

# A Conceptual Perspective on the Comparison of Object-Oriented Programming Languages \*

Bent Bruun Kristensen  
Kasper Østerbye

Aalborg University  
Institute for Electronic Systems  
Fredrik Bajers Vej 7, DK-9220 Aalborg Ø, Denmark  
e-mail: bbkristensen@iesd.auc.dk, kasper@iesd.auc.dk

## Abstract

The understanding of object-oriented programming languages is generally based on their included features. In this understanding, these features form the underlying concepts of both the languages and the modeling processes based on the languages. Consequently, object-oriented languages are generally compared and evaluated based on these features. Alternatively, object-oriented programming languages may be seen as supporting a conceptual perspective on programming. Instead of the features, the underlying concepts are then concepts such as *phenomenon* and *concept*, and also the abstraction processes in relation to these concepts. A comparison of object-oriented languages from the conceptual perspective gives additional understanding of each language and the mutual relations and differences between these.

## 1 Introduction

The paper presents a comparison of object-oriented programming languages using a conceptual framework. The framework is based on a conceptual perspective on programming and focuses on how and to what extent, the languages support various aspects of abstraction. As we shall see, even among the object-oriented languages, there are profound differences in their fundamental support for various aspects of the abstraction processes. The present framework is a further refinement of the initial framework presented in [Kristensen & Østerbye 94], where the focus is on the fundamentals of the conceptual programming perspective.

The following object-oriented languages are included in the comparison: Smalltalk (ST80) [Goldberg & Robson 83], Beta [Madsen et al. 93], C++ [Stroustrup 91], Eiffel [Meyer 92], CLOS [Keene 89], Self [Ungar & Smith 87] and [Agesen et al. 94], Objective C (O-C) [Cox & Novobilski 91] and [Pinson & Wiener 91], Ada [Dod 83], SIMULA [Dahl et al. 84], and Object Pascal (O-Pascal) [MPW 89]. In our comparison, the abstraction mechanisms of these languages are interpreted as examples of language mechanisms supporting conceptual understanding.

The use of our framework also gives a measure for the extent, to which a given programming language supports conceptual understanding. We need to distinguish between language support of some given concept, and simulation of the concept in some language: We shall use the definition in [Stroustrup 88]: “A language *supports* a programming style if it provides facilities that make it convenient (reasonable easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely *enables* programmers to use the technique.” In some situations we shall also distinguish between *provides support* and *enforces support*, where the meaning of *supports* is refined to respectively *allows you to use* and *forces you to use*.

A very general, classic characterization of object-oriented programming is given in [Stefik & Bobrow 86] simply by enumerating and discussing a number of features and their relations. A alternative, well-known feature-based characterization of object-oriented programming languages is given in [Wegner 87]: “object-oriented = objects + classes + inheritance”. In this definition the underlying concepts are the language features: *object*, *class*, and *inheritance*. These are examples on features underlying the general object-oriented understanding. Other examples of features can be found in feature-based comparisons such as [Saunders 89], [Blair et al. 91], [Booch 91], [Rumbaugh et al. 91], and [Floyd 93], e.g. encapsulation, operator overloading, garbage collection, meta data, typing. Such enumerations of features are very important, and deep insight in the languages and their differences can be obtained by evaluating and comparing languages in terms of these features. The combination of features, that are usually seen in connection with object-oriented languages, has proven to be very powerful. Still, the problem with this approach is, if these combinations of features are the only foundation of object-oriented programming, and no feature independent foundation is available also. A perspective is needed, because the perspective is fundamental for the programmer in the modeling activity; even very powerful features are insufficient here because they only contribute with isolated understanding of various aspects and do not cover the overall understanding of the modeling process. One promising perspective for object-oriented languages is *conceptual programming*. The following is a definition-like description of this perspective on the modeling process<sup>1 2</sup>:

---

\*This research was supported in part by the Danish Natural Science Research Council, No. 11-0061.

<sup>1</sup>This description must be further elaborated to be practical for the modeling activity. The methodologies for object-oriented analysis and design, such as [Booch 91] and [Rumbaugh et al. 91], may be seen as preliminary attempts to do this.

<sup>2</sup>In relation to the Beta language another similar definition has been proposed, where the emphasis is on the notion of “physical model”.

**Conceptual Programming:** *Programming is regarded as a modeling process of some referent system where the phenomena and abstractions from the referent system are expressed in a programming language supporting abstractions, which are based on a general understanding of phenomena and concepts.*

Our understanding of concept formation is based on phenomena and concepts. A concept has a denomination by which it is known, an extension, which is the set of all phenomena covered by the concept, and an intention, which is a description of the properties of the phenomena from the extension. The framework based on this understanding is organized as a number of aspects of conceptual abstraction:

- *The intention/extension relationship:* This aspect characterizes the perspective in determining if a phenomena is covered by a concept. We are interested in whether the properties are *mandatory* and must be possessed by all phenomena covered by the concept, or these are *characteristic*, – meaning that the properties do not have to be possessed by a phenomena, but typically are. We distinguish between the *Aristotelian* and the *prototypical* perspective on abstraction. The Aristotelian perspective is, that if a phenomena possesses all the mandatory properties of a concept, it is covered by that concept. To determine if a phenomenon is covered by a concept in the prototypical perspective, the phenomenon is compared with an existing phenomenon (the prototype), known to be covered by the concept.
- *The abstraction processes:* This aspect enumerates the abstraction processes supported and characterizes the different kinds of properties.

*Classification*, being the process of going from a phenomena to a concept, and the inverse *exemplification*, going from a concept to a concrete phenomena covered by that concept.

*Aggregation*, being the process of combining several concepts into a new concept. In this process we can look at the relationship between the properties of the aggregate and the properties of its parts. There are three different possibilities: *emerging*, meaning that the property is a new property of the aggregate, which is not a property of any (single) part, *hereditary*, meaning that a property of a part becomes a property of the aggregate as well, and *concealed*, meaning that the property of a part does not become a property of the aggregate. The inverse process is called *decomposition*.

*Specialization*, being the process of creating a more special concept on the basis of a general one. The extension of the specialized concept is contained as a subset of the extension of the general concept. We can distinguish between three relations between a property of the general and of the specialized concept: *inherited*, meaning that the property of the general is the same for the specialized concept, *modified*, meaning that the property is further elaborated in the special concept, and *additional*, meaning that the property is new to the special concept. *Generalization* is the inverse process of specialization.

- *The pragmatic aspects of abstraction processes:* For each of the abstraction processes the following pragmatic aspects are characterized: *run-time and/or at program-description time* for the abstraction process and *single or multiple order* of the result of the process.
- *The kinds of concepts:* This aspect characterizes the mechanisms of a language according to different kinds of concepts they support: *Thing* or *Action*, and how these may be combined.

In section section 2 we discuss which language mechanisms in the languages are interpreted as a model of a concept. The rest of the paper is organized according to the above listed aspects: The intention/extension relationship in section 3, the abstraction processes in section 4, the pragmatic aspects of abstraction processes in section 5, and the kinds of concepts in section 6. For each of these aspects we give an interpretation of the aspect in relation to object-oriented programming languages; we present a table showing our evaluation of the support of the aspect by the features of each of the languages; we add various remarks on the individual languages in relation to the evaluation; finally, we briefly discuss derived characterizations related to the aspect and present these results in additional tables. In section 7 we conclude from our experiences with the comparison.

## 2 Languages and Features

Abstraction mechanisms may support the *logical* and *physical* view of a system [Booch 91]. The logical mechanisms tend to be most important because these express the meaning of the description. The physical mechanisms are however necessary for the organization of the description in manageable pieces for different purposes. The two views have been mixed in the mechanisms in most languages throughout the history of programming languages and the properties of some given mechanism have not been presented clearly according to this distinction. Only the logical view is discussed here.

In order to describe how the object-oriented languages support the conceptual framework we have to discuss which abstraction mechanism in each of the languages is seen as a model of a concept. In most of the languages, classes model concepts and instances of classes (objects) model phenomena. In Beta, the model of a concept is the pattern. Self has prototypes only. Prototypes are objects and model phenomena and it is part of our comparison to discuss to which extent the prototype also models a concept. Various other abstraction mechanisms are available in the languages (some of which are included in section 6 describing different kinds of concepts):

**ST80.** Classes have *class-* and *instance-variables* and *methods*, including predefined methods (*primitives*).

**Beta.** Classes, methods, etc. are unified as *patterns*. Patterns may be nested.

**C++.** Classes have methods, called *members* and described as procedures and functions.

**Eiffel.** Classes have attributes and *features*. *assertions* in the form of *preconditions*, *postconditions*, and *invariants* are available for classes.

**CLOS.** Classes have methods, described as functions.

**Self.** No classes, but only objects are available. Any object can be used as a prototype for cloning. Variables and methods are unified as *slots*. Delegation combines inheritance and

instantiation.

**O-C.** Abstraction mechanisms are available in the form of the data-type *object* and the operation *message expression*. A method is a C function body. The notion of *protocol* may be seen as an alternative way of modeling concepts, where all objects responding to a protocol are in its extension.

**Ada.** The *package* is as an abstraction mechanism modeling an object and not a class. A package has local types, variables and procedures.<sup>3</sup>

**Simula.** Classes have procedures and functions. The methods of an object are the procedures local to the class.

**O-Pascal.** Classes are declared by an *object-construct*. Methods are procedures local to the class.

### 3 The Intention/Extension Relationship

The perspective on abstraction defines the relation between the intention and the extension of a concept. In the Aristotelian perspective the intention defines the extension. A phenomenon is covered by the concept if it has the properties defined in the intention. In the prototypical perspective each concept is characterized by a selected prototype phenomenon, which by definition is in the extension. The intention is the properties of the prototype.

Table 1 illustrates whether a language has the Aristotelian or prototypical (or possibly a combination of these) perspective in its support of abstraction. Also the possibility of having properties of different kinds such as *mandatory* (or *defining*) and *characteristic* is included in this aspect:

*Aristotelian or Prototypical Perspective:* The meaning of Aristotelian is interpreted to mean explicit description of concepts; class-like descriptions model concepts. Prototypical is interpreted to mean implicit description of concepts only. Object descriptions model phenomena, but can be seen as implicit descriptions of concepts.

*Mandatory or Characteristic Properties:* All properties described for a concept are interpreted as mandatory properties; all properties of implicit descriptions are interpreted as characteristic properties. In the languages with class-like descriptions, the objects have the properties described in the classes; in languages with prototypes the properties are given by the objects. (Instance) variables and methods (used generically to cover members, features, attributes etc.) model properties:

|             | ST80 | Beta | C++ | Eiffel | CLOS | Self | O-C | Ada | Simula | O-Pascal |
|-------------|------|------|-----|--------|------|------|-----|-----|--------|----------|
| Perspective | A    | A    | A   | A      | A    | P    | A   | P   | A      | A        |
| Properties  | M    | M    | M   | M      | M    | C    | M   | M   | M      | M        |

|             | Notation | Meaning                     |
|-------------|----------|-----------------------------|
| Perspective | A/P      | Aristotelian / Prototypical |
| Properties  | M,C      | Mandatory, Characteristic   |

Table 1: The Intention/Extension Relationship

**ST80.** Properties in the form of instance variables and methods are mandatory, and no object can be created which will not possess these properties. Methods can be overwritten, but the objects will always have the property, that they respond to this kind of message.

**Beta.** Properties in the form of attributes, enter/exit lists, and action part are mandatory. An action part cannot be overridden, but is extensible through the use of the *inner* construct.

**C++.** Properties in the form of data members and function members can be annotated as *private*, *protected*, or *public*. We could argue that private properties are not available in the specializations, and as such give support for characteristic properties. We have chosen not to do so, as private indicates

that they are not observable properties of the objects anyway. The same argument applies to private and protected inheritance.

**Eiffel.** Properties in the form of features can be redefined. This will, however, not remove the previous definition, but merely bind a new property to the same name (in the subclass). Features can also be renamed, which again does not remove the property; instead it is bound to a new name.

**CLOS.** Properties in the form of slots are mandatory, and cannot be made characteristic through specialization either. If a method is applicable for a general class, it will also be applicable for specialized classes, and as such act as a mandatory property.

**Self.** An object may be used as a prototype for cloning. The

<sup>3</sup>The description may be split into interface and body to support the physical view of the program. In addition the package may be seen as a mechanism supporting the physical view of the program by placing one or more "class"-definitions in a package in the form of type definitions together with corresponding procedure and function declarations.

properties are characteristic because objects, cloned from the same object, may respond to different messages and we can only tell which messages by actually sending messages to the object.

**Ada.** A package models an object; no cloning is possible and

the local declarations are fixed.

The *generic package* is a template for instantiating packages at program description time and implies explicit naming. Types and values are parameters to generic packages.

## 4 The Abstraction Processes

This aspect enumerates the abstraction processes, and the kind of properties of these processes, supported by the languages (Table 2 and Table 4) <sup>4</sup>.

### 4.1 Abstraction

This aspect enumerates which of the following abstraction processes are supported by the languages:

*Classification & Exemplification:* Classification is usually supported by an object being classified by the class from which it is instantiated; this classification of the object is kept throughout its lifetime. Exemplification may be supported either by the instantiation of an object given a class, or by returning the existing objects given the class. In addition in Table 3 we clarify the use of initial values as part of exemplification supported by instantiation.

*Specialization & Generalization:* Specialization is supported by various forms of inheritance. The inheritance mechanisms differ especially with respect to the modification of properties. In addition in Table 3 we clarify if the languages support the extensibility of all the kinds of properties. The kinds of properties may include not only variables and methods, but for example also a life-cycle part. If the extension of these is supported, it may be enforced or partially enforced only.

Looking at the result of specializing a (general) concept to a (more special) concept, the general concept is a generalization of the special concept. However, we do not consider the generalization process to be supported if it is available only implicitly as the result of specialization. None of the languages included in the comparison support generalization; this does not imply however, that it is impossible to design language constructs supporting this process; such constructs could exist together with specialization constructs, and as such support inverse processes, cf. [Pedersen 89].

*Aggregation & Decomposition:* Aggregation is usually supported only by means of instance variables. In addition in Table 3 we clarify if the languages support references to autonomous objects only or constituent (part) objects are supported also.

Looking at the result of aggregating (a set of part) concepts to a (whole) concept, the set of part concepts is a decomposition of the whole concept. However, we do not consider the decomposition process to be supported if it is available only implicitly as the result of aggregation. None of the languages support decomposition; this does not imply however, that it is impossible to design language constructs supporting this process.

| Processes       | ST80 | Beta | C++ | Eiffel | CLOS | Self | O-C | Ada | Simula | O-Pascal |
|-----------------|------|------|-----|--------|------|------|-----|-----|--------|----------|
| Classification  | Y    | Y    | Y   | Y      | Y    | N    | Y   | N   | Y      | Y        |
| Exemplification | Y    | Y    | Y   | Y      | Y    | -    | Y   | -   | Y      | Y        |
| Specialization  | Y    | Y    | Y   | Y      | Y    | Y    | Y   | -   | Y      | Y        |
| Generalization  | N    | N    | N   | N      | N    | N    | N   | -   | N      | N        |
| Aggregation     | Y    | Y    | Y   | Y      | Y    | Y    | Y   | -   | Y      | Y        |
| Decomposition   | N    | N    | N   | N      | N    | N    | N   | -   | N      | N        |

Table 2: The Abstraction Processes

**Beta.** For each of the abstractions unified by the pattern there may be explicit classification hierarchies; for this aspect we describe the pattern when used as class only. A pattern includes an *action part*, so that objects may be active, their life-cycle being defined by the action part. The *inner* construct is available.

**Eiffel.** *Features* are of two kinds: *routines* and *attributes*. Features may be renamed and redefined in subclasses. A rou-

tine may be *deferred*, i.e its implementation may be defined in a subclass (be *effected*).

**CLOS.** Methods are implementations of *generic functions*. Message passing is fundamental, and is also used to access variables. *closures* (blocks), and *procedures* (methods) are supported through prototypes of *activation record*.

**Self.** Prototyping combines inheritance and instantiation by the cloning operation and by *parent* references. New objects

<sup>4</sup>The following general notation is used:

| Notation | Meaning                                 |
|----------|---|
| Y        | Yes, it is supported (but not enforced) |
| (Y)      | Yes, but only partially supported       |
| N        | No, it is not supported                 |
| ?        | Is not (clearly) specified              |
| -        | Is not relevant                         |

are created by cloning an existing object used as a prototypical element.

Classification is not supported by Self because there is not a *single* prototypical element; therefore there is no denomination of a class and any object can be used as prototypical. Self only enables classification.

Specialization and aggregation are supported by Self because

the *single* prototypical element is not essential for these processes. For any object used as prototypical element the parent mechanism models specialization in relation to this object.

**Ada.** The (generic) package does not support general specialization.

**Simula.** A class may include a statement part, so that an object may act as a process.

### Exemplification.

*Initial Value:* In exemplification, as supported by instantiation, we can initialize the values (state) of the object <sup>5</sup>. The initialization may be supported by parameterization mechanisms. In ST80 initial values are supported by the use of the “new” method of classes. C++ has explicit initialization functions, *constructors*. In CLOS the initialization is supported by default initial values, *initform*, and initialization arguments, *initarg*.

It is important to distinguish between the parameterization with values (or state) as part of exemplification and other kinds of parameterization of generic abstraction mechanisms: In Beta a combination of specialization and instantiation of *singular* (one-of-a-kind) objects may involve the binding of virtual patterns by specialization. In Ada the instantiation of a generic package gives a package (modeling a singular object) and may involve values, types etc. as parameters.

### Aggregation.

*Action Part:* The action part may be aggregated from other action parts described as part of other concepts (for example as in Beta where *dynamic* or *inserted* objects may be used in the action part).

*Constituents:* As part of aggregation references may refer to autonomous independent objects or to constituent objects (dependent on and created as integral parts of the aggregated object, e.g *aggregate*'s in C++, *static* references in Beta, and *expanded* classes in Eiffel).

**Specialization.** References and action parts also model properties of a concept and may be included in the specialization of the concept.

*References:* References may have the object-classification extended (for example by *virtual* classes as in Beta). The fact that references in ST80 and CLOS are type-less is not considered to be support of extensibility of these properties; neither is the type-restriction possibility for instance variables in CLOS, because we only consider references.

*Action Part:* The action part may be extended (for example as in Beta and Simula by *inner* mechanism partially enforcing the extension and as in Beta by *virtual* classes used in the action part as *dynamic* or *inserted* objects).

| Processes       | ST80 | Beta | C++ | Eiffel | CLOS | Self | O-C | Ada | Simula | O-Pascal |
|-----------------|------|------|-----|--------|------|------|-----|-----|--------|----------|
| Exemplification |      |      |     |        |      |      |     |     |        |          |
| Initial Values  | (Y)  | N    | Y   | Y      | Y    | -    | N   | -   | N      | N        |
| Specialization  |      |      |     |        |      |      |     |     |        |          |
| References      | N    | Y    | N   | N      | N    | N    | N   | -   | N      | N        |
| Action Part     | -    | Y    | -   | -      | -    | -    | -   | -   | Y      | -        |
| Aggregation     |      |      |     |        |      |      |     |     |        |          |
| Constituents    | N    | Y    | Y   | Y      | N    | N    | N   | Y   | N      | N        |
| Action Part     | -    | Y    | -   | -      | -    | -    | -   | -   | N      | -        |

Table 3: The Abstraction Processes: Additional Characterization

## 4.2 Properties

This aspect enumerates in Table 4 which properties are supported in relation to specialization and aggregation. Among the mechanisms modeling various kinds of properties, only methods are discussed in the following:

**Specialization.** The following relationships between the properties are possible:

*Inherited:* This kind of property is usually supported directly by the basic inheritance mechanism of the language. Therefore, no additional language support is needed.

*Modified:* Modified properties are found in many forms. The interpretation of the term modified used here, is that the method may be either extended or redeclared. In addition in Table 5 we shall also discuss a more strict interpretation of a property being an extension of another property, namely that the extended property must have at least the same meaning as the property modified; this would imply that a redeclared property is not considered a modified property.

*Additional:* This kind of property is supported by simply declaring the new property. A restriction is that the property must have a unique name in the context in which it appears. A property may be marked as virtual, to explicitly indicate, that

<sup>5</sup>Exemplification is in this way related to specialization, where we can extend the set properties of concepts, and restrict the possible values of existing properties.

any declaration of this in a sub-class makes it a modified property; if a property is not marked virtual, its re-declaration is an additional property with an identical name.

**Aggregation.** The following relationships between the properties are possible:

*Emerging:* The property is specified as such only through a (new) declaration of it, where properties of selected parts may be combined to form the new property. Notice, that emerging properties do not emerge by themselves, but are explicitly described as properties of the aggregate.

*Hereditary:* A property is of this kind, only if it can be specified as such directly, i.e. the declaration of a new property, where the implementation just lifts a property of a part to the aggregate, is considered to be an emerging and not a hereditary property.

*Concealed:* This kind of property is usually supported by the basic visibility or accessibility rules of the language.

| Properties     | ST80 | Beta | C++ | Eiffel | CLOS | Self | O-C | Ada | Simula | O-Pascal |
|----------------|------|------|-----|--------|------|------|-----|-----|--------|----------|
| Specialization |      |      |     |        |      |      |     |     |        |          |
| Inherited      | Y    | Y    | Y   | Y      | Y    | Y    | Y   | -   | Y      | Y        |
| Modified       | Y    | Y    | Y   | Y      | Y    | Y    | Y   | -   | Y      | Y        |
| Additional     | Y    | Y    | Y   | Y      | Y    | Y    | Y   | -   | Y      | Y        |
| Aggregation    |      |      |     |        |      |      |     |     |        |          |
| Emerging       | Y    | Y    | Y   | Y      | Y    | Y    | Y   | Y   | Y      | Y        |
| Hereditary     | N    | N    | N   | N      | N    | N    | (N) | N   | N      | N        |
| Concealed      | Y    | Y    | Y   | Y      | Y    | Y    | Y   | Y   | Y      | Y        |

Table 4: Properties

**ST80.** *Selector* methods may be redefined by overriding. Super is available but no discipline is enforced.

**Beta.** References and (local) patterns may be overridden.

**C++.** Only *virtual* function can be redefined in derived classes; extension is possible by means of calling the inherited function directly. A virtual function must be defined, when it is declared unless it is declared to be a *pure* virtual function.

**Eiffel.** Any routine may be renamed or redefined. No special super- or inner-like mechanism is available.

**CLOS.** A generic function may be re-implemented as a method; fixed method combination according to a precedence list including *before*, *after*, and *primary* method is possible,

as well as programmer controlled combination by means of *around* methods.

**Self.** Slots may be overridden; no restrictions apply.

**O-C.** There is no direct language support for hereditary properties, but the use of automatic message forwarding can be said to be standard practice.

**Simula.** A procedure of a class may be *virtual*, so that it has no body, and may be specified in a subclass. Re-declaration (actually re-implementation) of a virtual procedure is only possible if its parameters and type are unchanged.

**O-Pascal.** The use of *inherited* corresponds to the use of super.

**Modified Properties.** We discuss the different possibilities of overriding, extension and combination:

*Overriding:* By overriding we mean that any method may be redeclared to replace an existing one.

*Extension:* Here the distinction is whether or not the languages force (possibly partially) extension or just support it. We consider virtual patterns (modeling methods) in Beta and the fixed method combination in CLOS to partially enforce the support of property extensions.

*Combination:* Finally we use the form of method combination available. This includes which class is in control over the combination, and also if the combination is declaratively controlled in the language. The use of “super” means that the more special class is in control whereas the use of “inner” means the more general class is in control; both these cases of combination are usually predefined, but other more advanced combinations may be available either predefined or as user-declarable.

## 5 Pragmatic Aspects of the Abstraction Processes

In Table 6 each of the processes is characterized by the following more pragmatic aspects:

- *Time:* The time of the use of process, i.e. dynamic (used at run-time) or static (used at program description time). We include the possibility that program description time can be interleaved with the application run-time, so that the program is changed during its execution.
- *Order:* The order of the products of the process, i.e. single (only) or multiple products of the abstraction process

*Classification:* By time we mean whether the object-class relation is given at program description time or whether it can be changed at run-time. In general object *evolution* means that the object may undergo some changes during its existence,

| Properties          | ST80 | Beta | C++ | Eiffel | CLOS | Self | O-C | Ada | Simula | O-Pascal |
|---------------------|------|------|-----|--------|------|------|-----|-----|--------|----------|
| Modified Properties |      |      |     |        |      |      |     |     |        |          |
| Overriding          | Y    | Y    | Y   | Y      | Y    | Y    | Y   | -   | Y      | Y        |
| Extension           | S    | (F)  | S   | S      | (F)  | S    | S   | -   | S      | S        |
| Combination         | S    | G    | S   | S      | D    | S    | S   | -   | G      | S        |

|             | Notation | Meaning                                   |
|-------------|----------|---|
| Overriding  | Y/N      | Yes: Allowed / No: Not allowed            |
| Extension   | S/(F)/F  | Supported / Partially Enforced / Enforced |
| Combination | S/G/D    | Special / General / Declaratively         |

Table 5: Properties: Additional Characterization

and then be dynamically classified according to its actual situation; this is usually simulated by exchanging the object with another object. In addition in Table 7 we illustrate typing in relation to implicit classification.

*Exemplification:* Single exemplification means that only one phenomena exists/can be created which is covered by the concept. This is usually unattractive, and can easily be confused with the idea of singular (one-of-a-kind) objects, that is, representations of individual unclassified phenomena. In general it may be difficult to tell whether we have a singular object, or a concept with only one instance; the difference is subtle.

*Specialization:* Single specialization is a special case only, namely, a concept with only one specialization. This should not be confused with specialization in connection with sular objects.

In multiple specialization the same concept is specialized to several concepts; this is the usual case. This is different from the effect obtained by various forms of multiple inheritance, where several (super) classes are combined to form a common subclass. Multiple inheritance may possibly be seen as supporting some kind of *simultaneous specialization* (although simultaneous specialization is not part our conceptual framework we include this possibility for completeness; alternatively, multiple inheritance may be seen as a support of the results of multiple generalizations, but not the processes). In addition in Table 7 we clarify the support of simultaneous specialization by means of multiple inheritance.

*Aggregation:* Multiple aggregation means that the same (set of part) concepts may be used to form several different aggregated concept.

| Processes       | ST80  | Beta | C++ | Eiffel | CLOS  | Self | O-C   | Ada | Simula | O-Pascal |
|-----------------|-------|------|-----|--------|-------|------|-------|-----|--------|----------|
| Classification  |       |      |     |        |       |      |       |     |        |          |
| Time            | S,D   | S    | S   | S      | S,D   | -    | S     | -   | S      | S        |
| Order           | S     | S    | S   | S      | S     | -    | S,(M) | -   | S      | S        |
| Exemplification |       |      |     |        |       |      |       |     |        |          |
| Time            | D     | S,D  | S,D | S,D    | S,D   | -    | D     | -   | D      | D        |
| Order           | (S),M | M    | M   | M      | (S),M | S    | M     | -   | M      | M        |
| Specialization  |       |      |     |        |       |      |       |     |        |          |
| Time            | S,D   | S    | S   | S      | S,D   | S,D  | S,D   | -   | S      | S        |
| Order           | M     | M    | M   | M      | M     | M    | M     | -   | M      | M        |
| Aggregation     |       |      |     |        |       |      |       |     |        |          |
| Time            | S,D   | S    | S   | S      | S,D   | D    | S     | S   | S      | S        |
| Order           | M     | M    | M   | M      | M     | M    | M     | M   | M      | M        |

|       | Notation | Meaning          |
|-------|----------|------------------|
| Time  | S,D      | Static, Dynamic  |
| Order | S,M      | Single, Multiple |

Table 6: Pragmatic Aspects

**ST80.** The method “changeClassToThatOf.” will change the class of an object dynamically. The method “allInstances” will return a list of all the instances of a given class; also the instances of subclasses may be obtained. New classes can be added at run-time. Single exemplification can be obtained by

changing the *new* method to check if an instance exists.

**Beta.** One-of-a-kind specializations, as a combination of specialization and instantiation is available, – also for patterns used as abstractions for actions.

**CLOS.** The method “change-class” will change the class of an object. The object is updated to the structure of the new class, which may involve deleting and adding slots. New classes may be added at run-time. A method will return a list of all the objects of a given class. Single exemplification can be obtained the same way as with Smalltalk.

**Self.** Changing a parent reference is seen as dynamic specialization.

### Implicit Classification.

*Typing:* A name (reference to an object)  $r$  may be bound to a class name  $C$  in its declaration. At run-time the reference  $r$  may be associated with an object from class  $C$ , the association may be static or dynamic, and several references may be associated with the same object. The reference  $r$  is said to be *qualified by* the class  $C$ , with the meaning that  $r$  is allowed to be associated with objects of  $C$  and that  $r$  is also qualified with any subclass of  $C$ . The result is that if  $p$  is a (mandatory) property of the concept modeled by  $C$ , then we know that  $r$  must have this property, and we may be allowed to access it, for example by  $r.p$ . In addition to this we may have the following alternative forms of correspondence between a reference  $r$  and the class  $C$ :

- References with a type (usually in the form of a protocol): A class  $C$  is used only as a common description of objects being instantiated from it, and not for classification. A reference may be associated with some object only if the object conforms with the (protocol of the) type  $T$ . The type  $T$  now works as some implicit, (usage dependent) classification of the object. In  $r.p$  the  $p$  must be in the protocol.
- References without qualification: Similar to the former case, except that now the protocol (for the use of the reference) is not explicitly defined by some declaration, but only implicitly by means of the actual use of the reference to access properties of the object, it is currently associated with. For the  $p$  in  $r.p$  there may or may not be a respond available from the object.

**Simultaneous Specialization.** In this aspect we clarify the support of simultaneous specialization by means of multiple inheritance.

*Duplication:* In multiple inheritance the inherited super classes may have a common superclass; if so, there is the possibility that this common class should be duplicated or not.

*Conflicts:* In both the above cases there may be name clashes, so that the same name may be inherited from several super classes; This may be allowed, and then (partially) resolved by the language, or it may be an error, and then be resolvable by the user.

| Processes                   | ST80        | Beta   | C++ | Eiffel | CLOS | Self | O-C | Ada | Simula | O-Pascal |
|-----------------------------|-------------|--|-----|--------|------|------|-----|-----|--------|----------|
| Implicit Classification     |             |  |     |        |      |      |     |     |        |          |
| Typing                      | N           | C  | C   | C      | N    | -    | N/T | -   | C      | C        |
| Simultaneous Specialization |             |  |     |        |      |      |     |     |        |          |
| Multiple Inheritance        | N           | N  | Y   | Y      | Y    | Y    | N   | -   | N      | N        |
| Duplication                 | -           | -  | D,U | U      | U    | D,U  | -   | -   | -      | -        |
| Conflicts                   | -           | -  | (A) | (A)    | (A)  | (A)  | -   | -   | -      | -        |
|                             | Notation    | Meaning  |     |        |      |      |     |     |        |          |
| Typing                      | N/C/T       | No typing / Classes as types / Types and classes                               |     |        |      |      |     |     |        |          |
| Duplication                 | D/U         | Duplication is allowed / Unique super classes only                             |     |        |      |      |     |     |        |          |
| Conflicts                   | A/(A)/(E)/E | Allowed, Resolved / Allowed, Partially Resolved / Error, User-Solvable / Error |     |        |      |      |     |     |        |          |

Table 7: Pragmatic Aspects: Additional Characterization

**ST80.** References are declared unqualified and type-less; an instance variable can be associated with any object.

**Beta.** References are qualified; a reference  $r$  declared in class  $D$  may be qualified by a *virtual* class  $V$ , meaning that in a subclass of  $C$ ,  $r$  may be further qualified to a subclass of  $V$ .

**C++.** References are qualified. Implicit and explicit conversion is possible.

**O-C.** In specialization, methods may be added and redeclared to replace existing methods. A super denotation is available. Multiple classification can be said to be supported if we view both classes and protocols as representing concepts. Dynamic specialization can be said to be supported through dynamic loading of new classes.

Per default the super classes inherited are copied and it is up to the subclass to resolve name clashes by means of *resolution*. Only one copy of the common *virtual* super classes is included.

**Eiffel.** References are qualified; possibly by a *deferred* class, describing an incompletely implemented abstraction.

Sharing under *repeated inheritance* is possible: If a feature has been redeclared on two separate paths a *select*-clause

must be included. If a feature is redefined only, this can only be solved either by re-declaration (by renaming) or by *undefining*. No duplication of superclass objects is possible.

**CLOS.** Slots may have a type. References are declared unqualified and type-less, and can be associated with any object.

Conflicts are resolved by rules as part of language definition, based on the class inheritance hierarchy, the order of the direct super classes, and a topological sort of the concrete inheritance lattice (error, if a conflict is unsolvable). No duplication of superclass objects is possible.

## 6 The Kinds of Concepts

This aspect characterizes the abstraction mechanisms according to a number of specialized kinds of concepts. As it is done in natural languages <sup>6</sup>, we can divide concepts into those that describe things (nouns), and those that describe actions (verbs). We can examine how things and actions are combined in concrete languages, if one of these kinds is subordinate of the other, and if the language allows more than one action to take place at a time. As illustrated in Table 8 we get the following aspects in relation to the kinds of concepts:

*Kind:* In general object-oriented languages support concepts of kind “thing” through classes and of kind “action” through methods. This aspect enumerates if the language supports both thing and action, and whether or not it actually differentiates between these. In addition in Table 9 we distinguish between various forms of program execution (in terms of the number of active threads in the execution). Finally in Table 9, we include the support of the special concept *concept* by means of meta classes or similar constructs.

*Dependence:* For all the languages with the dependence described as  $A < T$ , the actions, in the form of methods, must be specified with respect to a thing in the form of an object of some class.

|            | ST80 | Beta      | C++     | Eiffel | CLOS | Self | O-C   | Ada | Simula | O-Pascal |
|------------|------|-----------|---------|--------|------|------|-------|-----|--------|----------|
| Kind       | T,A  | N         | T,A     | T,A    | T,A  | T,A  | T,A   | T,A | T,A    | T,A      |
| Dependence | A<T  | I,A<T,T<A | A<T,(I) | A<T    | I    | A<T  | A<T,I | A<T | A<T    | A<T      |

|            | Notation | Meaning                            |
|------------|----------|------------------------------------|
| Kind       | T,A,N    | Thing, Action, No differentiation  |
| Dependence | I, X<Y   | Independent, X is subordinate to Y |

Table 8: Kinds of Concepts

**ST80.** All methods are defined as subordinate to classes.

**Beta.** The pattern mechanism is an abstraction over both kind thing and kind action. The differentiation between the pattern used to model a thing or an action is by convention only.

**C++.** A friend mechanism allows actions in the form of friend functions to be specified without any subordination relationship. However, the classes of the objects, that are passed as arguments, must agree to this friendship, so we have indicated this as a (I).

### Kind: Action.

*Combination:* We distinguish between *sequential* (one thread per program execution), *quasi-parallel*, (multiple independent threads but only one active thread per program execution), and *concurrent* languages (multiple active threads), cf. [Wegner 87]. Quasi-parallel execution is supported in Simula by objects being *coroutines*, and in ST80 by *processes*, and concurrent execution in Ada by *tasks*, and both these forms in Beta by *components*.

**Self.** An object can have multiple parents (because this is done explicitly by means of several *parent* objects, these may be identical as well as copies), and an error occurs if two slots are found in a message lookup; a conflict must be explicitly resolved by supplying a new method, that will delegate the message.

**O-C.** The protocols are seen as types. Multiple inheritance is only discussed in [Cox & Novobilski 91] (only predefined combination).

**CLOS.** Blocks are modeled by functions.

**Self.** The language does not enforce classification of objects into concepts of kind thing, but allows specification of methods.

**Ada.** The *task* models actions and supports concurrent execution: An *entry* is part of the interface to a task, and *accept* statements are implementations of an entry.

**Simula.** Objects interact as coroutines, to support a deterministic multi sequential execution.

<sup>6</sup>Our language is closely related to our thinking and understanding. To our knowledge the separation between nouns and verbs is fundamental. A few noticeable exceptions do, however, exist. In the language spoken in Greenland they do not distinguish between nouns and verbs: The verbs do not have the same active meaning as we are used to, but are passive and stationary like nouns. In the Indian language “Nootka” there are no nouns: There is only one word class and by the use of inflexion the words are associated with time and aspect. As an example a house is not a thing but it is an event, – it happens. The inflexion of the word indicates whether it is a prolonged or a momentary event. The language of the Hopi Indians have both nouns and verbs but the differentiation between these is done according to the duration: Lightning, wave and smoke cloud are examples of verbs because these are momentary events whereas cloud and storm are prolonged events and thus nouns.

**Kind: Concept.**

*Meta-Level & Class Methods:* The special concept *concept* has among others the properties: the *denomination* (some kind of name on the concept), the *extension* (the collection of phenomena covered by the concept), and the *intention* (the collection of properties for the concept). Thus this is the intention of *concept*. The denomination is *concept* and the extension are the various concepts (including itself).

We interpret *concept* to be supported by the notion of a metaclass-level in object-oriented languages. Meta classes enable us to interact with classes as objects. The metaclass hierarchy is seen as specializations of *concept* (the concepts *thing* and *action* may be such specializations of *concept*), and class methods (and variables) are seen as additional properties of these specializations. The pragmatic reason for supporting meta classes is meta programming, i.e. the ability to describe program manipulation directly and naturally in the language itself.

|                      | ST80 | Beta  | C++ | Eiffel | CLOS | Self | O-C | Ada | Simula | O-Pascal |
|----------------------|------|-------|-----|--------|------|------|-----|-----|--------|----------|
| Kind: Action         |      |       |     |        |      |      |     |     |        |          |
| Combination          | S,Q  | S,Q,C | S   | S      | S    | S    | S   | S,C | S,Q    | S        |
| Kind: <i>Concept</i> |      |       |     |        |      |      |     |     |        |          |
| Meta-level           | Y    | N     | N   | N      | Y    | N    | (Y) | N   | N      | N        |
| Class Methods        | Y    | N     | N   | N      | Y    | N    | Y   | N   | N      | N        |

|             | Notation | Meaning                                |
|-------------|----------|--|
| Combination | S,Q,C    | Sequential, Quasi-parallel, Concurrent |

Table 9: Kinds of Concepts: Additional Characterization

**ST80.** The instances of *meta classes* are classes; any class has a metaclass. A metaclass is an instance of the class *Metaclass*. *Class* is an abstract superclass for all meta classes; *Object* is an abstract superclass of all classes. By using objects of these predefined classes, other classes may be created and changed at run-time.

**CLOS.** *standard-class* is a predefined metaclass and *standard-*

*method* and *standard-generic-function* are predefined meta objects. New meta classes may be defined, and new meta object classes may be defined to be the classes of generic functions or methods.

**O-C.** A class is implemented as a *factory object*. Variables and methods are available for the class (factory variables and methods).

## 7 Conclusion

We have interpreted and further elaborated a framework based on a conceptual perspective on object-oriented languages in order to make a comparison of these. The main result of the comparison is an alternative understanding of these languages, their similarities and differences, from this perspective. We observe that

- The framework is intended to be used in a comparison restricted to the conceptual perspective only, and it is not just the starting point for a comparison, where any feature, not related to the framework, also can be considered. This implies, that there are important features of the languages, which have not been included, both for the languages in general, and for some specific feature of some specific language (e.g. the *message* mechanism of Smalltalk and Self, the *assertions* of Eiffel, and the *multi method* of CLOS). Examples of general features usually related to object-oriented languages, but not covered by the conceptual perspective, are: encapsulation, modularity, operator overloading and polymorphism, garbage collection, typing, persistence, assertions and constraints.
- The framework may still not be complete and could possibly also cover additional features, usually identified with object-oriented languages. An example of such a feature is *localization of descriptions*: To be able to describe one concept in the context of another. In programming languages this is usually supported by block structure. In Beta, patterns can be nested where the meaning of this is not only related to visibility.
- Although we may make the framework more complete by covering identified features, we find it more important however, that the framework has uncovered important modeling issues *not* identified in any existing language. Some examples of this are the hereditary properties of aggregation, and the insufficient support for concepts of kind *action*, especially with regard to specialization and aggregation of actions. We believe that further elaborations of the framework will uncover other missing features. As an example, we could imagine that the understanding of aggregation and specialization was elaborated to make clear the various ways in which these abstraction processes can take place. Also, we might develop our understanding of the different *kinds* of concepts.

We have chosen to use the definitions of “support” and “enable” as introduced in [Stroustrup 88] for simplicity reasons. From our experience with the comparison we are not completely satisfied with these, and we have felt the need for a more refined understanding of support. The following distinction appears to be more appropriate:

- *Enforce*: The language forces the programmer to use the concept. Example: Eiffel enforces actions to be subordinate to things.
- *Encourage*: The language provides convenient mechanisms to express the concept, but the programmer can choose not to do so. Example: C++ encourages the use of classes.
- *Enable*: The language does not have any language mechanisms to express the concept, but the programmer can easily establish a convention of using other mechanisms for the same purpose. Example: Beta enables the distinction between things and actions.
- *Discourage*: The language does not contain any language mechanisms to support the concept, and it will take extraordinary skills or great discipline to establish a simulation of the concept. Example: Smalltalk discourages simultaneous specialization, but it can be obtained if the programmer possesses the right skills (cf. [Borning & Ingalls 82]).
- *Prohibit*: The language semantics is such that any attempt to use the concept are hindered by the language. Example: simultaneous specialization is prohibited by BETA.

The use of this more refined understanding could have given a more refined result. As an example, in relation to the language Self, we may, with the use of this terminology, claim that Self enables concepts etc., whereas Self does not encourage the use of these. In the use of both definitions, it has been our experience that we cannot be exact about when a language supports a concept or not; on the other hand, it has been our experience that each decision on a Yes or a No (or similar) in the tables has elaborated our understanding of the languages, and as such the framework is very successful – as a framework for discussing languages – without being based on formal semantics, nor falling into the Turing tar pit (that all languages are equal in expressive power).

As expected none of the languages appear to be designed from the conceptual programming perspective. Instead they are designed from the current understanding, and as incremental innovations, of what has come to be known as object-oriented programming. But they support conceptual understanding to some extent. In the comparison process where the main task has been to interpret conceptual understanding in relation to the features of the languages, we have achieved a better understanding of the problems with, and the incompleteness of, our current definition of the conceptual perspective, basically because our definitions have been thoroughly tested by this interpretation:

- The languages are “not quite Aristotelian”:

There is always a well-defined relation between a class and an object; it may change but it is always there. There is no support for an objective determination of whether an object has some properties; it is born with the properties given by the class definition.

There is no support for characteristic properties; and because any method (modeling a property) can be overridden the support of mandatory properties is very primitive and insufficient.

- The languages are “not quite Prototypical”:

They have objects but no classes; because the object *models* a phenomenon we have some kind of abstraction (sometimes referred to as *representation abstraction*).

There is no distinguished prototypical element as required by the definition; therefore, we have decided in our interpretation that there is no classification, since this is exactly what the prototypical element should be used for.

At the object level there is partial support of specialization and aggregation by language constructs such as respectively delegation and slots. When cloning an object, the object may be seen as modeling a class and in this sense there is (partial) support for specialization and aggregation.

- The notion of property needs to be further developed:

In the discussion of the information process in [Nygaard 86] various definitions, in relation to the understanding of the modeling of phenomena in nature and society, are given. This includes the important distinction between measurable properties of substance and transformations of substance.

We still need to understand what properties really are, and what kind of abstraction processes are available for these: Can properties for example be classified?

- Generalization and specialization (and similarly for aggregation and decomposition) are inverse processes:

If so, it may still be valuable to have both available in a language; how can we support the fact that they actually are inverse processes? And, if for some reason it turns out that they are more than just inverse processes, what are their differences then and how can these be supported by language constructs?

- For some aspects supported by language constructs we have only a very limited and implicit understanding of these in the conceptual perspective, if any:

Has the duplication of super classes (for example in Eiffel) any meaning in the conceptual perspective? Or is this really a matter of aggregation in this perspective?

How can we be more specific about the meaning of life cycle of objects in the conceptual perspective?

How can we be more specific about the meaning of meta classes in the conceptual perspective?

To get a graphical illustration of the relative differences between the languages we introduce a metric <sup>7</sup> on the similarities and differences of the languages as given by all the tables in this paper. Figure 1 shows the result as relative differences between the languages given in %. From the Figure we notice that

- None of the languages appear to be identical according to the characterization obtained by using the framework. As expected, Ada (and also Self to some extent) appears to be very different from all the other languages included in the comparison.
- Some languages appear to be very similar according to the characterization, although these languages are found to be very different in a more pragmatic understanding of the languages such as Eiffel / C++ (and also ST80 / CLOS to some extent). As expected, languages such as Simula / Object Pascal (and also Beta / Simula and C++ / Objective C) appear to be similar.
- Some languages appear to be very different according to the characterization, although these languages are design from similar principles, such as Self / Beta, which both have aimed to unify traditional abstraction mechanisms. Despite similar design principles, languages such as Self / CLOS (and also C++ / Smalltalk to some extent) appear to be somewhat different.

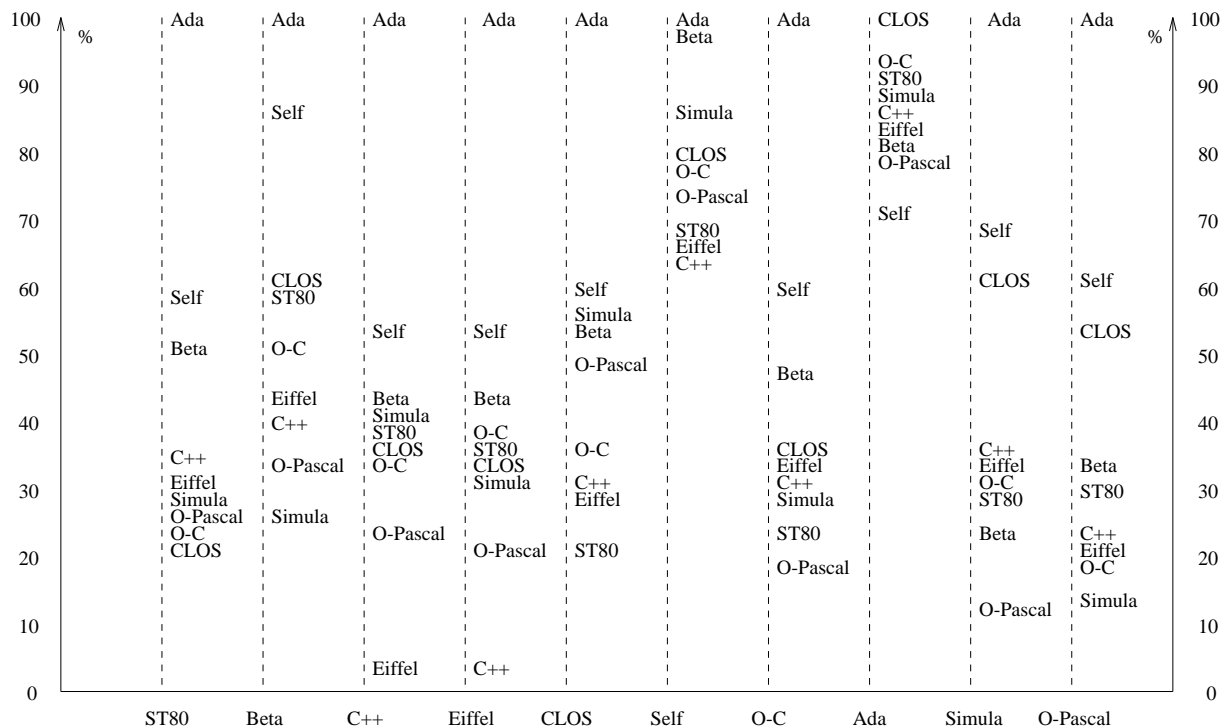


Figure 1: Illustration of the relative differences between the languages. For each language the column shows the distance to each of the other languages in %; the higher the number is, the bigger is the difference between the languages.

**Acknowledgment.** We thank Ole Agesen, Birger Møller-Pedersen, Jeppe Sommer, and Charlotte P. Lunau for inspiring discussions and constructive critique in relation to the characterization of the languages included in the comparison.

## References

[Agesen et al. 94] O. Agesen, L. Bak, C. Chambers, B. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, M. Wolczko: How to use Self 3.0 & The Self 3.0 Programmer's Reference Manual. Sun Microsystems Laboratories, 1994.

[Blair et al. 91] G. Blair, J. Gallagher, D. Hutchison, D. Shepherd: Object-Oriented Languages, Systems and Applications. Pitman 1991.

<sup>7</sup>We compare the languages in pairs for each table and accumulate the results. For any table we compare the characterization of any two languages. If the entries are identical the result is 0, if overlapping: 1, and if disjoint: 2. Markings in parentheses are counted without these, so that (Y) is counted as Y. Remark that the aspects of the table has the same weight, except that because some tables are refinements of aspects of other tables these aspects may implicitly have more weight.

- [Booch 91] G. Booch: Object Oriented Design with Applications. Benjamin/Cummings 1991.
- [Borning & Ingalls 82] A. H. Borning, D. H. H. Ingalls: Multiple Inheritance in Smalltalk-80. Proceedings of the National Conference on Artificial Intelligence, AAAI-82.
- [Cox & Novobilski 91] B. J. Cox, A. J. Novobilski: Object-Oriented Programming, An Evolutionary Approach. 2/E Addison-Wesley 1991.
- [Dahl et al. 84] O. J. Dahl, B. Myhrhaug, K. Nygaard: SIMULA 67 Common Base Language. Norwegian Computing Center, edition February 1984.
- [Dod 83] Reference Manual for the Ada Programming Language. United States Department of Defense. ANSI standard Ada. 1983.
- [Goldberg & Robson 83] A. Goldberg, D. Robson: Smalltalk 80: The Language and its Implementation. Addison Wesley 1983.
- [Floyd 93] M. Floyd: Comparing Object-oriented Languages. Dr. Dobb's Journal, 1993.
- [Keene 89] S. E. Keene: Object-Oriented Programming in Common Lisp. Addison Wesley, 1989.
- [Kristensen & Østerbye 94] B. B. Kristensen, K. Østerbye: Conceptual Modeling and Programming Languages. Sigplan Notices, 29 (9), 1994.
- [Madsen et al. 93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: Object Oriented Programming in the Beta Programming Language. Addison Wesley 1993.
- [Meyer 92] B. Meyer: Eiffel, The Language. Prentice Hall, 1992.
- [MPW 89] Macintosh Programmer's Workbench Pascal 3.0 Reference. Apple Computer, 1989.
- [Nygaard 86] K. Nygaard: Basic Concepts in Object Oriented Programming. Sigplan Notices, 21 (10), 1986.
- [Pedersen 89] C. H. Pedersen: Extending Ordinary Inheritance Schemes to include Generalization. Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications, 1989.
- [Pinson & Wiener 91] L. J. Pinson, R. S. Wiener: Objective-C, Object-Oriented Programming Techniques. Addison-Wesley 1991.
- [Rumbaugh et al. 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson: Object-Oriented Modeling and Design. Prentice-Hall 1991.
- [Saunders 89] J. Saunders: A Survey of Object-Oriented Programming Languages. Journal of Object-Oriented Programming, 1989.
- [Stefik & Bobrow 86] M. Stefik, D. G. Bobrow: Object-Oriented Programming: Themes and Variations. The AI Magazine, 6(4), 1986.
- [Stroustrup 88] B. Stroustrup: What is object-oriented programming ?. IEEE Software, 1988.
- [Stroustrup 91] B. Stroustrup: The C++ Programming Language. 2/E, Addison-Wesley 1991.
- [Ungar & Smith 87] D. Ungar, R. B. Smith: Self: The Power of Simplicity. OOPSLA'87.
- [Wegner 87] P. Wegner: Dimensions of Object-Based Language Design. Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications, 1987.