from the derived pattern as the latter does not explicitly refer to its identifier and the extension remains correct even if the basic version disappears. In our example, the version of **EmployeePattern** for **Check** implicitly extends the basic version defined in **PersonPattern** (line 27).

5.3 Instances and Class Components

Components of a class at the same time are considered common to all instances of that class. This especially means that an instance may implicitly access the state of its class. In ARISTARCH/L, this essentially works by thinking of patterns

```
1 OBJECT
    INTERFACES
      VIEW
        PATTERNS
5         PersonPattern INTERFACES
            VIEW Person ...
          END PersonPattern;
          EmployeePattern DERIVED FROM PersonPattern INTERFACES
            VIEW Employee DERIVED FROM Person ...
10        END EmployeePattern;
          TaxpayerPattern DERIVED FROM PersonPattern INTERFACES
            VIEW Taxpayer DERIVED FROM Person ...
          END TaxpayerPattern;
          TaxpayingEmployeePattern DERIVED FROM EmployeePattern, TaxpayerPattern
15        END TaxpayingEmployeePattern
      END VIEW

    IMPLEMENTATION
        PATTERNS
20        ASSERTED PersonPattern IMPLEMENTATION ...
              ASSERTED State () BY METHOD ... Basic Version;
              ASSERTED Action () BY METHOD ... ;
              Check () BY METHOD ... Basic Version
          END PersonPattern;
25        ASSERTED EmployeePattern IMPLEMENTATION ...
              ASSERTED State () BY METHOD ... Employee Version;
              Check () BY METHOD ... Employee Version
          END EmployeePattern;
          ASSERTED TaxpayerPattern IMPLEMENTATION ...
30            ASSERTED State () BY METHOD ... Taxpayer Version;
              Check () BY METHOD ... Taxpayer Version
          END TaxpayerPattern
33  END OBJECT -> personclass
```

Ex. 5.3. Derivation of Patterns

as being derived from the class. The difference to derivation from another pattern is twofold. First, components of a class exist only once. Second, neither are views of the class inherited by the pattern nor are interfaces of class views inherited by derived views of the pattern. This has the same effect as listing the class views and the components in interfaces in the **OMITTING** clauses of the pattern and the derived views of the pattern, respectively. Therefore, the version of implementation for a class component used by an instance is determined by the current view of the instance. For this reason, each view of a pattern has to be derived, directly or indirectly, from at least one view of the class.

5.4 Nested Patterns

ARISTARCH/L supports classes of instances which are classes themselves, by the means of *nested* patterns. Nested patterns may be derived from other patterns nested inside the same pattern. But they are also inherited by derived patterns like other components. This enables the programmer to derive nested

```
OBJECT
  INTERFACES
    VIEW
      PATTERNS
        StringClassPattern INTERFACES
          VIEW
            PATTERNS
              StringPattern ...
          END VIEW
        END StringClassPattern
    END VIEW

  IMPLEMENTATION
    PATTERNS
      ASSERTED StringClassPattern IMPLEMENTATION
        ATTRIBUTES
          ASSERTED Sequences: tTable
        PATTERNS
          ASSERTED StringPattern ...
      END StringClassPattern
END OBJECT
```

Ex. 5.4. Independent String Classes by Means of Nested Patterns

patterns from patterns nested in a basic pattern as well. It is therefore possible to abstractly categorize classes and thus hierarchies of patterns.

By the means of nested patterns we are now able to solve the problem of describing multiple classes of character strings with different tables of character sequences, as stated in Sect. 2.3. A string class contains the table of different sequences as part of the state common to all strings. In Ex. 5.4, these classes are again abstractly categorized by an object with a corresponding pattern. By instantiating the pattern of this object we get string classes with independent tables of sequences which in turn may be used to create strings implicitly accessing the same table.

Another example shall merely be outlined in the following since it is too complex to be explained in detail here. Access objects for a file may be regarded as instances of this file which are able to implicitly access its content. They permit stateful access of the file, for example, depending on the context of use of the file, to the current record by retaining a record index. Opening such a file then means to create an access object from a pattern contained in the file. There may be very different kinds of files in a file system, for instance sequential files, indexed-sequential files, and direct files. Indexed-sequential files may be thought of as specializations of sequential files, and thus the corresponding pattern may be derived from that for sequential files. All patterns together may be derived from a generic file pattern which, among other components, contains a universal pattern for access objects which is inherited by the other kinds of files. All files on a file system are considered instances of the file system which implicitly have access to its management information. What we have now is an object representing a file system which contains a set of patterns describing various

sorts of files which in turn contain patterns describing access objects. Since in general a computer system has several storage devices containing different file systems it is reasonable to finally describe a class of file systems. In this manner, the result is an object with nested patterns of *the first, second, and third order*.

This example clearly explains the way nested patterns extend the expressiveness of previous, object-oriented languages. Each call of a program written in such a language actually creates anew the classes contained in the program with a fresh state. Such programs may be thought of as classes of the second order, that is as objects with patterns which themselves contain other patterns. Unlike nested patterns, however, the conventional model allows neither the derivation of classes of the second order nor describing classes of higher orders.

5.5 Patterns and Delegation

Lieberman [11] proposes delegation as a means for the abstract categorization of objects. Since delegation always refers to another concrete object, *a priori* the entire behaviour of an object is determined by the abstract category. Most of the time this is not what is needed, so the behaviour of the object not part of the abstract category has to be explicitly cancelled or redefined. For example, Dony, Malenfant, and Cointe [6] correctly point out that, corresponding to a behaviour pattern, in most cases, abstract categorization relates only to the component structure of objects whereas their states shall remain independent. Therefore, when using delegation the object has to implement by itself all components carrying state information. The advantage of having a template is thus lost. In the prototype-based language SELF (Ungar and Smith [27]) *trait objects* are used to describe commands common to a set of objects. An object with attributes carrying state delegates its commands to such a trait object. Other objects are created by *cloning* this object, thus using it as a template. Because of this, they possess their own state but delegate to the same trait object. This way of using delegation is a contradiction to the notion of prototypes as concrete objects that also serve as templates because trait objects are stateless and abstract. They just serve as repositories for common commands and therefore may also be treated as optimizations not visible to the user. An additional disadvantage of delegation is the lack of support for concrete categorization.

Delegation may be used as well if access to a common state is the decisive factor for categorization of a group of objects. Lieberman for example describes a set of pens with the same vertical but different horizontal positions. To achieve this, each pen with one exception possesses a variable for the horizontal position only and delegates to the single pen with both coordinates. For instance, we might think of an electrocardiograph here, in which the vertical position common to all pens is given by the position of a carriage relative to a strip of paper. As the actual reason for the common state, an object such as this carriage always exists. It is this carriage that really possesses the part of the state common to all pens. Consequently, in our opinion the implicit access to the common part of the state should not be done by means of delegation, as its carrier may not be considered a prototype of the delegating objects. In our model, the carrier

rather is the class of the objects, as classes may also be abstractly categorized. In this example, the carriage would serve as the class of those pens mounted on it. Nested patterns permit to describe a class of carriages, and thus a class of electrocardiographs.

We would like to emphasize that delegation is neither supported by our model in the form of inheritance between classes (Stein [24]). This is an immediate consequence of our criticism of delegation on the level of objects and the notion of classes as objects with patterns. In contrast to conventional object-oriented languages a class, in our terminology an object with a pattern, may not be derived from another class. Nevertheless, categorization of classes is straightforward. If we want to describe a class whose instances are specializations of the instances of another class this corresponds to the conventional notion of a subclass and a superclass. According to the difference between concrete and abstract categorization, our model, besides a tight connection, permits to describe independence between these two classes. In the first case, superclass and subclass are instances of an object with two corresponding patterns, the second derived from the first, respectively. Both patterns contain patterns themselves, the pattern in the second one derived from the pattern in the first one. As a result, if the pattern nested in the first one is modified the pattern contained in the second one changes as well. Nevertheless, in contrast to previous models, the states of these two classes are independent of each other. The second case is given if a programmer wants to reuse a hierarchy of patterns but is neither interested in later changes nor wants to have his or her own modifications taken over into the existing hierarchy. He or she will then provide for this by cloning the superclass and changing the clone to become the subclass.

6 Related Work

6.1 Views

Hailpern and Nguyen [7] propose a model in which the method to be executed not only depends on the target object but also on its client. Consequently, the model really supports views with different versions of implementation for components. The notion of views as such is not present, however. In addition, the model does not relate to the current view whenever an object calls itself. Hailpern and Ossher [8] describe a model with views but it does not permit different versions.

Shilling and Sweeney [23] develop a model that provides views with different method versions for commands. In contrast to our model however, it does not allow different versions for attributes and patterns. In Shilling's und Sweeney's model views may possess their own attributes and may be *multiply activated*. Each activation has its own variables for attributes of the view but may also share variables with other activations. Views therefore do not model aspects but rather different *roles* an object may *play* in a given context. The relationship between the object and the activations of its views is very similar to that between a class and its instances. We believe that aspects and roles are orthogonal concepts

and thus should be catered to by separate linguistic elements. Richardson and Schwarz [21] model roles as well, despite calling the linguistic elements *aspects*. Basically, aspects are objects with a pointer to a *base object* the components not implemented by the aspect are forwarded to. The aspect only supports those components of the base object that are explicitly exported, which is a shortened form of using the components by means of the base pointer. Pernici [20] describes a similar model for the specification of objects in the area of office information systems which is primarily aimed at supporting object reuse.

Wieringa, de Jonge, and Spruit [28] distinguish between aspects and roles of an object, like we do. In their model, different aspects of an object are described by means of *dynamic subclasses*. In contrast to conventional or *static* subclasses an object may change its membership to a dynamic subclass over time while its identity remains the same. However, an application may not select a *current* dynamic subclass as it may with views. This is only possible with roles which are described by *role classes*. An object may play a role as given by one role class several times. As with representing roles by instances of patterns of the object in our model, the identity of the object may be pushed to the background in favour of the identities of the roles.

Harrison and Ossher [9] argue that by looking at objects from different aspects the notion of object gets a distributed character. It is their goal to develop a model with the following two features. First, it should make possible the development of applications depending on different aspects of an object separately while at the same time allowing for their cooperation. In particular, it should facilitate applications initially not provided for, without rendering already existing objects and applications useless or subjects to modification and recompilation. Second, each application should be able to utilize separately encapsulation, polymorphism, and inheritance. Particularly, it should enable all applications to classify independently objects according to the corresponding aspects. Harrison and Ossher propose several heuristics to establish automatically the relationships among classes describing different aspects of the same object, that are necessary for different applications to interact with each other. In our opinion this approach has a serious disadvantage. For all aspects of a single object to cooperate, the programmer of an application, and thus of a certain aspect, has to either anticipate other possible aspects of the object to allow for later cooperation by defining appropriate components, or announce modifications to other applications for the appropriate change of corresponding classes. It seems to be unfeasible to separate different aspects to the extent intended by the authors. We have consciously decided to describe different aspects by views inside a single object description. Nevertheless, ARISTARCH/L allows to list several versions of implementation for a component corresponding to several cooperating aspects in a view mapping, thus achieving a degree of separation among aspects that depends on the given situation.

FLAVORS [19], COMMONLOOPS [4], and the COMMON LISP OBJECT SYSTEM (CLOS) [5] try to avoid multiple execution of the same method in the face of multiple inheritance by means of *method combinations*. CLOS, for instance, offers a *standard method combination* which distinguishes between *primary methods* and

auxiliary methods. *Auxiliary methods* in turn are split up into `:before`, `:after`, and `:around` methods. Whenever a command, or *generic function*, is called on an object, first of all the nested `:around` methods are called which in turn call the `:before` methods, the *primary methods*, and finally the `:after` methods. The sequence of execution is determined by topologically sorting the inheritance graph. Unfortunately, the standard method combination is insufficient, for instance, if one wants a certain method in any case to be executed before all others. To this end, CLOS supports the definition of customized method combinations. It has turned out, however, that this feature is used seldom even by experienced programmers because of its high complexity (Winston and Horn [29]). We restrict ourselves to an approach that corresponds to *primary methods*.

6.2 Delegation and Inheritance

Delegation is described with extensive examples by Lieberman [11]. Lieberman also explains the philosophical background of delegation and claims that delegation is more expressive than inheritance. Many of his examples are picked up in later papers by other researchers to describe problems with delegation and argue for alternative models.

Such an alternative model is presented by Stein [24]. Stein shows that delegation and inheritance have the same expressiveness, and thus Lieberman's claim is wrong, because inheritance is delegation on the level of classes. Our model differs insofar as we refuse delegation both on the level of objects and as inheritance on the level of classes. Specialization takes place by changing clones and deriving patterns. The proposal to reduce the exact guarantee of structure by a class to a minimal guarantee, that is to describe additional behaviour for an abstractly categorized object, is also by Stein.

In the *Treaty of Orlando* [12, 25] Lieberman, Stein, and Ungar, one of the developers of SELF, agree on the fact that a model for sharing of common behaviour should allow for static as well as dynamic and implicit as well as explicit sharing while applying to both single objects and groups of objects. In contrast to our model, their classification scheme is oriented to the mechanical features rather than the purpose of the model.

Ungar and Smith [27] describe the prototype-based language SELF. SELF uses delegation for abstract categorization by factorizing common components of objects into *trait objects* and solving the problem of independent object states by cloning single objects. We use cloning of several objects at the same time for concrete categorization and patterns for abstract categorization. We regard the combination of trait objects and delegation as an optimization by the language system.

Other researchers try to solve the problem of unconstrained generality of delegation by using a rule-based approach. Minsky and Rozenshtein [17] describe their notion of *law-governed object-oriented systems*. In such a system messages between objects may be modified, blocked, or delivered to a different target object by means of rules. It does not support delegation directly but may emulate it by appropriate rules. Minsky and Rozenshtein [18] show how their approach

may be employed to constrain the use of delegation. A similar approach is taken by Almarode [2] with *rule-based delegation*. But like Minsky and Rozenshtein, Almarode uses delegation in various ways that are incompatible with the notion of prototype-based programming. For instance, class-based systems are modelled by delegation and appropriate rules. Almarode uses delegation even to forward messages in a *part-of* hierarchy. In this case the *extended self* is unnecessary which is a sure sign that this use of delegation is not adequate. We believe that rule-based approaches are not appropriate for general use because they are too complicated to apply. ARISTARCH/L tries to be straightforward to use by restricting itself to a concrete model of sharing. On the other hand, rule-based approaches surely are suitable for experimenting with different models.

The proposal of *composition filters* by Akşit, Bergmans, and Vural [1] is similar to *law-governed object-oriented systems*. A set of filters accepts messages on behalf of an object and under conditions given by *filter elements* passes them on, for instance to the object itself or to one of its *interface objects*. Interface objects are encapsulated inside the object or given by reference. Different aspects of an object may be modelled by a composition filter with one filter element for each aspect. It is not possible, however, to relate to the current view whenever the object calls itself. By means of the pseudo variable **server** which refers to the used object, delegation may be modelled with interface objects given by reference, that is it is possible to alternatively represent views by delegating objects. Therefore, our arguments against delegation as a linguistic element by itself as well as rule-based approaches are also valid in the case of composition filters.

Dony, Malenfant, and Cointe [6] try to extract features for a language most suitable for prototype-based programming by means of a taxonomy of existing prototype-based languages. They describe different problems of delegation, among them the problem of independent object states and the lack of support for *split objects*. Further, they discover that the description of a guaranteed common component structure of a set of objects, as it is possible with the aid of classes, can be achieved neither by cloning nor by delegation. We support split objects by views and by implicit access to components of a class by its instances. The necessity of independent objects as well as structural guarantee leads us to offer both cloning and patterns in ARISTARCH/L. Delegation is not supported as we believe it to be suitable as a basis for implementing some of the linguistic elements of a model of sharing but not as such an element by itself.

7 Conclusion

We have presented an object model which meets the requirements of a system with a uniform object space. The model permits the description of objects without classes using object expressions, as well as the modification of existing objects to preserve information. Different aspects of objects can be described independently by employing different views and different versions of implementation for components. The model supports concrete and abstract categorization

as well as the extension of abstractly categorized objects by cloning, patterns, and object expressions which may refer to patterns, respectively. Classes with different states but instances sharing the same behaviour pattern are expressable by applying nested patterns. Through these features, our model supports new ways of sharing common behaviour and behaviour patterns among objects without using delegation at the language level, either on the level of objects or as inheritance on the level of classes.

References

1. M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In: Madsen [13], pp. 372–395.
2. J. Almarode. Rule-based delegation for prototypes. In: Meyrowitz [16], pp. 363–370.
3. A. P. Black, N. C. Hutchinson, E. Jul, H. M. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, **SE-13**(1):65–76, January 1987.
4. D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In: Meyrowitz [14], pp. 17–29.
5. L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In: J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, eds., *Proceedings of the European Conference on Object-Oriented Programming* (Paris, France, June 15–17, 1987), vol. 276 of *Lecture Notes in Computer Science*, pp. 151–170. Springer-Verlag, Berlin, 1987.
6. C. Dony, J. Malenfant, and P. Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In: A. Paepcke, ed., *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, British Columbia, Canada, October 18–22, 1992), *ACM SIGPLAN Notices*, **27**(10):201–217, October 1992. ACM.
7. B. Hailpern and V. Nguyen. A model for object-based inheritance. In: B. Shriver and P. Wegner, eds., *Research Directions in Object-Oriented Programming*, Series in Computer Systems, pp. 147–164. MIT Press, Cambridge, Massachusetts, 1987.
8. B. Hailpern and H. Ossher. Extending objects to support multiple interfaces and access control. *IEEE Transactions on Software Engineering*, **16**(11):1247–1257, November 1990.
9. W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In: A. Paepcke, ed., *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Washington, Washington, D.C., September 26–October 1, 1993), *ACM SIGPLAN Notices*, **28**(10):411–428, October 1993. ACM.
10. W. R. LaLonde, D. A. Thomas, and J. R. Pugh. An exemplar based Smalltalk. In: Meyrowitz [14], pp. 322–330.
11. H. Lieberman. Using prototypical objects to implement shared behaviour in object-oriented systems. In: Meyrowitz [14], pp. 214–223.
12. H. Lieberman, L. A. Stein, and D. M. Ungar. Of types and prototypes: The treaty of Orlando. In: L. Power and Z. Weiss, eds., *Addendum to the Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Orlando,

Florida, October 4–8, 1987), *ACM SIGPLAN Notices*, **23**(5):43–44, May 1988. ACM.

13. O. L. MADSEN, ED. *Proceedings of the European Conference on Object-Oriented Programming* (Utrecht, Netherlands, June 29–July 3, 1992), vol. 615 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
14. N. MEYROWITZ, ED. *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, September 29–October 2, 1986), *ACM SIGPLAN Notices*, **21**(11), November 1986. ACM.
15. N. MEYROWITZ, ED. *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Orlando, Florida, October 4–8, 1987), *ACM SIGPLAN Notices*, **22**(12), December 1987. ACM.
16. N. MEYROWITZ, ED. *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, October 1–6, 1989), *ACM SIGPLAN Notices*, **24**(10), October 1989. ACM.
17. N. H. MINSKY AND D. ROZENSHTEIN. A law-based approach to object-oriented programming. In: Meyrowitz [15], pp. 482–493.
18. N. H. MINSKY AND D. ROZENSHTEIN. Controllable delegation: An exercise in law-governed systems. In: Meyrowitz [16], pp. 371–380.
19. D. A. MOON. Object-oriented programming with FLAVORS. In: Meyrowitz [14], pp. 1–8.
20. B. PERNICI. Objects with roles. In: F. H. Lochovsky and R. B. Allen, eds., *Proceedings of the Conference on Office Information Systems* (Cambridge, Massachusetts, April 25–27, 1990), *ACM SIGOIS Bulletin*, **11**(2/3):205–215, April/July 1990. ACM.
21. J. RICHARDSON AND P. SCHWARZ. Aspects: Extending objects to support multiple, independent roles. In: J. Clifford and R. King, eds., *Proceedings of the International Conference on Management of Data* (Denver, Colorado, May 29–31, 1991), *ACM SIGMOD Record*, **20**(2):298–307, June 1991. ACM.
22. C. SCHAFFERT, T. COOPER, B. BULLIS, M. KILIAN, AND C. WILPOLT. An introduction to TRELLIS/OWL. In: Meyrowitz [14], pp. 9–16.
23. J. J. SHILLING AND P. F. SWEENEY. Three steps to views: Extending the object-oriented paradigm. In: Meyrowitz [16], pp. 353–361.
24. L. A. STEIN. Delegation is inheritance. In: Meyrowitz [15], pp. 138–146.
25. L. A. STEIN, H. LIEBERMAN, AND D. M. UNGAR. A shared view of sharing: The treaty of Orlando. In: W. Kim and F. H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*, ACM Press Frontier Series, pp. 31–48. Addison-Wesley, Reading, Massachusetts, 1989.
26. C. A. SZYPERSKI. Import is not inheritance – why we need both: Modules and classes. In: Madsen [13], pp. 19–32.
27. D. M. UNGAR AND R. B. SMITH. SELF: The power of simplicity. In: Meyrowitz [15], pp. 227–242.
28. R. WIERINGA, W. D. JONGE, AND P. SPRUIT. Roles and dynamic subclasses: A modal logic approach. In: M. Tokoro and R. Pareschi, eds., *Proceedings of the European Conference on Object-Oriented Programming* (Bologna, Italy, July 4–8, 1994), vol. 821 of *Lecture Notes in Computer Science*, pp. 32–59. Springer-Verlag, Berlin, 1994.
29. P. H. WINSTON AND B. K. P. HORN. LISP, p. 509. Addison-Wesley, Reading, Massachusetts, third edition, 1989.