Open Issues in Object-Oriented Programming

A Scandinavian Perspective

OLE LEHRMANN MADSEN

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

(e-mail: olmadsen@daimi.aau.dk)

SUMMARY

We discuss a number of open issues within object-oriented programming. The central mechanisms of object-oriented programming appeared with Simula developed more than 30 years ago including class, subclass, virtual function, active object and the first application framework, `Class Simulation`. The core parts of object-oriented programming should be well understood, but there are still a large number of issues where there is no consensus.

The term object-orientation has been applied to many subjects such as: analysis, design implementation, data modeling in databases, and distribution. In this paper the term object-oriented programming is meant to cover all these subjects, since one of the advantages of object-orientation is that it provides a unified approach to these subjects.

The issues being discussed in this paper are: modeling versus reuse as the main benefit of object-orientation; the need for a language independent conceptual framework; abstraction mechanisms for supporting object-oriented programming including classification and composition, single and multiple inheritance, inner versus super, genericity versus virtual classes and issues related to typing; class-based versus prototype-based languages; and concurrency.

The point of views presented in this paper reflects the authors background in the Scandinavian tradition as manifested in the BETA language and the Mjølner project.

KEY WORDS   Languages, object-orientation, modeling, reuse, conceptual framework, abstraction mechanisms, class-based languages, prototype-based languages, concurrency.

1.   Introduction

Although the history of object-oriented programming goes back more then 30 years to the development of the Simula languages, there are still a number of issues that may be considered open, in the sense that there is no consensus about them in the object-oriented programming community. Open is a relative term here: there are issues where everybody agrees, some where most people agree, and some where there is a lot of disagreement. There is also a difference in what is considered open in the research community and what is considered open among practitioners.

The point of view presented in this paper reflects the author's background in the Scandinavian tradition of object-oriented programming. This tradition started with the development of the Simula languages in the early sixties by Ole-Johan Dahl and Kristen Nygaard. Simula was originally developed as a simulation language (Simula I [DN66]), but it was soon realised that the principles behind Simula I could be used for programming in general, leading to the development of Simula 67 [DNM68]. An account of the history of Simula may be found in Reference [DN81] and an introduction to Simula may be found in Reference [Mag94a]. The BETA language [KMMN83a, MMN93] has been developed in

Scandinavia as a successor to Simula. Another Scandinavian project that has influenced this work, is the Mjølner project [KLMM94] in which tools for supporting object-oriented software development have been developed.

In this paper the term object-oriented programming is used in a broad sense to cover what other authors designate object-oriented analysis, design, implementation, data modeling in databases, and distributed programming. The reason for this broad definition is that modeling and design have always been considered central elements in the Scandinavian school of object-orientation. Modeling is about creating a description of phenomena and concepts from a given application domain and design is about creating new phenomena (objects) and concepts (classes) and representing these in a program. These processes cover what is often called object-oriented analysis and design in popular text books [CY89, Boo91, RBPL91]. As mentioned later in this article, there is, however, more to analysis and design than creating object and class diagrams.

Some of the issues being discussed in this paper are not considered open in the Scandinavian tradition, but are mentioned since they have not (yet) been adopted in general despite the fact that they have proven their usefulness during more than 30 years of experience with Simula and BETA. Other issues being discussed are considered open in general.

The issues discussed in this paper are:

- **Modeling versus reuse.** During the last 10 years a change of focus seems to have happened in the object-oriented community. From being mainly interested in reuse of code, the focus has shifted to analysis and design. We argue that there should be a balance between these extremes to fully realize the benefits of the potential for object-orientation to integrate analysis, design and implementation.

- **Explicit conceptual framework.** The conceptual framework underlying object-oriented modeling should be formulated independently of the languages. This separation is necessary to avoid limiting the developer by the (current) expressive power of the languages and to produce new requirements for language design.

- **Abstraction mechanisms.** Although there is some agreement about the core abstraction mechanisms, there are still a number of differences between the common languages. We discuss alternative mechanisms for supporting classification and composition, single and multiple inheritance, inner versus super, genericity versus virtual classes, encapsulation, dynamic versus static typing, contravariance versus covariance, and types versus classes.

- **Class-based versus prototype-based languages.** Prototype-based languages are one of the most interesting developments within object-orientation. Prototype-based languages make it possible to program without classes, but they do in general not support programming using a class-like style. We argue that the class-like style should be supported and that it would be desirable to unify the two approaches.

- **Concurrency.** Simula includes a notion of quasi-parallel processes in the form of active objects that are used for representing concurrent processes. They have also been the basis for the design of concurrency in BETA. No other object-oriented language seem to have adapted the Simula notion of active object. There are a number of

proposals for concurrency in object-oriented languages, but no single model has been widely accepted. We argue that the Simula approach is (still) considered a good approach to concurrency.

The rest of this paper consists of a section corresponding to each item in the above list of issues, followed by a conclusion. Due to the Simula/BETA background of the author, there will often be arguments in favour of the approaches taken in BETA, which is a statically-typed[1] class-based object-oriented language. Some knowledge about BETA is assumed, although we have tried to limit this as much as possible. An introduction to BETA may be found in Reference [Mad94].

2. Modeling versus reuse

In the Scandinavian tradition, the focus has always been on modeling and design. Simula was originally developed as a simulation language and in simulation it is essential to be able to model processes from the real world. Thus, the development of language constructs for representing real world phenomena and concepts has been central in the design of Simula and BETA. Efficient implementation and software engineering techniques were also important. In comparison with other language projects from the sixties and seventies, modeling and design seem to have been more important in the Simula and BETA development.

It is well known that the initial breakthrough for object-orientation came with Smalltalk [GR83] and the industrial breakthrough with C++ [Str86]. Since then object-orientation has entered the area of data bases and methodologies. In the same period there seem to have been a major change of focus:

2.1 Focus on code reuse

In the beginning when object-oriented programming was becoming popular, there was a major interest in reuse of code. The vast majority of people advocating and practising object-oriented programming claimed that the main benefit of object-orientation was the great possibilities of reusing code. It is true that object-oriented programming languages provided new possibilities for reusing code compared to traditional programming languages. For instance the class-, subclass- and virtual mechanisms provide excellent support for factoring code and thereby reuse of code.

While code reuse is certainly an important aspect of object-oriented programming too much emphasis on this had the consequence that little attention was paid to modeling and design. The common rule seemed to be that once a class was written, one should never need to modify it. New software was developed by creating new subclasses that inherited from one or more other classes and redefining the stuff that needed to be different. Since reuse was in focus, there was a lot of interests in multiple inheritance and mixins, as these features offer good possibilities for the programmer to reuse bits and pieces from different classes. A class hierarchy developed solely for the purpose of reuse may result in an inheritance

---

[1]Like Simula, and Eiffel, BETA uses a limited amount of run-time type checking, i.e., it is not completely statically typed. This is discussed in section 4.5.2, but for simplicity we refer to BETA as a statically-typed language.

structure that has no conceptual relationships between the classes, but instead reflect their development over time. Such class hierarchies are difficult to use and maintain.

From the Scandinavian perspective, the use of subclassing to reuse code was quite strange, since subclasses were originally intended to model a specialisation of concepts. This form of subclassing is called *incidental inheritance* [Sak89]. Use of incidental inheritance may be useful when developing an initial prototype of a system, but the class-structure and classes should be redone until they reflect a proper conceptual understanding of the domain, since they will then be easier to reuse. Reuse is a side-effect of good abstractions.

2.2   Focus on analysis and design

In the late eighties, a number of object-oriented methodologies were proposed, reflecting that modeling and design was gaining importance. We consider this development positive, since it encourages programmers to think about their designs before implementing tricky inheritance relations. In analysis and design it is obvious that subclassing should be used for representing specialisation of concepts instead of incidental inheritance.

The methodologies do, however, seem to introduce some new problems to object-orientation:

- Their importance are often overestimated. A lot of effort seems to be used in drawing boxes and bubbles with little semantic meaning and often no code is ever produced.

- New analysis and design languages are proposed. These languages are mainly graphical notations for expressing object-oriented language constructs like class, subclass, virtual procedure, etc. These new languages are often supported by CASE tools that support construction of diagrams and code generation. As discussed in section 2.3 this has reintroduced the traditional problems with CASE gap and reverse engineering.

The real benefit of object-orientation compared to traditional structured analysis and design (SA/SD) is that object-orientation provides a unifying perspective on analysis, design and implementation. This unification also includes data modeling in databases, since object-oriented databases provides much richer modeling capabilities than relational databases.

Traditional SA/SD has never been used in the Simula/BETA community and probably the same can be said about most Smalltalk programmers. It has always been considered something that was needed when programming in COBOL, since there was a need for something at a more abstract level. In contrast, Simula/BETA have been used for design since they were designed to be able to represent real world phenomena and concepts.

It has always been difficult to teach object-orientation to people working with traditional SA/SD and Cobol, since they have insisted on having more than 'just a programming language' for design and they did not accept object-orientation as useful in design. It helped when Yourdon together with Coad published his book on OOA [CY89]. This immediately made it much easier to advocate object-orientation.

The problem now seems that there is too much focus on analysis and design. In addition people that used to be engaged in SA/SD methods are introducing problems in object-orientation that did not use to be there. In SA/SD, the programming language has not been considered important partly because it is usually at a much lower level than the languages used for analysis and design. When converting to object-orientation, many methodology

people continue to ignore the importance of the programming language. It is, however, the power of the programming languages that have made object-orientation so successful.

2.3 Object-orientation as an integrating perspective

One of the advantages of object-orientation is that it provides an integrated perspective on analysis, design, and implementation, including data modeling in databases. Compared to methodologies like structured analysis and design, it is possible to use the same concepts and languages during analysis, design and implementation. The same underlying conceptual framework including notions such as classification, generalisation/specialisation, and composition may be applied to analysis, design and implementation. Similarly language constructs such as class, subclass, (virtual) procedure attribute, part object and reference may be used in analysis, design and implementation. One of the main goals in the development of the Mjølner BETA System has been to to support use of the same language for design and implementation.

In practice, however, the current methodologies [CY89, Boo91, RBPL91] and programming languages do not fully obtain this integration. Most analysis and design methodologies propose their own graphical notation for constructs like class, subclass, etc. It is then assumed that these diagrams are mapped into a programming language like C++. In order to automate code generation, most analysis and design notations are supported by one or more CASE tools that can translate them to code.

For traditional SA/SD methodologies it is well know that mapping between different representations cause problems. Two such, the CASE gap and the reverse engineering problem have been identified. For object-oriented methods such problems *should* not exist. However, by using different languages for analysis and design versus implementation the CASE gap and reverse engineering problems have been re-introduced in the object-oriented context.

Almost all authors describing object-oriented methods recommend an evolutionary approach in which the developers alternate between analysis, design and implementation. This alternation makes CASE gap and reverse engineering even worse than for traditional waterfall methods. The transitions between these phases should be easy and supported by tools. If, however, different notations are used, it is difficult to keep all the representations consistent and up-to-date, just as there is a mental overhead in dealing with several models.

One may question whether it is possible to use the same language for analysis, design and implementation in the sense of this being applied by the main stream methodologies. We claim that this is indeed possible. Most programming languages already include the constructs that are found in the analysis and design notations. Hence, a subset of most object-oriented languages can be used in the analysis and design phases. As mentioned in the introduction to this paper, we consider analysis, design and implementation as programming at different abstraction levels. Thus we claim that it is far more productive to use the same language in all three phases instead of shifting between different languages. It of course puts some additional requirements on a programming language that it should also be suited for analysis and design. In analysis and design languages one should not be forced to deal with low level programming language issues. BETA has explicitly been designed to support design as well as implementation.

BETA may also to some extent be used to support analysis, but there are aspects of analysis that are not well supported by a formal language notation. During analysis, one should not be limited by the expressive power of a formal language notation. There is a need to make informal descriptions, and examples. Moreover, as mentioned in the next section, it may also be useful to be able to use prototypical concepts[2] that are not easily represented in any existing object-oriented language. However, the object-oriented analysis presented in e.g. Reference [CY89] uses a formal notation for expressing the result of an analysis, and for this purpose BETA might be used. For analysis Reference [CY89] has the same weaknesses as BETA.

For static aspects such as classification and composition structures, there is no reason to use different languages for analysis, design and implementation. It is more open whether or not this is possible with respect to dynamic aspects. The various methodologies provide different alternatives like state-transition-diagrams, and Petri Nets. It is not obvious how to map between such diagrams and programming languages. One problem in particular is that few of the popular programming languages supports concurrency. However, for BETA the dynamic aspects may be described using the concurrent parts of BETA.

One difference between analysis/design languages and implementation languages is that the former usually have a graphical syntax and the latter a textual syntax. Many designers prefer a graphical syntax for analysis/design since it is useful for providing an overview of a design and thereby easier to comprehend. It is the other way around for implementation where most implementers prefer a textual notation, since it is more compact than a graphical notation. There is, however, no general consensus on this. In analysis/design it may only be in the initial phases that a graphical design is preferably, since when the designers become familiar with the design, a compact textual description may be more manageable than a large number of drawings. For implementation there may be people who prefer a graphical language to a textual language. The ideal solution seems to be that the programmer should be able to alternate between a graphical notation and a textual notation no matter whether he/she is doing analysis, design or implementation.

For BETA a graphical syntax has been developed to supplement the original textual syntax. The graphical syntax covers patterns[3], subpatterns, part objects, references, etc., and corresponds to the analysis and design notation found in popular analysis and design methods [CY89, Boo91, RBP+91]. The Mjølner BETA System includes a structure editor, called Sif [ES94a], which is an integrated structure- and text-editor with support for abstract presentation and browsing. In addition the system includes a CASE tool, called Freja [ES94b], supporting the graphical syntax. The two editors are integrated: it is possible to alternate freely between the graphical and textual syntax. This alternation is possible because there is a one to one correspondence between the graphical syntax and the textual syntax. A BETA program is represented in the form of an abstract syntax tree (AST). The textual syntax is one presentation of the AST and the graphical syntax is another. When a

---

[2]Prototypical concepts will be introduced in section 3 and should not be confused with prototype-based languages.

[3]A BETA pattern is a unification of abstraction mechanisms such as class, procedure, function, process type, exception type, etc. Examples of patterns will be given in section 4 and later sections.

change is made to either of the presentations, the changes are reflected in the AST, and the other presentation is immediately updated.

Because the user may freely alternate between the two forms of presentation, the editors are well suited for supporting an evolutionary approach to software development. In the analysis and design phases, the graphical syntax may be used. Since the textual syntax is immediately available, the transition to implementation is immediately possible. During the implementation phases, any changes to the class structure are immediately reflected in the graphical notation, i.e. the transition back to analysis and design is also immediately possible. I.e there is no CASE gap and reverse engineering is directly supported.

On the other hand, the user is not limited to use only the graphical notation during analysis and design, but may use the textual notation as well. Since the Sif editor also supports abstract presentations of a program, it is possible to present a program at the same abstract level as the design diagrams.

To obtain the full benefit of object-orientation, we think that the same language should be used for (analysis), design ,and implementation. Above we described how Sif and Freja supports this for BETA. Current analysis and design languages are primarily a graphical syntax for constructs found in most programming languages, and by supplying a graphical as well as a textual syntax and supporting this with integrated editors like Sif and Freja, we have demonstrated one way of improving the benefits of object-orientation.

3.   Explicit conceptual framework

It should be clear from the above that modeling and design is considered an important aspect of object-orientation. In this section we shall briefly summaries a conceptual framework for object-oriented programming[4]. The purpose of this summary is to be able to discuss language support for modeling, integration of analysis, design and implementation. There may be a distinction between the conceptual framework underlying modeling in class-based languages and the one underlying modeling in prototype-based languages. The conceptual framework described in this section is mainly for class-based modeling.

The reason for devoting a special section to the conceptual framework for (class-based) modeling is that it is important to distinguish between the conceptual framework and the languages for expressing object-oriented analysis, design and/or implementation. No object-oriented notation is rich enough to be able to express the modeling of all relevant aspects of the real world. Any methodology should distinguish between the conceptual framework and the notation. In most presentations, the notation/language is presented as the central element, and the conceptual framework is restricted to what can be expressed in the language. The problem with this is that the developer is limited by the language since he/she is only able to model the real world by terms that can be expressed in the language. It is important that the developer has a conceptual framework that is richer than the language being used in order to have the best means for understanding and organising knowledge

---

[4]The conceptual framework presented here has been formulated as part of the development of Simula and BETA, but most of it is based on common philosophical terms and it may be applied to (class-based) object-oriented development in general.

about the real world. Of course, the model will eventually have to be expressed in the formal language, but the designer will then be explicitly aware of the compromises that have been made when doing this.

A more detailed presentation of the following conceptual framework may be found in Reference [MMN93, KLMM94].

In class-based languages, an object may be used to represents a physical phenomenon from the real world. A class may be used to represent a concept from the real world. This perspective on modeling is based on a real world conception in terms of phenomena and concepts.

A *phenomenon* is a thing that has a definite, individual existence in reality or in the mind; anything real in itself.

A *concept* is a generalised idea of a collection of phenomena, based on knowledge of common properties of instances in the collection.

The following individuals are all examples of phenomena: 'Indira Gandhi', 'Winston Churchill', and 'John F. Kennedy'. They are all covered by the general concept 'Person', but also the more special concept 'Statesman.'

A concept is traditionally characterised by the following terms:

- The *extension* of a concept refers to the collection of phenomena that the concept somehow covers.
- The *intension* of a concept is a collection of properties that in some way characterise the phenomena in the extension of the concept.
- The *designation* of a concept is the collection of names by which the concept is known.

The extension of the concept 'Person' includes the phenomena 'Indira Gandhi', 'Winston Churchill', and 'John F. Kennedy'. The intension of 'Person' includes the following properties: 'able to think', 'walks upright', and 'uses tools'. In addition to 'Person', the designation includes 'Human Being' and 'Homo Sapiens'.

In the above definition of concept, the properties constituting the intension are those that 'in some way' characterise the phenomena in the extension. Different views of concepts give different interpretations to 'in some way'.

In the classical Aristotelian view of concepts, concepts are organised in a logical hierarchical concept structure and the extension of a concept contains phenomena that all satisfy certain precisely defined requirements:

The *intension* of a concept is a collection of properties that may be divided into two groups: the *defining properties* that all phenomena in the extension must have and *the characteristic* properties that the phenomena may or may not have. The nature of the properties is such that it is objectively determinable whether or not a phenomenon has a certain property.

The Aristotelian view of concepts has through centuries demonstrated its usefulness in modeling systematic and scientific concepts as found within well established fields like mathematics, physics, zoology, and botany.

For most everyday concepts, it is, however, difficult to describe these as Aristotelian concepts. Examples of such concepts are, 'rock music', 'intelligence', and 'food'. Also some more technical concepts such as 'object-oriented programming' and 'structured programming' are difficult to describe as Aristotelian concepts. This has lead to the development of the so-called prototypical view of concepts (or fuzzy view on prototype theory). In the prototypical view, the intension is defined in the following way:

The *intension* of a concept consists of examples of properties that the phenomena may have, together with a collection of typical phenomena covered by the concept, called *prototypes*.

(Prototypical concepts are not the same as prototypical objects in prototype-based languages. A prototypical concept is still a concept - prototypical objects are not based on any notion of concept. We will return to this discussion in section 5.1.1).

The prototypical view differs from the Aristotelian view in a number of ways:

(1) It may not be objectively decidable whether or not a phenomenon has a property of the intension.

(2) The intension is given by examples of properties. All phenomena in the extension will have some of the properties, but rarely all. A phenomenon having some of the properties may not belong to the extension of the concept.

(3) The prototypes are typical phenomena belonging to the extension of the concept.

The major consequence of this is that the extension of a prototypical concept is not uniquely determined by the intension. It requires a human judgement to decide whether or not a given phenomenon belongs to the concept. The phenomena in the extension will have varied typicality and the borders between the concepts are blurred.

The prototypical view of concepts is more suited than the Aristotelian view to describe most everyday concepts. This is also often true for the problem-specific concepts identified during analysis of a given problem domain. However, most object-oriented methodologies are based on Aristotelian concepts in the sense that one of the purposes of analysis and design is to identify phenomena and concepts in the problem domain and to give Aristotelian definitions in terms of classes. A methodology should allow for prototypical concepts to be used during analysis. Due to the inabilities of the languages to support prototypical concepts, these may eventually have to be reformulated as Aristotelian concepts, but this reformulation should be an explicit step in the methodology.

A class may be used to represent an Aristotelian concept. The class description constitutes the intension of the concept, the instances of the class represents the extension of the concept and the name of the class represents the designation of the concept.

3.1   Means to organise knowledge

In the process of creating and producing knowledge, we use various means for organising and structuring our knowledge. The following are three commonly applied means of organisation for apprehending the real world:

Identification of phenomena and their properties. In perceiving the real world, people identify phenomena and their properties. The result of this is a number of *singular*

phenomena characterised by a selected set of properties. The phenomena are singular since they have not been classified in terms of concepts.

Classification. Classification is the means by which we form and distinguish between different classes of phenomena. That is we form concepts. Having identified phenomena and their properties and concepts, we group similar phenomena and concepts. A classification is often called a taxonomy. It is very common to construct taxonomies to compare various subjects. When classification is applied repeatedly, classification hierarchies may be obtained.

Composition. A phenomenon may be understood as a composition of other phenomena, i.e. there is a distinction between the whole phenomena and their component phenomena. A car consists of body, four wheels, motor, etc. A process of making pizza may be understood as a composition of several sub-processes, including: making the dough, making the tomato sauce, preparing the topping, etc. Repeated application of composition leads to composition hierarchies.

In class-based languages, classification may be represented by class/subclass hierarchies (inheritance). The use of inheritance for this purpose is often mentioned as one of the main characteristics of object-oriented languages. In the object-oriented literature, there is a lot of discussion of single versus multiple inheritance. This will be further discussed in section 4.2.

Composition has been supported by programming languages for a long time. Algol 60 [Nau63] supports composition of procedures, Pascal [Wir71] supports composition of record variables, etc. In object-oriented languages, composition has often been neglected since inheritance has been (mis) used also to achieve composition. In the design of BETA, support for composition has been just as important as support for classification (inheritance). (Composition will be further discussed in section 4.1.)

Identification is supported in BETA by means of so-called singular (classless) objects. It is possible to describe an object directly without describing it as an instance of a class. In real world modeling, it is often the case that such singular objects are identified and it is not necessarily a good idea to describe them as instances of some concept. From a technical point of view this is also found in Pascal where a variable may described directly without being an instance of a type. Similarly an inner block in Algol 60 is a singular procedure instance. However, no other class-based language supports singular objects with the same generality as BETA.

In summary: class-based languages are in general based on modeling the real world in terms of phenomena and concepts. They are good at modeling an Aristotelian view of concepts. They all support classification, some support composition, but few support singular objects. No class-based language is well suited for supporting a prototypical view of concepts. An explicit conceptual framework should be formulated in order to guide analysis, design and implementation and to identify the limitations of the programming languages. The conceptual framework should continuously be developed.

4. Abstraction mechanisms

In the following examples, BETA is used as a reference language. One of the unique characteristics of BETA is that abstraction mechanisms such as class, procedure, function, control abstractions, process type, exception type, etc, have been unified into one single abstraction mechanisms called *pattern*. By abstraction mechanism we mean any language construct that describe the structure of a set of entities and may be used to instantiate such entities. A class describe the structure of objects and may be used to instantiate objects; a procedure describe the structure of procedure activations and may be used to instantiate procedure activations, etc. The following is a simple example of BETA patterns:

```
Account:
    (# balance: @integer;
        deposit:
            (# amount: @integer
            enter amount
            do balance + amount -> balance
            exit balance
            #);
        withdraw: (# ... #)
    #)
```

The pattern `Account` describes objects that have 3 attributes: `balance`, `deposit` and `withdraw`. `Balance` is a simple integer variable. `Deposit` and `Withdraw` are patterns. In this example `Account` corresponds to a class and is called a *class pattern*, whereas `deposit` and `withdraw` corresponds to procedures and are called *procedure patterns*. The enter-part (`enter amount`) describes the input parameter, and the exit-part (`exit balance`) describes the output parameter of `deposit`. If `myAccount` refers to an instance of `Account`, then the following imperative makes a deposit of `500` and returns the new balance in `currentBalance`

```
500 -> myAccount.deposit -> currentBalance
```

4.1   Classification and composition

Object-oriented languages are often identified with languages supporting inheritance and of course it is difficult to claim that a language supports object-orientation if it does not include a mechanism for supporting inheritance. In class-based languages inheritance is supported by the class/subclass mechanism whereas in prototype-based languages it is supported by delegation. As mentioned elsewhere in this paper, it is not universally agreed whether or not the main benefit of inheritance is support for classification or reuse of code. However as said, in the Scandinavian school classification is more important than reuse.

A problem with the object-oriented community, is that there has been a tendency to overuse inheritance. Often inheritance is used to represent composition. However, from a modeling point of view, classification and composition are two distinct means for organising knowledge and a language intended for supporting modeling should have good support for both.

An object may contain components in the form of objects that are physically part of the object – such components are called part objects. An object may also contain components that are references to other objects

An object may also contain components that are patterns. This is often called *block-structure*. Finally an object may contain components that are references to patterns. Such references are in general closures that denote patterns and their lexical scopes.

The following table summaries the 4 forms of components. The rows indicate whether or not the component is a (physical) part or a reference. The columns indicate whether or not the kind of the component is an object or a pattern:

component/kind	object	pattern
part	part object	part pattern
reference	object reference	pattern reference

It is important to distinguish between 'object' and 'object reference'. In the next two sections we discuss these distinctions in further detail and give some examples.

4.1.1. Part object versus object reference

A part object is an object that is physically a part of the enclosing object. In BETA, part objects may be described as follows:

```
Car:
   (#  motor: @ (# ... #)
       body: @ (# ... #);
       wheels: [4] @ Wheel
   #)
```

A car object consists of part objects corresponding to the motor, body and 4 wheels. The symbol '@' means part object in BETA. The properties of motor and body are described directly using singular objects ((# ... #)), whereas the 4 wheels are described as an array consisting of 4 instances of pattern `Wheel`.

An object reference may also be part of an object, but in this case it is the reference that is a part. The reference refers to an object that is not necessarily part of the object. The following pattern shows examples of object references in BETA:

```
HotelReservation:
   (#  theCustomer: ^Customer;
       theHotel: ^Hotel;
       theRoom: ^Room;
       theDate: @ date
   #)
```

A hotel reservation consists of components that are references (described by '^') to `Customer`-, `Hotel`- and `Room` objects. The same customer, hotel, or room may be referred from many different hotel reservations. The corresponding attributes are therefore represented as references, not as a part objects. In the car example, a motor is a physical part of a specific car, whereas a customer is not considered part of a specific hotel reservation. I.e. a `Car` object has a physically embedded `motor` object, whereas a `HotelReservation` object has a reference to a `Customer` object and each `Customer` object may be referred/shared by many `HotelReservation` objects.

4.1.2. Part pattern versus pattern reference

A part pattern is a pattern that is physically part of an object. This corresponds to textually nesting of procedures and classes and is often called *block-structure*. It is well known from mainstream object-oriented languages that the attributes of an object may be procedures/methods, but classes as attributes are quite uncommon. In BETA class nesting may be used as in the following example:

```
Document:
    (#  content: @text; pos: @integer;
        new: ...; open: ...; close: ...;
        Selection:
            (#  first, last: @integer;
                cut: (#do (first,last)->content.delete #);
                copy: (#exit (first,last)->content.copy #);
                paste:
                    (# destination: ^document
                    enter destination[]
                    do (first,last)->destination.content.insert
                    #)
            #);
    #);
D1,D2: @Document;
S1,S2: @D1.Selection;
S3,S4: @D2.Selection
```

A document-object consists of a content that is a text. A variable `pos` contains the current (cursor) position in the document. A document has operations `new`, `open` and `close`. In addition a document object has a class pattern attribute `Selection` that is an object representing a selection in the document. A selection is characterised by an interval in the content, represented by the integers `first` and `last`. A selection has three operations, `cut`, `copy` and `paste`. `Cut` deletes the selection from content, `copy` returns a copy of the selection and `paste` copy the selection into a destination document.

Since `Selection` is a pattern declared textually within `Document`, each instance of `Document` has its own `Selection` class. The documents `D1` and `D2` thus each has a class `D1.Selection` and `D2.Selection` that are in fact different classes. The objects `S1` and `S2` are instances of `D1.Selection` and `S2`, `S3` are instances of `D2.Selection`.

Since `Selection` is part of a `Document` object, it is possible from a `Selection` object refer to variables and operations in the enclosing `Document` object. For example `S1.cut` will execute the operation

```
(first,last)->content.delete
```

where `content` will refer to the `content` variable in the enclosing `D1` document.

Textual nesting of classes is a common used feature in most of the Mjølner BETA libraries and application frameworks. The above example is a simplified example of how components (like class document) and anchors (like class Selection) are represented in the DEVISE hypermedia framework [GT94] .

Part patterns/nested patterns are similar to part objects in the sense that they are a fixed part of an object. Like it is possible to have references to objects, it is also possible to have references to patterns. Such references are closures in the sense that they refer patterns and

their textual enclosing scope. For a description of pattern references in BETA, see Reference [MMN93].

4.1.3. Support for composition in other languages

Composition is to some degree supported in object-oriented languages

- **Part objects.** C++ and Eiffel [Mey88] support part objects. In C++ a class attribute may be a instance of another class. In Eiffel it is a property of the class whether or not an instance of it is a part object. This means that all instances of a class are either part objects or referenced from other objects. They cannot be both.

  In BETA a part object may be an instance of any class and a part object may also be referenced from other objects.

  In BETA a part object may also bind virtual patterns[5] in the object being instantiated and operations in the part object may thus have an effect upon the enclosing object. This is possible by combining a part object with block structure and singular objects. This feature is a useful alternative to multiple inheritance, which is further discussed in section 4.2.

- **Object references.** Most object-oriented languages support object references. It is rarely considered a modeling feature, but mainly a mechanism for representing the state of objects. In Smalltalk it is the only way to represent the state of an object and this state is considered a private part of the object. In Reference [BC87] the representation of part objects in Smalltalk has been considered, and an extension of Smalltalk was proposed.

- **Part patterns.** Nesting of procedures is supported by most Algol like languages, including Pascal, but rarely in object-oriented languages. Nesting of classes is only fully supported by Simula and BETA. C++ allows nesting of classes, but within a nested class it is not possible to refer to names in the enclosing class. Nesting of classes in C++, is thus a pure scope mechanism.

- **Pattern references.** References to patterns has little support in most object-oriented languages. References to procedure patterns resemble procedure variables that may be assigned procedures as values. If a procedure value includes the static environment (object) of the procedure then it is a closure, but if it only includes a pointer to the code it is not. The function pointer in C is an example of procedure variables, but since C does not have bock structure there is no issue of a procedure value being a closure. C++ supports closures via pointers to member functions. Algol 60 supports closures via its formal procedure parameter mechanisms, and Lisp-like languages including Scheme [RC86] and CLOS [Kee89], have direct support for closures as first class values. Smalltalk and Self [US87] have closures in the form of blocks. A Self block can, however, not be invoked after its enclosing method has returned. In Smalltalk and Self it is not possible to pass a reference to a method as a closure. In Self it is, however, possible to obtain the same effect by using reflection.

---

[5]A virtual procedure pattern is similar to a virtual function in C++. Virtual patterns are further discussed in section 4.3.

References to classes are even more rare and are not supported by any of the mainstream statically-typed languages such as C++ and Eiffel. In Smalltalk classes are objects and in Self objects are used instead of classes. I.e. references to classes (in Self objects used as classes) are directly supported by the language.

4.2   Single and multiple inheritance

Since the early days of Simula, multiple inheritance is perhaps the issue that has been discussed mostly in the object-oriented community. There are people who think that multiple inheritance is a central element of object-oriented programming whereas other people think it should be avoided since it easily leads to complicated inheritance graphs.

If reuse of code is a major concern, then multiple inheritance is a technical solution to combining classes into new classes and to inherit the 'useful' parts and redefine whatever needs to be redefined. Multiple inheritance may be a quick way to write new code based on existing code. The disadvantage of multiple inheritance is that the inheritance structure express an evolution of the class hierarchies and the class/subclass relation does not express a conceptual relation.

A number of examples where multiple inheritance is used are often examples where composition/aggregation may be more natural. However, due to the poor support for composition in most object-oriented languages, people use multiple inheritance instead.

As mentioned, inheritance may from a modeling point of view be used to represent classification hierarchies. A classification or taxonomy is often tree structured since the phenomena being classified are grouped into disjoint classes. Tree structured classification hierarchies have been used within science for centuries to organise knowledge. However, as mentioned in section 3, everyday concepts may not easily be described as Aristotelian concepts organised in a tree structure and there are a number of examples where it is useful to organise concepts in a non-tree structured way and this is of course supported by multiple inheritance.

From a modeling point of view, multiple inheritance is useful in the cases where there is a need to represent a non-tree structured classification hierarchy. However, many so-called 'modeling' examples of multiple inheritance are really examples of a combination of two or more independent tree structured classification hierarchies. There is often a need to classify a set of phenomena according to several properties. A group of people may be classified according to their profession, nationality, religion, etc. and each of these classifications may be expressed by means of a tree structured classification hierarchy. This is called *multiple classification*. There is, however, no programming language that is able to directly represent multiple classification. Instead multiple inheritance may be used, but a multiple inheritance hierarchy does not properly represent this, since it is not possible to identify the original tree structured hierarchies.

To sum up: single inheritance is well suited for representing tree structured classification hierarchies. It would be desirable to have language support for representing multiple classification hierarchies including non-tree structured hierarchies, but multiple inheritance as found in most languages does not seem to be the solution. For BETA it was decided not to include multiple inheritance since it is not the right mechanism for modeling multiple classifications and since it seems technically complicated as a mechanism for reuse. Finally

the generality of BETA with respect to block structure, part objects and singular objects may be used to handle a number of cases where multiple inheritance is used in other languages. Examples of this are shown in the next section.

Multiple inheritance from part objects

Consider a class pattern for describing a street address:

```
Address:
    (#  street: ...; streetNo: ...; town: ...; country: ...;
        printLabel:<
            (#
            do  INNER;
                {print street, streetNo, Town, Country}
            #)
    #);
```

Besides attributes like `street`, `streetNo`, `town,` and `country` for describing the elements of the address, an address object has an operation for printing the address.

An address may be used for describing persons and companies. One approach is to make class `Person` and class `Company` inherit from class `Address`:

```
person: Address
    (#  name: ...;
        printLabel::<(#do {print name} #)
    #);
company: Address
    (#  name, director: ...;
        printLabel::< (#do {print name and director } #)
    #)
```

This is a typical example of how inheritance should not be used. A `Person` or `Company` is not a specialisation of an `Address`. It seems intuitively more correct to represent the address as a component (part-object) of a `Person` or `Company`. Using instances of class `Address` as part objects in `Person` and `Company` will not work, since `printLabel` needs different bindings corresponding to `Company` and `Customers`. In most languages it is necessary to introduce additional classes for this purpose [MM92]. Using part-objects and introducing additional classes, the above example may be rewritten as follows:

```
PersonAddress: Address(# printLabel::<(#do {print name} #)#);
CompanyAddress: Address
    (# printLabel::<(#do {print name and director } #)
    #);
Person:
    (#  name: ...;
        adr: @PersonAddress;
    #);
Company:
    (#  name, director:  ...;
        adr: @ CompanyAddress;
    #)
```

A problem with this approach is that inside `printLabel` of `PersonAddress` and `CompanyAddress` it is not possible to refer to the `name` and `director` attributes of `Person` and `Company` without forcing every caller to pass the enclosing `Person` or `Company` as an argument. The example does not show this argument.

In C++ it is possible to declare `PersonAddress` and `CompanyAddress` within `Person` and `Company`, but it is not possible to refer to variables in the enclosing `Person` and `Company` objects thus C++ only offers half of the functionality of block structure as mentioned above.

In a block-structured language, `PersonAddress` and `CompanyAddress` may be declared within `Person` and `Company` and thereby solving the scope problem. Thus in BETA it is possible to declare the `PersonAddress` and `CompanyAddress` classes within `Person` and `Company`. Moreover since BETA also supports singular objects, it is not even necessary to introduce the additional classes:

```
Person:
    (#  name: ...;
        adr: @address(# printLabel::<(#do {print name} #)#)
    #);
Company:
    (#  name,director:  ...;
        adr: @ address
            (#  printLabel::<
                    (#do {print name and director} #)#)
    #)
```
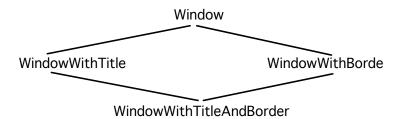
As the example shows, BETA supports inheritance from part objects in a similar way to inheritance from super patterns. Inheritance from part-objects is often used in this situation, since this makes it possible to bind/redefine the virtuals of, in this case, `Address`.

Furthermore it is possible to inherit from multiple part objects. In the following example, the class pattern `T` inherits from two part objects, being instances of class `A` and `B` respectively:

```
T:  (#  x: @ A (#  f1::< (# ...  #);
                   f2::< (# ... #);
               #);
        y: @ B (# g1::< (# ... #) #);
        ...
    #)
```

The `x`-objects binds the virtual patterns `f1` and `f2` and the `y`-objects binds the virtual pattern `g1`. In all 3 virtual bindings, it is possible to refer to variables in the enclosing `T`-class. This form of multiple inheritance from part objects is technically as powerful as multiple inheritance without overlapping superclasses suggested in Reference [Kro85] and corresponds to multiple inheritance in C++ with non-virtual base classes [Str87].

Block structure and multiple inheritance

Block structure can sometimes be used as an alternative to multiple inheritance. In Reference [Øst90] it is shown how a common use of multiple inheritance for modeling windows with titles and borders may be handled using block structure. Since a window may have a title, a border or both, the following multiple inheritance class hierarchy is often used:

```
                         Window
              /                        \
      WindowWithTitle            WindowWithBorde
              \                        /
             WindowWithTitleAndBorder
```

This window hierarchy may be described using nested patterns:

```
Window:
    (#  Title: (# ... #);
        Border: (# ... #);
        ...
    #);
aWindow: @ Window(# T: @Title; B: @Border #)
```

The descriptions for title and border are made using nested patterns. For a given window, like `aWindow`, a title object and a border object may be instantiated. For example if two titles are needed, two instances of `Title` are made.

4.3   Genericity and virtual classes

One of the distinguishing features of BETA is the unification of abstraction mechanisms like class, procedure, function, etc. into one general abstraction mechanisms: the pattern. In the previous section we have discussed one aspect of patterns in the form of virtual procedure patterns. Virtual procedure patterns correspond to using virtual patterns as virtual functions in C++ or methods in Smalltalk.

A virtual class pattern is a virtual pattern used as a class. Virtual class pattern is the BETA construct corresponding to class parameters of generic classes in Eiffel and template classes in C++. Consider the following example:

```
Set:
    (#  elm:< Object;
        R: [100] ^elm; top: @ integer
        Insert:
            (# E: ^elm
            enter E[]
            do E[]-> R[top+1->top][]
            #);
        ...
    #)
```

The `Set` pattern has a virtual class pattern attribute, `elm` that is the pattern of elements to be inserted into the set. Since `elm` is a virtual class pattern, it may be constrained in subpatterns of `Set`. In the above declaration it is constrained (or qualified) by the pattern `Object`. Since all patterns are subpatterns of `Object`, any pattern may be used for `elm`. I.e. any object may be inserted into a `Set` object. The declaration of `Set` thus corresponds to an unconstrained generic class in Eiffel, or a template class in C++.

In general pattern `Set` may be used as any other pattern in BETA, including as a super pattern:

```
PersonSet: Set
    (#  elm::< Person;
```

```
Find:
    (# name: ^text; E: ^elm
    enter name[]
    do {find an element E, where E.name=name}
    exit E[]
    #)
...
#)
```

In pattern `PersonSet`, `elm` has been constrained to pattern `Person`. Consequently only instances of pattern `Person` may be inserted in a `PersonSet`. Within `PersonSet`, all references qualified by `elm` are known to refer to instances of at least pattern `Person`. I.e. it is possible to refer to `Person` attributes through such references. For example in pattern `Find`, the expression `E.name` is legal since `E` must refer to a `Person` object.

Pattern `PersonSet` corresponds to a constrained generic class in Eiffel. In C++ this kind of constrained genericity is not possible.

It is generally agreed that some form of parameterised class mechanism is needed for statically-typed languages. C++ offers a simple form; Eiffel and BETA offer more advanced forms. Whether or not the differences between the approaches taken in the latter languages are important may be difficult to judge. In practice it seems that all three languages can express the same. However, there is a difference with respect to the definition of subclass/subtype/-conformance of such parameterised classes and in the handling of covariance and contravariance. The latter will be further discussed in section 4.6 below. For a further discussion of parameterised classes see Reference [MMM90].

4.4   Inner versus super

The inner mechanism is often mentioned as a characteristic feature of BETA and BETA is sometimes characterised as 'doing inheritance the opposite way' [Coo89]. However, the BETA's class/subclass mechanism is very similar to most other class-based languages in the sense that instances of a subclass have all the attributes defined in the superclass. The main difference is with respect to method combination: In most languages, a method from a super class may be redefined in the subclass and there is then a notation for executing the method in the superclass if this is needed. In BETA a method may not be redefined, but only extended (specialised) and the inner mechanism is used for expressing this.

Consider a class `T` with an init method. In Smalltalk style this may look like

```
T: class
    (# ...
        init: method (#do {initialise variables in T} #)
    #)
TT: class T
    (#  ...
        init: method
            (#do super init; {initialise variables in TT} #)
    #)
```

In BETA this may be described in the following way

```
T:   (# ...
        init:< (#do {initialise variables in T}; INNER #)
    #);
TT: T(# ...
```

```
        init::< (#do {initialise variables in TT}; INNER #)
    #)
```

In the Smalltalk style, the subclass may override a method definition in the superclass and it is the responsibility of the subclass to execute the corresponding method in the superclass. The subclass may also decide not to execute the method in the superclass. In BETA, the subclass may extend the definition of the method in the superclass. It is then the responsibility of the superclass designer to insert `inner` where the subclasses may need to extend the behavior. The motivation for this is that its should not be possible to eliminate properties that have been established in the superclass. If certain properties have been proved these should also hold for the subclass.

There are pros and cons for both alternatives. The Smalltalk redefinition style is more flexible when it comes to reusability, since it is possible to inherit from a class and redefine any parts that needs to be different. This can be done without regard to any conceptual relation between a subclass and its superclass. As discussed below, this may be the reason that many people find it necessary to distinguish between class and type. The Smalltalk style of redefinition has been used so much in practice that there exist a lot of subclasses that are not proper specialisations of their superclasses.

The disadvantage of the BETA style is that it may be difficult to anticipate a proper placement of inner. This is often the case when a method in a class defines some default actions that may redefined in a subclass. For BETA it has been considered to introduce a more direct support for default actions.

The most useful applications of inner is for defining control patterns. All BETA libraries and frameworks contains numerous examples of control patterns defined using inner. The container libraries define iterators by means of inner, the concurrency library defines abstract classes for implementing monitors as in Concurrent Pascal [Bri75] and rendezvous as in Ada by means of inner. The monitor example is shown in section 6 below; for other examples on this see Reference [MMN93]. Such control patterns cannot be defined using the Smalltalk redefinition style. The block concept of Smalltalk may be used to define control patterns, but this is less elegant. C++, and Eiffel have no similar feature.

CLOS has richer support for method combination than BETA. The inner mechanism may be modeled using the CLOS meta-object protocol [KBR91]. Fundamentally, however, the CLOS philosophy is basically redefinition.

4.5   Dynamic typing versus static typing

One of the classical discussions within (object-oriented) programming languages is the issue of static versus dynamic typing. The arguments in favour of dynamically typed languages are:

• The programming environments available for these languages are in general more advanced, especially in their support for incremental compilation and execution.

• Static typing hinders code reuse [SU95].

The arguments for statically-typed languages are:

- Strong typing is an advantage when developing production software since it improves the readability of the code and makes it possible to to catch more errors at compile-time.
- Compilers for statically-typed languages in general generate more efficient code.

It should be obvious that static typing makes it possible to catch more errors at compile-time. Type inference [APS93] may help in catching some of these errors before the program is run, but it cannot completely compensate for the lack of type information.

It is true that it requires more work to implement an incremental environment for a statically-typed language. However, as demonstrated by the Mjølner ORM System [Mag94b] it is possible to develop efficient incremental environments for statically-typed languages.

The dynamically-typed languages have the opposite problem: it is easy to implement incremental environments for such languages, but it requires more work to develop compilers that can produce efficient code. In the literature, there is a large collection of articles describing implementation techniques for dynamically-typed languages [DS84, Ung86, HCU91, HU94, AH95]. Some of the compilation techniques developed for dynamically typed languages are also relevant for statically typed languages, especially the ones allowing inlining of virtual function calls.

State-of-the-art research has thus shown that it is possible to implement incremental environments for statically-typed languages and to have efficient implementations of dynamically-typed languages.

The remaining issue thus seems to be whether or not static typing hinders code reuse and whether or not static typing improves readability of code. Since most pros and cons in this discussion are considered well know, this issues will not be further discussed in this paper.

4.6. Contravariance and covariance

For statically-typed languages there is an open issues with respect to covariance, and contravariance. In the following example, the difference between covariance and contravariance is illustrated[6]:

```
Pair: class x,y: integer end;
Point: class Pair
          procedure equal(p: Point): boolean
             return (x = p.x) and (y = p.y)
          end
       end;
ColorPoint: class Point
               c: color;
               procedure equal(p: ColorPoint): boolean
                  return super.equal(p) and (c = P.c)
               end
            end
```

---

[6]The syntax used below is not BETA and is invented to illustrated the co/contra-variance discussion. `Point` and `ColorPoint` inherits from `Pair` and `Point` respectively, otherwise the syntax should be self explanatory.

The class `Pair` defines objects with 2 instance variables, `x` and `y`. `Point` is a subclass of `Pair` and defines a (virtual) procedure `equal`. The parameter of `equal` is of class `Point`. `ColorPoint` is a subclass of `Point`. It adds an instance variable `c` and redefines the `equal` procedure. The parameter of `equal` for `ColorPoint` is of class `ColorPoint`. The procedure `equal` is a said to be *covariant* (in its argument), since its argument varies in the same sense as class `Point` and `ColorPoint`: `ColorPoint` is a subclass of `Point` and the argument of `ColorPoint::equal` is a subclass of the argument of `Point::equal`.

Let `p1` and `p2` be instances of class `Point` and `c1` and `c2` be instances of class `ColorPoint`. It should be obvious that the following expressions are legal:

```
p1.equal(p2)                                    (1)
c1.equal(c2)                                    (2)
```

The so-called *subtype substitutability* property says that at any place where an instance of class `Point` is legal, an instance of class `ColorPoint` is also legal. This implies that the following expression is legal:

```
p1.equal(c1)                                    (3)
```

The following expression is not legal, since `c1.equal` expects a `ColorPoint` and `p1` is an instance of `Point`.

```
c1.equal(p1)                                    (4)
```

The purpose of *static typing* is to have the compiler detect the legality of the above expressions. Assume that `p1, p2, c1` and `c2` are declared in the following way:

```
p1,p2: Point; c1,c2: ColorPoint
```

It seems that the compiler should be able to check that (1-3) are legal and that (4) is illegal, but in general the compiler does not have enough information to do this. In the above example we said that `p1` and `p2` referred to instances of `Point`, but `p1` may also refer to an instance of `ColorPoint` that will make (1) illegal and (4) legal. In general run-time checks are needed to check the legality of the above expressions.

The heart of the problem is the ability to specialise the type of the arguments of virtual procedures. In the above example the argument of `equal` happens to be the same as the enclosing class, but this is not essential.

Consider instead the opposite possibility that the argument can only be generalised. In the above example this will correspond to defining `ColorPoint::equal` in the following way:

```
procedure equal(p: pair): boolean
    ...
```

I.e. the argument of `equal` is a superclass of the argument of `point::equal`. `Equal` is said to be *contravariant* (in its argument). In the body of `equal` it is thus not possible to refer to attributes of `ColorPoint` and all of the above examples can therefore be checked at compile-time. While theoretically appealing for its type-safety, contravariance is rarely useful in practice. For instance it does not seem useful to generalise the argument of `equal` to be a `Pair`. On the other hand examples of covariance are often found.

It is well known that at most two of the following three properties can be obtained at the same time:

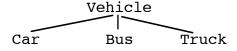1. Static typing
2. Subtype substitutability
3. Covariance

Some languages, like Trellis/Owl [SCBKW86], and Modula-3 [Nel91] do not have covariance in order to be able to perform static type checking. Instead they support contravariance. C++ has neither covariance nor contravariance (often called no-variance). BETA has abandoned full static typing and performs run-time checks of covariant parameters. Eiffel has some form of covariance, but rejects cases where a simple flow analysis (called system validity check) cannot assure the correctness.

The main arguments for the BETA approach is that covariance is needed in practice, whereas there don't seem to be any useful examples of using contravariance. In many cases, the run-time checking can be avoided, since instances of higher-order classes (patterns with virtual class parameters and covariant parameters) are often declared as singular static instances. I.e. the compiler knows the exact type of the arguments. This corresponds to the type exact arguments [PS90]. It would also be possible in BETA to perform flow analysis as in Eiffel and thereby at compile eliminate some run-time checks or detect errors that otherwise would have been deferred to run-time. It would thus be possible to translate[7] all Eiffel programs into BETA and no run-time checks will be needed. On the other hand there will be BETA programs (with run-time checks) that cannot be translated into Eiffel.

Type inference [APS93] is another technique that may be used to avoid some of the run-time checks and run-time errors. However, the Mjølner BETA implementation does currently not perform, any of these optimizations. Covariance in BETA is not supported in the direct form shown above, but is available using virtual class patterns [MMM90].

In BETA run-time checking for covariant parameters is considered another variant of the run-time check for reverse assignment being carried out in most statically-typed languages. Consider a class hierarchy

```
              Vehicle
                 |
    Car         Bus        Truck
```

and reference variables

```
    aVehicle: ^ Vehicle
    aCar: ^ Car
    aBus: ^ Bus
```

Assignments of the form are legal

```
    aCar[] -> aVehicle[]
```

whereas assignments of the form

```
    aCar[] -> aBus[]
```

---

[7]In this context *all Eiffel* and *all BETA* programs are meant to refer to subsets of the languages with covariant parameters. There are other elements of theses languages that are not easily translated into the other language.

are illegal and both assignments can be checked at compile time. Reverse assignments of the form

```
aVehicle[] -> aCar[]
```

can in general not be detected at compile-time. In Simula, BETA, and Eiffel the validity is checked at run-time. If the assignment is not valid, a run-time error/exception is generated for Simula and BETA. In Eiffel, the destination reference is given the value NONE. In C++, it is the responsibility of the programmer that the assignment is valid and no run-time checking is carried out. As can be seen, reverse assignment is an example of run-time type checking and run-time checking for covariance parameters is an example of the same kind of checking.

4.7   Types versus classes

One of the open issues within the object-oriented community is the discussion of type versus class. Is it the same or should there be a distinction. Traditionally it has been considered a great advantage that class/subclass relations also define type/subtype relations. However, as mentioned above, subclassing and redefinition has been used a lot to obtain code reuse, leading to subclasses without any conceptual relations to their superclasses. Consequently, it is now more common to distinguish between classes and types. In this section we shall try to summaries the discussion and some of the various point of views.

The main arguments for distinguishing between type and class are: a type defines the interface of a set of objects and a class defines its implementation; for a given type there may be many different implementations; a subclass of a given class does not necessarily define a subtype of the type implemented by the type of the class; and specification and interface should be kept separate.

Type definition. The type and subtype relation is often defined in terms of interfaces:

* A type is a set of interface functions.

* A type B is a subtype of A if all interface functions of A are also included in B. In addition there are some constraints on the parameters of the interface functions depending on whether or not covariance, contra variance or no-variance is supported. This however, is not important for the discussion here.

The problem with such a definition is that the designation (name) of a type has no importance. This means that incidental name collapse may result in unintended subtypes. Consider the well known example (due to Boris Magnusson)

```
Symbol: type
   (#  move: proc(...);
       draw: proc(...);
    #)

Cowboy: type
   (#  move: proc(...);
       draw: proc(...);
       shoot: proc(...);
    #)
```

The type Cowboy is a subtype of Symbol since it includes all the operations of Symbol. From a modeling point of view this is not desirable. A type or class represents a concept and

a concept is characterised by its extension, intension and designation. In the above definition of type, only the intension is used to characterise the type. The designation is not used in the definition of the subtype relation.

When comparing object-oriented programming to functional programming and relational databases, one of the main arguments for object orientation is the explicit support for identity of objects. In functional programming and relational databases, only the value of attributes are important. Two objects are considered identical if they have the same value. In object-oriented programming two objects may have the same value but still be different with respect to their identity. The designation of a type (or class) is analogous to the identity of an object. Defining a type without using the designation is a value oriented approach eliminating identity for types. (Markuu Sakkinen)

The above definition of type and subtype is based on the intension and ignoring the designation. In some definitions of type and subtype, the extension also seems to be ignored, since any object satisfying the interface is considered an instance of the type.

In the discussion on support for classification it was mentioned that it is often useful to be able to describe two or more classifications of the same objects. One way of classifying objects is based on the set of interface functions they define. Such a classification leads to a classification hierarchy corresponding to a subtype relation based only on interfaces. This may in many situations be a useful way of classifying objects, but it should not be the only one, and it is not the most important one from a modeling point of view. From a pure software engineering point of view it also cannot be useful just to base subtypes on interfaces, since this means that one accidentally may replace one object with another just based on their interfaces.

It possible to define a subtype relation that takes the name of the types into account. This will then correspond to separating the interface of class from its implementation as discussed in the next section.

Separation of interface and implementation

An argument for distinguishing between type and class is the separation of interface and implementation. The type defines the interface and the class defines the implementation. It is definitely necessary to be able to separate the interface of a class from its implementation, and most object-oriented languages supports such a separation. Basically there are three ways:

The abstract data types (ADT) approach. There are pre-defined rules for what is visible via the interface. Often operations (methods) are visible whereas (instance) variables are hidden. This is called the ADT-approach since an ADT is a type characterised in terms of its operations. Smalltalk is an example of a language using the ADT-approach.

The export approach. The programmer must define which attributes are visible and which are hidden. This is the case in Simula , C++, and Eiffel where mechanisms like hidden and protected (Simula), public, private, protected (C++), and export (Eiffel) are used for this purpose.

The modularisation approach. A class definition is split into an interface part and an implementation part that often resides in different modules and/or files. This is the case for Modula 3 and BETA.

In the ADT-approach, the interface only consists of operations whereas the export- and modularisation approach allow other attributes as well. The difference in approach is an open issue since there is no general consensus on which one is best. When programming general libraries and frameworks of classes, the public interface should be well defined and the internal details of a class should not be exposed. The ADT-approach seems to support this directly, but has no way of hiding private operations, and when instance variables are part of the public interface, extra interface operations will have to be invented. The export- and modularisation approaches can simulate the ADT-approach, but require convention and discipline. In addition they make it possible to hide private operations and to define instance variables as part of the public interface.

The advantage of the modularisation approach over the export approach is that the interface and the implementation are split into two or more modules or files. It is therefore possible to change the implementation parts of a class without changing the file containing the interface. In addition it is simple to implement a compiler where the implementation of a class can be changed and recompiled without having to recompile all clients of the class. Support for incremental compilation is an important practical issue for statically-typed languages.

One reason for not choosing the ADT-approach, is that in many modules there are a large number of local classes for which it is inconvenient to access objects via interface operations. When classes have local visibility only, accessing instance variables directly is harmless.

When defining data models, the bottom layer of the database often consists of raw data to be represented by objects. These objects may be stored in an object-oriented data base. Another set of classes is often provided to access these data, such that change in the representation does not affect the user. When accessing the basic data, it is both inconvenient and unnecessary to do so via interface functions.

Finally from a modeling perspective, it is desirable to describe a class and a set of objects in terms of classification and composition. In addition to describing the class/subclass relations, it is thus also necessary to be able to describe the composition structure in terms of part objects/patterns and object/pattern references. Making part objects and references part of the public interface may be a dilemma with respect to encapsulation and uniform reference, since a change in the model affects all usage's of the description. A uniform notation for accessing attributes may compensate for this, but such a notation may be difficult to obtain.

It may also be the case that there is a need to implement the model in another way than it is described. This is easy when the interface/model is just a set of operations, but more difficult when part objects, and references are also part of the interface.

In general there seem to be a conflict between modeling and information hiding. For modeling one wants to describe the structure of a object in such a way that its elements represent aspects of the real world phenomenon. For information hiding one wants to hide as much of the structure of an object as possible.

5. Prototype-based languages

One of the most interesting developments in object-orientation is the languages based on prototypical objects [Bor86, Smi86, Lie86, US87]. These languages offer a new perspective on modeling and from a technical point of view are simpler yet in some aspects more general than class-based languages. In this section we shall comment on the modeling capabilities of prototype-based languages and compare them with class-based languages. We argue that there is a need for both perspectives on modeling. Finally we compare class-based and prototype-based languages from a technical point of view.

5.1 Modeling in prototype-based languages

The modeling perspective of prototype-based languages is based on a theory of people identifying new phenomena by comparing their properties to phenomena they already know. The standard example [Lie86] is the elephant Clyde that represents a prototypical elephant in the mind of some person. If this person encounters a new situation with another elephant, say one named Fred, facts about Clyde can be used for understanding properties about Fred.

In general a new phenomenon is identified as being similar to one or more (prototypical) phenomena well known to the perceiver. 'Similar' means that the new phenomenon has the properties of the parent phenomenon except for certain specific properties that may differ, not exist, or be new. In general the description of a new phenomenon lists the properties that differ and for all other properties the description may be found at the parent phenomena. Technically, references to descriptions in parent objects are handled by delegation.

The rationale behind this modeling perspective is that this is how children perceive the world and that people in general perceive new phenomena in this way.

Consider some additional examples: Max the dog may be grasped by a child who notices properties of Max, like '4 legs', 'a tail', 'a mouth', 'sharp teeth', 'is soft', etc. The child then meets Kit (a cat). Kit is like Max, except that it also has 'sharp nails' and can 'crawl in trees'. The child may then meet Boni (a pony) that is like Max except that it does not have sharp teethes, but it has 'hooves', 'mane', etc. Then Lassie (the dog) shows up. It is definitely like Max. New animals horses, cows, and even the elephants Fred and Clyde may be perceived by this child.

The understanding of real world phenomena using the prototypical view may be useful in early stages of analysis where new phenomena are encountered and their properties and relations to other phenomena are unknown. However, as time goes on and more and more properties become known, the knowledge must be structured and organised systematically - otherwise there will be chaos.

The prototype-based view does not capture what is special about Fred and Clyde and what is general about them. What does 'like' mean? Is it size, form of teeth, male/female. During the knowledge acquisition process you will organise your phenomena into groups according to similarities. Lassie, Max and Kit don't eat grass. You realize that Lassie and Max are more alike than Kit, Fred and Clyde, etc. You form concepts/abstractions and make generalisations/specialisations of these concepts. I.e. you classify your phenomena and concepts. Eventually you form concepts, like Dog, Cat, Elephant, Horse, etc. You can now use the new understanding/knowledge in the form of concepts to get a better understanding of your domain.

5.1.1. Prototypical concepts versus prototype-based languages

A natural question to ask is whether or not the prototypical concepts described above correspond to modeling based on prototype-based languages. The answer is no! Prototypical concepts are based on a distinction between phenomena and concepts. The intension of a prototypical concept is described/characterised by means of properties and prototypes. The prototype-based modeling is only based on phenomena and relations between these. There is no notion of concepts involved.

When prototype-based modeling is applied in practice, it seems that new phenomena are modeled (cloned) based on a few distinguished prototype objects. These objects are thus very close to serving the same purpose as a concept. We return to this issues below.

5.2 Combining prototype-based and class-based modeling

Prototype-based modeling is probably useful in the early analysis and design phases of a project. Here you identify single phenomena without grouping them into concepts. You understand a new phenomenon in terms of its similarities with other phenomena. In this stage you have little structure on your knowledge of the application domain of which your are trying to obtain knowledge. The situation is the same in a design situation. When you create a new artifact, you may base it on another artifact with similar properties, but modify it to suit your new requirements.

However, during the knowledge acquisition process you obtain more and more knowledge of your domain and you obtain certain means for organising your knowledge. These means are often classification and composition as mentioned above. You group your phenomena into concepts. You understand your phenomena in terms of compositions of other phenomena. You classify your concepts into classification hierarchies (generalisation/specialisation hierarchies).

During this process you constantly reorganise your concepts and phenomena. You may consider a concept as an idealised thing, but eventually you think of the idealised thing, not a specific object. In your programming language, a class represents a concept. Whether or not you prefer to view a concept as an idealised thing or immutable object does not really matter.

It is of course debatable how people form concepts. We claim that the above description characterise how people forms everyday concepts. Within science, there are many examples where knowledge is organised in this way. Scientists organise their knowledge about a certain application domain by means of classification and composition. In a recent paper [DMC92] prototype-based languages are classified according to traditional Aristotelian principles - i.e. the principles underlying class-based modeling are used to characterise prototype-based languages. If the prototype-based modeling principles should be followed, the languages should instead have been characterised in terms of clones of each other.

We thus claim that prototype-based modeling is useful in the early stages of analysis and design, where you identify new phenomena and their properties without having a clear understanding of their relations to each other and to already well known phenomena and concepts. When more and more knowledge is obtained about the concepts, class-based modeling starts to apply in the sense that phenomena are classified according to common

properties. I.e. concepts and classes are formed and organised into generalisation/specialisation hierarchies

This view of how prototype-based modeling and class-based modeling can be combined is in accordance with the way Karl Marx has described the *process of producing and using knowledge* [Mat81], i.e. issues related to the theory of knowledge (also called epistemology). The *knowledge acquisition process* may be split into three levels:

The level of empirical concreteness. At this level we conceive reality or individual phenomena as they are. We do not realize similarities between different phenomena, nor do we obtain any systematic understanding of the individual phenomena. We notice what happens, but neither understand why it happens nor the relations between the phenomena. In the programming process this corresponds to a level where we are trying to understand the single objects that constitute the system. We have little understanding of the relations between the objects, e.g. how to group them into classes.

The level of abstraction. To understand the complications of the application domain, we have to analyse the phenomena and develop concepts for grasping the relevant properties of the phenomena that we consider. In the programming process this corresponds to designing the classes and their attributes and to organising the classes into a class/sub-class hierarchy. At this level we obtain a simple and systematic understanding of the phenomena in the referent system.

The level of thought concreteness. The understanding corresponding to the abstract level is further developed to obtain an understanding of the totality of the application domain. By having organised the phenomena of the application domain by means of concepts, we may be able to understand relations between phenomena that we did not understand at the level of empirical concreteness. We may also be able to explain why things happen and to predict what will happen.

The above is an identification of three levels appearing in the process of creating and producing knowledge. In this process the means used for organising and understanding knowledge depend on the *perspective* of the person. Object-orientation is an example of a perspective. Other examples are function-oriented and constraint-oriented.

5.3   Programming language support

The above discussion may be used to set up requirements for a programming language and environment. Overall the environment (and language) should support:

The exploratory phase. This is the phase where new phenomena/objects are identified. In this phase it is necessary to be able to describe single objects without describing them as instances of classes. And it is necessary to be able to describe new objects as clones modified from existing objects. Prototype-based languages are clearly superior here. As mentioned previously, BETA is one of the few class-based language where it is also possible to describe a single object. Many class-based languages also supports cloning of objects. However, class-based languages do in general not support implicit delegation, but explicit delegation is usually possible.

In this phase it is also necessary to be able to make incremental modification of classes. Most class-based languages support some form of incremental modification of classes

by means of subclassing with replacement of virtual procedures. Use of multiple inheritance and mixins improve the possibilities for exploratory development.

BETA has less support for incremental modification than most class-based languages, since it is impossible in BETA to completely redefine virtual attributes. They can only be extended.

The structural phase. This is the phase where the current understanding of concepts and phenomena are organised into structures. The exploratory style easily leads to a complicated structure of objects and classes with unclear dependencies on parent objects and classes. It is absolutely necessary that the structure and relations between objects and classes are clarified. This may be done by using classification and composition. Classification for forming new classes and organising these into generalisation/specialisation hierarchies. Composition for organising objects into compound objects as described above.

Prototype-based languages are weak here, since classification is only possible by conventions.

Class-based languages by definition support classification, but only some supports composition. The virtual concept of BETA is a strength here, since descriptions of classes are clearer and more concise than when using a language with redefinition of virtuals as opposed to extension.

Support for prototypical concepts

In the exploratory phase and perhaps also the structural phase, it might be useful to have support for prototypical concepts also. A language based on prototypical concepts should among others include the following mechanisms for describing concepts and generating instances of it:

- It should be possible to describe a set of properties that phenomena in the extension may or may not have

- It should be possible to describe one or more prototypes representing typical phenomena in the extension of the concept.

- New objects in the extension of the concept may be created by cloning one of the prototype objects.

5.4   Class- and prototype-based languages

From a language design point of view, it is desirable with as few concepts and constructs as possible. BETA has unified class, procedure, function, etc. into one generic concept, the pattern. The motivations for doing this was:

- A proliferation of abstraction mechanisms in programming languages was introduced during the seventies. Examples of abstraction mechanisms being introduced were procedure, function, class, type, module, package, process, task, generic package, generic task, exception, iterator, etc. Some of these mechanisms are just different names for the same concept, but often with a slightly different semantics. It is possible that there is a need for a number of different abstraction mechanisms, but in that case they should be treated in a uniform way. Most abstraction mechanisms are treated quite

differently in most languages. A excellent discussion of this with respect to Ada may be found in Reference [Weg83].

- From a modeling point of view, we start with phenomena and concepts in the real world. A program execution should similarly be described in terms of phenomena and concepts. The phenomena are then viewed as objects and concepts as classes.

What has been obtained? It has been shown that it is possible to unify abstraction mechanisms from both a conceptual and technical point of view. Even though, it may still be useful to distinguish between abstraction mechanisms like class, procedure, process, exception, etc., these are handled in a uniform way. Subclassing is available for procedures, processes, exceptions, etc. The virtual concept is also defined for classes, processes, exceptions, etc. Classes, procedures, processes, exceptions, etc. can be arbitrarily nested. The latest addendum to BETA was the introduction of patterns as first-class-values. Again this mechanism handles all abstraction mechanisms in a completely identical way.

In Smalltalk classes are objects and expressions and control structures are messages sent to objects, but there is no unification of class, method and block. In prototype based languages, objects are used instead of classes and new instances are created by cloning existing objects and by adding and removing attributes of objects. Sharing of state and behavior is obtained by means of delegation. This is simple and elegant and gives a number of new possibilities not found in class-based languages. For a discussion of this we refer to papers about protype-based languages [Bor86, Lie86, Smi87, US87, DMC92, SU95].

Self provides an elegant unification of state and behavior: an object may redefine instance variables as well as methods from its parent-objects. An instance variable of a parent-object may be redefined as a method or vice versa. See [US87] for a further description of this.

Since the topic of this section is the extent to which language mechanisms have been unified, we shall briefly summarize the various techniques for simulating class-based programming in prototype-based languages. It is often claimed that prototype based languages have unified classes and objects. In general prototype-based languages do not support programming in a class-like style. A simple hierarchy of classes with instance variables and methods cannot be directly described in a prototype-based language. The hierarchy can be simulated, but the same degree of sharing of attributes as in class-based languages cannot be realised. See Reference [DMC92] for a further discussion of this.

In Self the common style for expressing class-like sharing is through the use of *traits*-objects and *copy-down*. A class is split into two objects a prototype containing the instance variables of the class, and a traits-object containing the methods of the class. A subclass is similarly split into a prototype-object and a traits-object. The traits-object inherits from the traits-object of the superclass. The prototype-object is marked as a so-called *copy-down* of the prototype of the superclass. This implies that all instance-variables of the prototype of the superclass are copied to the prototype of the subclass. Copy-down is not part of the Self language but a mechanism in the Self programming environment. Consider the following class A with subclass B:

```
A: class
       var x;
       var y;
```

```
            method m1: ...
            method m2: ...
       end
   B: class A
            var z;
            method m2: ...;
            method m3: ...;
       end
```

These classes cannot be represented as straight-forward objects a and b where b inherits from (or delegates to) a[8]:

```
a = (| x. y.
       m1 = ( ... ).
       m2 = ( ... )
     |)
b = (| parent* = a.
       z.
       m2 = ( ... ).
       m3 = ( ... )
     |)
```

The problem is that when instances of b are generated through cloning of b, all these instances will share the same parent object a. I.e. there will be only one set of instance variables x and y. Instead, these classes may be represented by using traits-objects and prototypes:

```
traitsA = (| m1 = ( ... ).
             m2 = ( ... )
           |)
a = (| parent* = traitsA.
       x.
       y
     |).
traitsB = (| parent* = traitsA.
             m2 = ( ... ).
             m3 = ( ... )
           |).
b = (| parent* = traitsB .
       x. "marked as copy-down from A"
       y. "marked as copy-down from A"
       z
     |)
```

This scheme works because traits-objects have no state, only methods. When instances of b are generated through cloning of b, the clone gets its own copies of all the instance variables. The b-clones all inherits behavior from traitsB that again inherits from traitsA, thus any method defined for traitsA also applies to b-clones.

In Self it is also possible to simulate class-based programming using dynamic inheritance. Then the splitting of a class into the proto traits pair can be avoided. Consider the following new version of objects a and b. In this version the clone-operation for b also clones its parent-object a, and the parent of the b-clone is set to be the a-clone.

```
a = (| parent* = traits clonable.
```

---

[8]Self syntax is used below.

```
        x.
        y.
        m1 = ( ... ).
        m2 = ( ... )
    |).
b = (|  parent* <-   a. "dynamic parent"
        z.
        m2 = (...).
        m3 = (...).
        clone = (| s. p |
            s: resend.clone. "s is a clone of b sharing a with b"
            p: a clone.      "p is a clone of a"
            s parent: p.     "p is made the parent of s"
            s                "s is returned as the value"
        )
    |)
```

Dynamic inheritance is an example of a feature that is not supported by class-based languages. As shown it may be useful to support class-based programming. On the other hand it may lead to quite complicated programs and may be difficult to implement efficiently [SU95].

Cecil [Cha92] takes the distinction between prototypes and traits-objects a step further. Cecil has three types of objects: (1) abstract objects that are only used to inherit from, (2) prototype-objects that may be used for cloning only, and (3) real objects. Messages may not be sent to abstract objects and prototype objects.

Summary

BETA has succeeded in unifying class, procedure/method, etc into the pattern concept. We are more skeptical to the extent to which objects and classes have been unified in prototype-based languages. If a prototype-based language does not support class-based programming then it has not unified class and object. One may argue of course that it is not important to support class-based programming, but this is another discussion.

The division between class-based and prototype-based languages is perhaps more blurred than it may appear: In BETA it has always been possible to describe so-called singular objects. A singular object is described directly, not as an instance of a class. This was motivated from a modeling as well as a technical point-of-view. In the real world you often encounter one-of-a-kind phenomena that you may not wish to model as an instance of a class. Technically it is inconvenient to invent a class name and an instance name for such single objects. A (singular) object in BETA can be used to generate a new object with a similar structure. It cannot be cloned, but such an operation could easily be added to the implementation. BETA does not support implicit delegation, but explicit delegation is possible, see e.g. Reference [MM92]. Finally BETA does not support operations for adding and removing attributes (slots in Self terminology). Such operations could be added to the language, but will in general require an incremental environment for handling type checking in order to be useful.

Pattern variables in BETA make classes, procedures, etc., first class, which means that they can passed around as objects.

It would be desirable if class and object could be completely unified and still provide the style of class-based programming that has proven its usefulness for modeling as well as implementation. This unification should also cover method and block.

6.   Concurrency and distribution

One of the most open issues in object-oriented programming is concurrency: should it be part of the language and if so, how should it be supported. In Simula any object can also act as a co-routine. This feature has been considered essential for modeling real-life systems consisting of a number of independently acting processes. Any Simula object may be an active object in the sense that it may have an associated set of imperatives execute as an active thread. Simula co-routines may be viewed as non-preemptive lightweight processes, in the sense that control transfer between active objects is only carried out when specific control operations are executed. On top of the basic mechanisms, Simula has various libraries including schedulers for management of such processes. As said in Reference [Mag93a], it is a great advantage that lightweight processes are built into the language, since it is then integrated with the memory management system (the programmer does not need to bother about stack sizes for processes).

There exists a number of different proposals for handling concurrency in object-oriented languages [Agh86,YT87,Cacm93] but the Simula approach has not been taken over by any main stream language. In Smalltalk and Self it is possible to start concurrent processes and synchronise these by means of semaphores, but using semaphores for synchronisation is very low-level.

BETA has a concept of active objects similar to Simula's, but in addition BETA objects may execute in true concurrency. Basically a concurrent BETA program may be viewed as a Simula program where a number of coroutines are executed concurrently. Since Simula only supports non-pre-emptive scheduling, there is no need for synchronisation mechanisms. In BETA synchronisation between processes may be achieved by means of semaphores. As mentioned above, semaphores are very low-level and should not be the only synchronisation mechanisms offered by a high-level language. In one of the first versions of BETA, synchronisation was based on a rendezvous mechanism similar to CSP and Ada. It was, however, realised that the rendezvous in many cases was not sufficient. There are many types of concurrent programs where monitors are much more elegant, just as there are many examples where neither monitor nor rendezvous are the right choice. The best proof of this is the very large number of different proposals for concurrency abstractions made in the literature.

For BETA it was decided to replace the rendezvous mechanism with semaphores. It may seem a step backward to introduce semaphores, especially in light of the criticism of Smalltalk and Self for using semaphores expressed above. The justification for doing this was that the semaphore is a simple and very general mechanism, but just as important that the BETA abstraction mechanisms makes it relatively easy to define abstract patterns that implements monitor, rendezvous and most other concurrency abstractions known from the literature. In most cases, a BETA programmer can use these high-level abstractions and will rarely have to use semaphores. In addition, the user of BETA is not tied to one specific

form for synchronisation, but can easily implements his/her own. The Mjølner BETA libraries offer a number of such concurrency abstractions.

The use of inner to implement concurrency abstractions like monitors in Simula, was proposed in Reference [Vau75]. The simplified version below shows the power of the BETA abstraction mechanisms. The example includes three objects: a bank account of a person (Joe), an object representing Joe, and one representing a bank agent:

```
(#  Account: ...;
    JoesAccount: @ Account;
    bankAgent: @ |
        (#
        do cycle(#do ...; 400->JoesAccount.deposit; ... #)
        #);
    Joe: @ |
        (#
        do cycle(#do ...; 150->JoesAccount.withDraw; ... #)
        #)
do  bankAgent.fork;{start concurrent execution of bankAgent}
    Joe.fork;       {start concurrent execution of Joe}
#)
```

The details of `Account` will be given later. The '`|`' following '`@`' in the declarations of `Joe` and `bankAgent` describe that these objects are active objects. The do-parts of these objects contain a `cycle` imperative that executes a list of actions forever. The `bankAgent` deposits money on Joe's account and Joe withdraws the money. Since the `bankAgent` and `Joe` execute concurrently, access to the account must be synchronised. The `Account` pattern is defined as follows:

```
Account: monitor
    (# balance: @integer;
       deposit: entry
           (# amount: @integer
           enter amount
           do balance+amount -> balance
           #);
       withdraw: entry
           (# amount: @integer
           enter amount
           do balance-amount -> balance
           #)
    #)
```

Account is a subclass of monitor and the operations `deposit` and `withdraw` are subpatterns of pattern `entry` defined within `monitor`. The definition of `monitor` and `entry` ensures that `Account` behaves like a monitor. Pattern `monitor` is defined in the following way:

```
monitor:
    (#  mutex: @semaphore;
        entry: (#do mutex.P; INNER; mutex.V #);
        init:< (#do mutex.V; INNER #)
    #)
```

A `monitor` object has a semaphore attribute, `mutex` that is controlled by the abstract superpattern `entry`. Any operation inheriting from `entry` can only execute its do-part (via `inner` in `entry`) if the `mutex` semaphore is not blocked. When the operation

completes it releases the semaphore after returning from `inner` in `entry`. Thus all details of the `monitor` implementation are hidden in the abstract `monitor` pattern. Recalling the discussion about `inner` and `super` in section 4.4 above, it is not possible to define such a monitor class using `super`. I.e. in languages like Smalltalk, Eiffel and C++ it is not possible to define a monitor class using this style.

In Reference [MMN93] a number of more elaborate examples of implementing concurrency abstractions are shown, including a complete implementation of a monitor (as in Reference [Vau75]) and Ada-like rendezvous.

The current release of the Mjølner BETA System does not implement true concurrency within a single process, but will eventually be supported. True concurrency is available via a framework for supporting distribution in BETA [BM94]. The framework supports distribution of a concurrent BETA program over a network of heterogeneous computers. The distribution framework puts certain requirements on the structure of such a distributed program, but distribution is almost orthogonal to the standard BETA language. Distribution is considered a means for organising the physical structure of a concurrent BETA program on a number of (physical) processors. This is analogous to the persistence library [BM94], which is considered a means for organizing BETA objects into transient and persistent objects. The persistent library supports persistence for any BETA object. On top of the persistence- library and distribution framework, an object-oriented database [GHMS94] for BETA has been implemented. In addition to a persistent store for BETA, there is support for concurrency control, transactions, and change notification. The BETA OODB and distribution have been used as a basis for implementing a distributed hypermedia system [GT94].

In section 2 it was said that in order to obtain the full benefit of object-orientation, one language should be used for analysis, design and implementation. The same is true for distribution, persistence, and data definition in object-oriented databases. For BETA it has been proved that it is possible to design a language and implement integrated support for analysis, design, implementation, distribution, persistence, and data definition.

7.   Conclusion

In this paper a number of open issues in object-oriented programming has been discussed:

- **Modeling versus reuse.** The main point here is that there has been a shift of focus from reuse of code to modeling and design. In order to obtain the full benefit of object-orientation there should be a balance between these extremes. In addition it has been argued that it is technically possible to use one language for analysis, design and implementation, including data modeling in object-oriented databases. This integration greatly simplifies the development process and make it possible to construct better tools.

- **Explicit conceptual framework.** In any phase of (object-oriented) development, it is important the developer is not restricted by the technical limitations of the language notation in use. During analysis and design, this may limit the possibilities for understanding the application domain. It is necessary that the developer makes use of a conceptual framework that is richer than the current languages. This may also make it easier for the developer to put up requirements for new languages.

- **Abstraction mechanisms**. In this part it has been argued that composition should be supported as well as classification. Aspects of single and multiple inheritance, multiple classifications and dynamic classification have been discussed. It has been shown that languages with strong support for block structure, part objects and singular objects offer certain alternatives to multiple inheritance. There have also been discussions of the pros and cons of inner versus super, various alternatives for supporting parameterised classes. issues related to typing including covariance and contravariance, and types versus classes.

- **Class-based versus prototype-based languages**. Prototype-based languages offer interesting alternatives to class-based languages with respect to modeling and technical possibilities. It has been argued that it should be possible to unite the two approaches. From a modeling point of view, the two alternatives seem to support different phases of the modeling activity. Technically there don't seem to be major differences in the sense that the new possibilities offered by prototype-based languages can be integrated with class- based languages. The real difference seems to be with respect to philosophy: some people may prefer a pure class-based approach whereas others may prefer a pure prototype-based approach.

- **Concurrency**. It has been argued that the Simula approach to active objects is still one of the best alternatives for supporting concurrency in object-oriented languages.

There are a number of issues that have not been covered in this paper, but where there also seems to be some disagreement. Examples are:

- **Multi-methods.** Some people argue that the unification of abstraction mechanisms into the concept of pattern as in BETA and the unification of class and object as in Self is going too far in the generalisation of programming language constructs. Multi-methods is another example of a generalization that may be unnecessary. They are definitely more general and more powerful than single-methods. Except for CLOS, multi-methods have not been adapted in practice. To the present author it is open whether or not multi-methods are really needed, since the few practical examples encountered can be handled by double dispatch.

- **Class versus module.** There seem to be some disagreement on to what extent class and module should be the same mechanism. Eiffel requires them to be the same and many other object-oriented languages do not have a special module mechanism. In C++ a module is really a file. In BETA there is a special language independent system for supporting modularisation [KMMN83b, MMN93]. In Reference [Szy92] it is argued that class and module are two different mechanisms. Self [Ung95] also has a modularisation mechanism that supports splitting the slots of an object into different modules.

- **Exceptions**. Each object-oriented languages has its own exception handling mechanism, but most of them are based on a dynamic approach similar to the one originally proposed in Reference [Goo75] and found in e.g. Ada. In BETA exception handling is based on a static approach defined using patterns [Knu84].

The following issues deserve more attention, since they may contribute to a qualitative improvement of object-oriented programming:

- **Reflection**. For a number of years, reflection [Smi84] has been a research issue, but it has had little impact on main-stream object-oriented languages. Notable exceptions are CLOS and Self.

- **Multiparadigm languages.** As discussed in Reference [MMN3] other programming perspectives like procedural, functional [Wik87] and constraint programming [FB92] have advantages for certain kinds of applications. It should be possible to combine all these perspectives into a unified perspective. The resulting language should not force a choice of one of the perspectives, but they should be integrated in a consistent way. BETA supports procedural programming and to some extent also functional programming. The Leda language [Bud95] is a new proposal for a language supporting a number of perspectives.

8. References

1. [DN66] O.J. Dahl, K. Nygaard: SIMULA, an ALGOL-based Simulation Language, Comm. ACM, 9(9), pp. 671-678, Sept. 1966.

2. [DNM68] O.J. Dahl, K. Nygaard, B. Myrhaug: Simula 67 Common Base Language, Technical Report Publ. no. S-2, Norwegian Computing Center, Oslo, 1968.

3. [DN81] K. Nygaard, O.J. Dahl: The History of the Simula Languages, in R.W. Wexelblat, (ed.), History of Programming Languages, Academic Press, New York, 1981.

4. [Mag94a] B. Magnusson: An Overview of Simula, in [KLMM94].

5. [KMMN83a] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Abstraction Mechanisms in the BETA programming Language, in Proc. POPL'83, Austin, TX, 1983.

6. [MMN93] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Object-Oriented Programming in the BETA programming Language, Addison Wesley/ACM Press, 1993.

7. [KLMM94] J.L. Knudsen, M. Löfgren, O.L. Madsen, B. Magnusson: Object-Oriented Environments – The Mjølner Approach, Prentice Hall, 1994.

8. [CY89] P. Coad, E.Yourdon: Object-Oriented Analysis, Prentice Hall, 1989.

9. [Boo91] G. Booch: Object-Oriented Design with Applications, Benjamin Cummings, 1991.

10. [RBPL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-Oriented Modeling and Design, Prentice-Hall, 1991.

11. [Mad94] O.L. Madsen: An Overview of BETA, in [KLMM94].

12. [GR83] A. Goldberg & D. Robson: Smalltak-80: The Language and its Implementation, Addison-Wesley Publishing Company, 1983.

13. [Str86] B. Stroustrup: The C++ Programming Language, Addison-Wesley, 1986.

14. [Sak89] M. Sakkinen: Disciplined Inheritance, ECOOP'89, Nottingham, England, BCS Workshop Series, Cambridge: CUP, 1989.

15. [ES94a] E. Sandvad: Hypertext in an Object-Oriented Programming Environment. In [KLMM94].

16.   [ES94b] E. Sandvad: An Object-Oriented CASE Tool, in [KLMM94].

17.   [Nau63]: P. Naur: Revised Report on the Algoritmic Language ALGOL 60 Comm. ACM, 6(1), pp. 1-17, Jan. 1963.

18.   [Wir71] N. Wirth: The Programming Language Pascal, Acta Informatica, 1, pp. 35-62, 1971.

19.   [GT94] K. Grønbœk, R. Trigg: Design Issues for a Dexter based Hypermedia System, Comm. ACM , 37(2), Feb. 1994.

20.   [Mey88] B. Meyer: Object-Oriented Software Construction, Prentice-Hall, 1988.

21.   [RC86] J. Rees, W. Clinger (eds.), Revised Report on the Algoritmic Language Scheme, MIT, TR No. 174, August 1986.

22.   [Kee89] S. E. Keene: Object-Oriented Programming in Common Lisp. Addison Wesley, 1989.

23.   [US87] D. Ungar, R.B. Smith: SELF - The Power of Simplicity, in Proc. OOPSLA'87, Orlando, FL, 1987.

24.   [BC87] E. Blake, S. Cook: On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk, in Proc. ECOOP'87, Springer LNCS Vol. 276, Paris, 1987.

25.   [MM92] O.L. Madsen, B. Møller-Pedersen: Part Objects and Their Location**Error!**

26.   [Kro85] S. Krogdahl: Multiple Inheritance in Simula-like Languages, BIT, 25, pp. 318-326,1985.

27.   [Str87] B. Stroustrup: Multiple Inheritance for C++ , in Proc. EUUUG Spring'87 Conference, Helsinki, Finland, 1987.

28.   [Øst90] K. Østerbye: Parts, Wholes and Sub-Classes, in Proc. European Simulation Multiconference, ISBN 0-911801-1, 1990.

29.   [Coo89] W. Cook: A Proposal for making Eiffel Type Safe, in Proc. ECOOP'89, Nottingham, England, BCS Workshop Series, Cambridge: CUP, 1989.

30.   [Bri75] P. Brinch-Hansen: The Programming Language Concurrent Pascal, IEEE Trans. Software Engineering, 1(2), 149-207,1975

31.   [KBR91] G. Kiczales, D.G. Bobrow, J. des Riviéres.: The Art of the Metaobject Protocol, MIT Press, 1991.

32.   [SU95] R.B. Smith, D. Ungar: Programming as an Experience: The Inspiration for Self, in Proc. ECOOP'95, Aarhus, Denmark, Springer LNCS, 1995.

33.   [APS93] O. Agesen, J. Palsberg , M. I. Schwartzbach: Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance, in Proc. ECOOP'93, pp.247-267, Kaiserslautern, Germany, Springer LNCS Vol. 707, 1993. Revised version to appear in Software Practice & Experience.

34.   [Mag94b] B. Magnusson: The Mjølner ORM System, in [KLMM94].

35.   [DS84] P. Deutsch, A.M. Schiffman: Efficient Implementation of the Smalltalk-80 System, Proceedings of the 11th Annual ACM SIGACT news-SIGPLAN Notices Symposium on the Principles of Programming Languages, Salt Lake City, Utah, January 1984.

36.   [Ung86] D. Ungar: The Design and Evaluation of a High Performance Smalltalk Processor. The MIT Press, 1986.

37.   [HCU91] U. Hölzle, C. Chambers, D. Ungar: Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches, in Proc. ECOOP'91, Geneva, Springer LNCS Vol. 512, 1991.

38.   [HU94] U. Hölzle, D. Ungar: Optimizing Dynamically-Dispatched Calls with Run-time Type Feedback, in Proc. SIGPLAN'94 Conference on Programming Language Design and Implementation, pp. 326-336, SIGPLAN Notices 29(6), June 1994.

39.   [AH95] O. Agesen, U. Hölzle: Type Feedback vs. Concrete Type Inference: Comparison of Optimization Techniques for Object-Oriented Languages, in Proc. OOPSLA'95, Austin TX, October 1995.

40.   [SCBW86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, C. Wilpot: An Introduction to Trellis/OWL, in Proc. OOPSLA'86, Portland, OR, 1986.

41.   [Nel91] G. Nelson: System Programming with Modula-3, Prentice Hall, 1991.

42.   [MMM90] O.L. Madsen, B. Magnusson, B. Møller-Pedersen: Strong Typing of Object-Oriented Languages Revisited, in Proc. OOPSLA'90, Ottawa, Canada, 1990, also in [KLMM94].

43.   [PS90] J. Palsberg , M. I. Schwartzbach: Type Substitution for Object-Oriented Programming, in Proc. OOPSLA'90, Ottawa, Canada, 1990.

44.   [Bor86] A. Borning: Classes versus Prototypes in Object-Oriented Languages, in Proc. ACM/IEEE Fall Joint Computer Conference, 1986.

45.   [Smi86] R. Smith: Experience with the Alternate Reality Kit, an Example of the Tension Between Literalism and Magic, in Proc. CHI + GI Conf., pp. 61-67, Toronto, 1987.

46.   [Lie86] H. Liebermann, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, in Proc. OOPSLA'86, Portland, OR, 1986.

47.   [DMC92] C. Dony, J. Malanfant, P. Cointe: Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and their Validation, in Proc. OOPSLA'92, pp. 201-217, Vancouver, Canada, 1992.

48.   [Mat81] L. Mathiassen: Systems Development and Systems Development Methods (in Danish), PhD Thesis, Institute of Informatics, Oslo University, 1981.

49.   [Weg83] P. Wegner: On the Unification of Data and Program Abstraction in Ada, in Proc. POPL'83, Austin, TX, 1983.

50.   [Cha92] C. Chambers: Object-Oriented Multi-Methods in Cecil, in Proc. ECOOP'92, Utrecht, Netherlands, Springer LNCS Vol. 615, 1992.

51.   [Agh86] G. Agha: Actors: a Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

52.   [YT87] A. Yonezawa, M. Tokoro (eds.): Object-Oriented Concurrent Programming, MIT Press, 1987.

53.   [Cacm93] Special issue on Concurrent Object-Oriented Programming, Comm. ACM, 36(9), Sept. 1993.

54.   [Vau75] J. Vaucher: Prefixed Procedures, a Structuring Concept for Operations, INFOR, 13(3), Oct. 1975.

55.   [BM94] S. Brandt, O.L. Madsen: Object-Oriented Distributed Programming in BETA., in Object-Based Distributed Programming (R. Guerraoui, O. Nierstrasz, M. Riveill, eds.), in Proc. ECOOP'93 Workshop, Kaiserslautern, Germany, July 1993, Springer LNCS Vol. 791, 1994.

56.   [GHMS94] K. Grønbœk, J, A. Hem, O.L. Madsen, L. Sloth: Cooperative Hypermedia Systems: A Dexter-Based-Architecture, Comm. ACM , 37(2), Feb. 1994.

57.   [KMMN83b] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Syntax-Directed Program Modularization, in Interactive Computing Systems (P. Degano, E. Sandewall, eds.), North-Holland, 1983.

58.   [Szy92] C. A. Szypersky: Import is not Inheritance - Why we need both Modules and Classes, in Proc. ECOOP'92, Utrecht, Netherlands, Springer LNCS Vol. 615 1992.

59.   [Ung95]: D. Ungar: Annotating Objects for Transport to Other Worlds, in Proc. OOPSLA'95, Austin, TX, 1995.

60.   [Goo75] J. B.Goodenough: Exception Handling: Issues and a Proposed Notation, Comm. ACM, 18(12), pp. 436-49,1975.

61.   [Knu84] J. L. Knudsen: Exception Handling - A Static Approach, Software Practice and Experience, 429-49, May 1984.

62.   [Smi84] B. Smith: Reflection and Semantics in Lisp, in Proc. POPL'84, Salt Lake City, Utah, 1984.

63.   [Wik86] A. Wikstrøm: Functional Programming Using Standard ML, Prentice-Hall, 1987.

64.   [FB92] B. N. Freeman-Benson, A. Borning: Integrating Constraints with an Object-Oriented Language, in Proc. ECOOP'92, Utrecht, Netherlands, Springer LNCS Vol. 615, 1992.

65.   [Bud95] T. Budd: Multiparadigm Programming in Leda, Addison Wesley, 1995.