Conceptual Modeling and Programming Languages

Bent Bruun Kristensen & Kasper Østerbye
Aalborg University *

January 1994

Abstract

Programming is a modeling process where phenomena and abstractions from a referent system are expressed using a programming language. To improve the efficiency of program development, programming languages should be designed to reflect human conceptualization as well as possible. This will make programs easier to comprehend, thus aiding during both development and maintenance, and it will aid in making the developed programs conform better to the users expectations. This paper will present a model of the interactions between epistemology, concept formation, programming language analysis and design, and programming. The model will be used to develop a taxonomic framework for analyzing and designing abstraction mechanisms found in object-oriented languages.
keywords: object oriented modeling, language design, language comparision, programming paradigms

1 Introduction

This paper is an attempt to define and illustrate an approach to programming which we call *conceptual programming*. The main goal is to develop a framework for the design of programming languages that will reduce the gab between analysis, design and implementation, while improving the analysis and design phases. We do this by examining the interrelationship between epistemological models, their influence on concept formation, how they can be supported in programming languages, and how the concept formation can be expressed in a programming language.

By basing programming language design on an epistemological foundation, we believe that we can improve the programming process in two major ways. Firstly, by stating explicitly how we see human conceptualization, the concept formation used during the concept formation taking place during the analysis and design phases will be improved, which will lead to programs which conforms better to the users expectations. Secondly, by having the programming language support the same abstraction mechanisms as used during early concept formation, the developed programs will be easier to comprehend, as the programmer will not have to deal two different models of abstraction. By letting the abstraction mechanisms be based on a understanding of human conceptualization, rather than say a mathematical foundation as e.g. in functional programming, we believe that programs will be easier to understand.

In section 2 we will take a closer look at our approach to programming, and we will present a modeling diagram which forms the basis for our understanding of the interrelationship between our understanding of concepts, concept formation, language design, and programming; and we will further argue why we believe the approach will fulfill its objectives. The relationship between our work and object-oriented analysis and design is also addressed, the main similarity being the emphasis on abstraction mechanisms which goes beyond current programming languages.

We then proceed to present our understanding of conceptualization. In section 3 we present a general model for abstraction. The abstraction mechanism are *classification* of phenomena into concepts, *generalization* of specific concepts into more general ones, and *aggregation* of simpler concepts into complex concepts. For all types of composition, we take a close look at how the intentional properties of the constituent concepts affect the composed concept.

Ideally, we would like to present a programming language *designed* from the outset using our approach. However, we need a better understanding and more practical experiments in order to come up with a language essentially different from existing languages. Hence it is for the moment more profitable to come up with proposals for separate language mechanisms rather than entire languages, as illustrated in our papers [Østerbye, 1990; Kristensen, 1993b; Kristensen, 1993a; Kristensen, 1994; Østerbye and Kristensen, 1994].

Instead we will illustrate our approach by an example of the use our model for language *analysis*. In section 4 we take a look at some existing concepts from programming languages in the perspective of how these support abstraction. In section 5 we expand further on the idea of using our model for language analysis by developing a taxonomic framework of abstraction mechanisms found in object-oriented programming languages. The major contribution from that section is an attempt to identify some of the pragmatic aspects and choices that arise in concrete programming languages, and to identify them as aspects of the concept understanding described in section 3.

We round of with a section where we summarize the approach, and give some directions for future work.

Background

Any program is a model of something. This something may be a part of either the real or the imaginary world. Any language is based on some selected set of concepts. The choice of this set of concepts determines the fundamental characteristics of the model. The choice is therefore very important for the language and thus for the user of the language because it defines the underlying perspective for any programming activity in the language.

The choice of concepts in e.g. *functional programming* is the mathematical function and the associated theory whereas, in *conceptual modeling* the choice is concepts such as phenomena, concept and various others associated with these, including mappings of abstraction such as specialization and aggre-

*Fredrik Bajers Vej 7, DK-9220 Aalborg Ø, Denmark (+45 98158522), (bbkristensen@iesd.auc.dk). (kasper@iesd.auc.dk).

gation.

The paradigm that are closest to our is the object–oriented paradigm as defined in connection with the Beta language [Madsen *et al.*, 1993]:

- Object–oriented Programming: A program execution is regarded as a physical model simulating the behavior of either a real or an imaginary part of the world. Physical modeling is based upon the conception of reality in terms of phenomena and concepts.

This definition of object–oriented programming is not generally accepted in the object–oriented community. We believe that we share the intent behind the above definition, but would like to change it in three ways. Firstly, we want to avoid the term object–oriented, as we believe that it could soon turn into a straight-jacket, secondly, we would like our definition to say something about programm*ing* rather than program executions, and finally we would like to get rid of the word *physical* as we find it misleading. We then come to the following definition:

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

This definition can be contrasted to the definition of an object–oriented language given by P.Wegner in [Wegner, 1987]:

- object–oriented = objects + classes + inheritance

One problem with this definition is that is based on an enumeration of (selected) language features; admittedly in this case this combination has proven to be powerful, but still it is only features and in this sense just as insufficient to define a programming perspective as e.g. an enumeration based on the features lexical scoping, recursive procedures and lazy parameter evaluation. Another problem with these features is what the meaning of them are. The meaning has been incrementally defined during the history of object–oriented programming and the meaning is not clear at all, of course we may find some very general definition of these but then the definition is almost empty, and still without any foundation.

2 An approach to Programming

Conceptual modeling denotes an understanding and construction process with the purpose of forming a model of some part of the world expressed by concepts and phenomena, i.e. a concept formation process. The purpose of the model may either be the understanding of some existing system or the design of some new model system. The process is based on the observation and identification of various phenomena and the conceptualization of these in the form of descriptions of concepts capturing properties of the phenomena. The observation and identification activities are influenced by some chosen perspective of the process, usually given as part of the overall purpose of the model. The perspective controls the abstraction involved in the determination of the actual concepts.

The purpose of this paper is to discuss an approach to programming involving concept formation and language design based on conceptual understanding. The elements of this approach is given by the *modeling diagram* illustrated in figure 1.

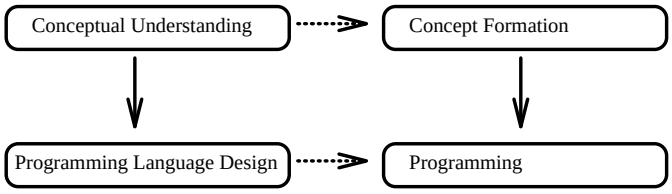


Figure 1: Modeling Diagram

2.1 Modeling Diagram

The modeling diagram contains the following elements:

Conceptual Understanding: Conceptual understanding means that we build models of concepts, phenomena, abstraction etc, in order to understand the fundamental characteristics of these in general.

The assumption is that human recognition is based on concepts and phenomena: The natural way for human beings to understand and communicate about parts of the world is by identifying relevant components and building an abstract model for characterizing the components and the relations between them. General models of this form of recognition can be developed. Such models are catching the essence conceptual understanding. The models may be accompanied by informal conceptual languages.

Concept Formation: Concept formation means the construction of a *conceptual model* for some specific application area in terms of concepts and phenomena from the area. This model may be expressed in some informal conceptual language.

The assumption is that specific, complete but informal models can be described using informal conceptual languages for concept formation. A model is conceptual if it focuses on the concepts and phenomena in the observed system in order to describe these directly in the model system.

Programming Language Design: Programming languages may be (analyzed and) designed based on conceptual understanding. To some extent this has already been done in the case of object–oriented languages. Intentionally or not the mechanisms of these languages, especially the class and the object, support the modeling of concepts and phenomena. Languages in the style of object–oriented languages, but designed directly from conceptual understanding, are not yet available.

The assumption is that programming languages, i.e. formal languages, can be designed based on conceptual understanding.

Programming (with arrows from both concept formation and programming language design): A program is a formal model expressed in a programming language to be designed based on conceptual understanding. The program may or may not be refined or transformed from a conceptual model. Anyway an informal conceptual model may exist.

The assumption is that specific, complete and formal models can be described using such programming languages. And that the distance between the conceptual model and the program will be short.

2.2 Efficiency

The most important of the expected results of a more extensive use of conceptual modeling in the program development process is *efficiency*: By efficiency we mean a measure for the time used in the program development process, i.e. we are not concerned with time requirements for the program execution. We are especially thinking of the development and maintenance phases of the program development process. The efficiency obtained is mainly due to the *comprehensiveness* and *conformance* obtained. We claim that the development process is typically inefficient because – the program turns out not to conform to its intention mainly due to communication problems between the user and the programmer – and the program is not comprehensible due to the difference between the understanding directly obtainable from the program and the intention of the program:

- *Comprehensiveness:* By the comprehensiveness of a program we mean to what extent the program is understandable compared to the understanding obtainable from the conceptual model (or from some other description of the vision of the system). In the modeling diagram a program corresponds closely to the conceptual model. The program is constructed based on the conceptual model and the programming language is designed based on the conceptual understanding, – hence the program is very likely to be highly comprehensible. This claim covers both the development and maintenance phases of program development, i.e. both the situation where we are creating the program and have full control and insight – and the situation where we may not have been involved in the creation of the program but have to examine the program in order to make some corrections or improvements.

- *Conformance:* By conformance of a system we mean to what extent the system actually developed corresponds to the expectations of the users. The program is supposed to solve some problem for the user in some given application area. The user has some (usually unclear) expectations to the program. A well–known problem is how the user can express his expectations and thus communicate these to the programmer. Also it is a problem how the programmer should express his understanding of the problem and its solution to the user without using the actual program description. A conceptual model is a very promising candidate for common understanding and successful communication: It is close to the users understanding of the reality, the language used is close to his implicitly used language for usual understanding of and thinking about the problem. It is informal as well as there are no technical confusing details in the description. It is also close to the programmers tool for expressing the model, namely the programming language. It is a complete description and it is not completely informal without any structure. The claim is that the conceptual model may ensure a very high degree of conformance.

2.3 Analysis, Design & Implementation

The concept formation process from our conceptual diagram is related to object–oriented *analysis, -design, and -implementation*. The methodologies for analysis and design based on the object–oriented perspective on programming can be characterized as follows:

- The mechanisms from the object–oriented languages are the basic features of the notation of the methodologies.
- The notation of the methodologies is to a greater extent based on conceptual understanding than the programming languages and are adding more advanced conceptual mechanisms.
- The processes of the methodologies are very important for the use of the object–oriented programming languages in the future.

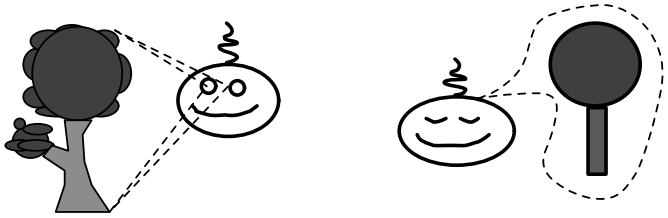
There seems to be no conflict between the object–oriented analysis and design methodologies and object–oriented programming languages as such, rather it appears that there are very promising mutual interactions between these with conceptual understanding as a common underlying perspective.

We understand the term object–oriented *programming* to include the modeling activities and products of the analysis and design phases: Analysis, design, and implementation is just an important organization in phases of the programming process. This means that we see the concept formation box of our conceptual diagram to be a part of any of these phases.

3 Conceptual Understanding

The purpose of conceptual understanding is to be able to build models of subparts of the world. We assume that phenomena exist. We can identify phenomena and we can observe their behavior and identify some properties of the phenomena. We assume that concepts are created as the result of model building. Concepts are part of mental models that we build in order to understand and communicate about a part of the world, i.e. its phenomena. Phenomena are collective, shared among all humans. Concepts are individual, existing only as mental models. An important aspect of conceptual understanding is how to make concepts collective.

A phenomenon is collective, but the identification of the various phenomena and the observation and measuring of a phenomenon is individual. This process is influenced by background, experience etc. And more important it is controlled by the perspective of the observer. The purpose of the modeling, i.e. the intended use of the model system, highly influences the perspective. In general it is neither possible nor desirable to make a complete model. We want intentionally to include certain aspects in our model and exclude other aspects. So we focus on certain aspects and the focus is controlled by the perspective.



Abstract Modeling

A set of related concepts forms some understanding of a part of the world from a given perspective. Concepts are related to phenomena and to other concepts. A concept is an abstraction of phenomena or concepts. The set of concepts forms an abstract model of the phenomena involved. As such the model may be used for understanding of – and communication about – this part. The model may be utilized to create artificial phenomena representing the phenomena of the real world. The artificial phenomena are abstract versions of the real phenomena. I.e. we may use the model as a prescription of a model system with active, interacting artificial phenomena. The model system may be processed independent from the reality, i.e. as a simulation. E.g. a simulation of a crossroad to measure how long the lines of waiting cars are, given some traffic intensity. Or the model system may be processed as part of the reality in an highly interacting manner, i.e. as most software systems do. E.g. a payroll system modeling among others selected properties of the people hired.

Concepts are formed mentally. The question is how we describe concepts. If we can describe concepts then concepts become collective and we may even create model systems automatically. To describe concepts we must know more about what a concept is. We have to build a model of concepts. In this modeling we have no special perspective other than we want to be able to make a description of the concepts. And we are looking for an abstract, but complete model.

3.1 A Model of Concepts and Phenomena

We discuss the following important example of a model for concepts. A concept is composed of the parts:

- The *denomination* of the concept: The name of the concept – or some other way to refer to the concept (e.g. by means of some drawing).
- The *extension* of the concept: The collection of phenomena covered by the concept. E.g. the extension of the concept Dog is all the animals we think of as being dogs (e.g. the Fido living at our neighbors, Balder being the dog of the Queen of Denmark, the famous Lassie, etc).
- The *intention* of the concept: The collection of properties that the phenomena in the extension of the concept have in common (e.g. dogs Bark, dogs have four Legs etc).

The denomination usually gives no problems. The controversial question is how to decide whether or not some phenomenon is covered by some concept. We can discuss both what is most correct with respect to how people actually do this and what is most appropriate for the purpose of actually coming up with some language for making descriptions of concepts. For this purpose properties we distinguish between mandatory or characteristic properties.

Mandatory and Characteristic Properties

We define mandatory (or defining) and characteristic properties as follows:

- A *mandatory* (or defining) property is a property which *must* be valid for all phenomena in the extension of the concept.

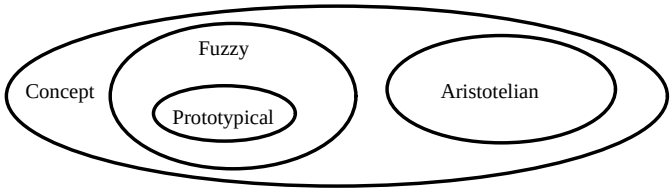


Figure 2: perspectives on concepts

- A *characteristic* property is a property which *may* be valid for some of the phenomena in the extension of the concept.

The properties: can Run, has four Legs are examples of (what we usually think of as) mandatory properties for dogs. The property: can Perform, e.g. can Shake Hands, is an example of (what we usually think of as) a characteristic property for dogs.

Perspectives

The following perspectives, cf. figure 3, on the matching of phenomena and concepts utilize the distinction between mandatory and characteristic properties:

Aristotelian Perspective In the Aristotelian perspective the set of properties in the intention are all *considered* mandatory (or defining) properties. For each of these mandatory properties we can objectively decide if some given phenomenon actually possesses that property. If and only if some phenomenon possesses all the defining properties then the phenomenon is in the extension of the concept.

The Aristotelian perspective is usually used in formalized descriptions of models. E.g. the Department of Motor Vehicles has a precise definition of a Bus: It is a Vehicle with Seats for more than 8 persons.

Fuzzy Perspective In contrast to the Aristotelian perspective on concepts this perspective is based on subjective decisions. This perspective is called the "Fuzzy" perspective because often it is uncertain which phenomena are in the extension. In the Fuzzy perspective the set of properties in the intention are all *considered* characteristic properties, i.e. properties we would expect to find for the phenomena in the extension. E.g. birds can Fly, penguins and emus cannot Fly but still they are considered birds.

The point is that we cannot decide whether or not some phenomenon is covered by a concept just by looking at the description of the concept. We have to make a subjective decision, i.e. a decision outside of the scope of the concept.

Prototypical Perspective A special variant of the Fuzzy perspective is the prototypical perspective, cf. [Lieberman, 1986]. In this perspective a special element in the extension of a concept is selected as a distinguished phenomenon. This phenomenon is used as the prototype for the concept. Maybe the neighbors dog, Fido, just fits with the essence of our understanding of dogs. Or Balder is the prototype for the concept PetDog. To decide whether or not some phenomenon is in the extension the phenomenon is compared to the prototypical phenomenon. If some animal, White looks like Fido it is

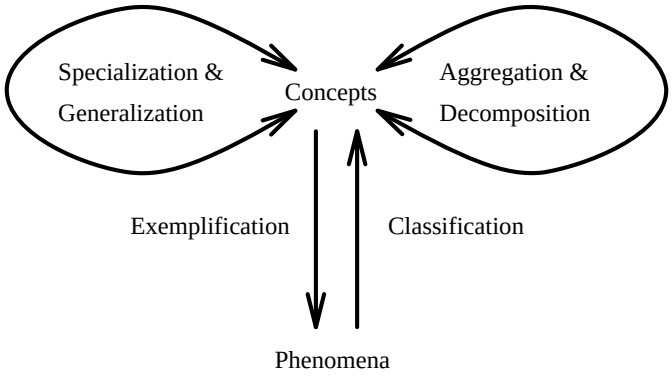


Figure 3: Abstraction

probably a Dog. If it looks more like Tom (and not Jerry), it is more likely that White actually is a Cat.

The intention for a prototypical concept is that all properties of the prototype, as a starting point, are seen as characteristic properties.

3.2 A Model of Abstraction

The phenomena and concepts captured by some model are related by the various elements of abstraction, such as classification & exemplification, generalization & specialization and aggregation & decomposition, cf. [Wand, 1990]. Such elements introduce structures among the concepts and phenomena such as classification and aggregation structures.

The following model for abstraction may not be complete but it has at least shown it applicability in many cases. The basic elements of abstraction, classification, aggregation and specialization and their respective dual elements, may be seen as mappings between concepts and phenomena. The mappings are illustrated in figure 3. The figure shows that aggregation and specialization are mappings between concepts, whereas classification is a mapping between phenomena and concepts.

Let P denote the set of phenomena, C the set of concepts and S set of sets of concepts.

Classification

- *Classification*: P $\rightarrow$ C

  Classification is the most basic element relating a concrete phenomenon to an abstract concept. E.g. Balder is (classified as) a Dog. A phenomenon classified by some concept is in the extension of that concept.

- *Exemplification*: C $\rightarrow$ P

  Exemplification is a mapping from concepts to phenomena. E.g. Balder is an example (an exemplification) of a Dog.

Aggregation

- *Aggregation*: S $\rightarrow$ C

  In aggregation a new concept is formed on the basis of more simple concepts. Eg. Tail, Head, Body and Leg can be aggregated to form Dog.

- *Decomposition*: C $\rightarrow$ S

  The (more complicated) concept is decomposed into more simple concepts. E.g. Dog is decomposed into Leg, Body, Head and Tail.

Specialization

- *Specialization*: C $\rightarrow$ C

  In specialization a closely related concept is formed by adding more properties to the concept. By adding more properties only a subset of the phenomena in the extension of the general concept may be in the extension of the more special concept. E.g. Dog may be specialized to Watchdog and it may be specialized to HuntingDog. Lassie is a Watchdog but no HuntingDog where Balder certainly is no Watchdog but may be a HuntingDog (actually it is a "dachshund").

- *Generalization*: C $\rightarrow$ C

  In generalization a closely related concept is formed by omitting some properties from the concept. By omitting properties all the phenomena in the extension of the special concept will be in the extension of the more general concept. But more phenomena may be covered by the general concept. E.g. Dog may be generalized to Animal and then not only Balder is in the extension of Animal but also Tom and Jerry are Animals.

3.3 Properties of Abstraction

Aggregated Concepts

Assume that some concept, c, is aggregated from other concepts which are all denoted cc in the following discussion. And let p denote a property in I(c), and pp a property in I(cc). Various relations may exist between p and pp. For some of these relations the properties are characterized as emerging, hereditary and concealed properties, cf. [Wand, 1990]:

- *Emerging Property*: The property p is an emerging property if p is not any pp. I.e. p is not just some of the properties of some of the components (or parts) of c.

  The Breaking Distance of a Car is an example of an emerging property that can not be attributed to any single part.

- *Hereditary Property*: The property p is a hereditary property if p is equal to some pp. I.e. p is exactly the same as one of the properties of some of the components (or parts) of c.

  As an example of a hereditary property we usually consider the Color of a Car to be the Color of the Body of the Car – and no other part of the Car.

- *Concealed Property*: The property pp is a concealed property if pp is not any p. I.e. pp is a property of some part of c but it is not also a property of c.

  As an example of a concealed property we usually do not consider the Age of the Tires of a Car to be the Age of the Car. Neither do we consider the Age of the Tires to be a property of the Car.

Specialized Concepts

Assume that some concept, c, is specialized from some other concepts which are all denoted c' in the following discussion. And let p denote a property in I(c), and p' a property in I(c'). Various relations may exist between p and p'. For some of these relations the properties are characterized as inherited, modified and additional properties, cf. [Thomsen, 1986]:

- *Inherited Property*: The property p is an inherited property if p is equal to some p'. I.e. p is just some of the properties of some of the more general concepts c'.

  Assume that Car is a specialization of Vehicle and that Age is a property of Vehicle. As an example of an inherited property we usually consider the Age of Car to be the Age of the Vehicle.

- *Modified Property*: The property p is an modified property if p is a refinement of some p'. I.e. p is seen as a refinement of some of the properties of some of the more general concepts c'.

  Assume that Wheel is a property of Vehicle and that Tractor be a specialization of Vehicle. As an example of an modified property we usually consider the properties Frontwheel and Rearwheel of Tractor to be refinements of the Wheel of the Vehicle.

- *Additional Property*: The property p is an additional property if p is not equal to some p' and is not a modified property (of some p'). I.e. p is some of property added in the description of c as as specialization of c'.

  Assume that Cab is a specialization of Car. As an example of an additional property we usually consider the On/Off Sign on the Cab to be an additional property which ordinary Cars do not have.

3.4 Multiple Abstraction

The mappings described above may actually be relations:

- *Classification*: A phenomenon may be classified by more than one concept i.e. the phenomenon has certain characteristics such that several classifications are meaningful. E.g. Balder is both a PetDog and HuntingDog. (Thus multiple classification means more than the additional, implicitly given classification of the phenomenon by all the generalizations of the concept used for classification.)

- *Exemplification*: A concept may be exemplified by more than one phenomenon. Obviously all the phenomena in the extension may serve as examples of the concept. E.g. Fido, Balder, Lassie etc are all examples of Dog.

- *Aggregation*: A set of concepts may be used to aggregate more than one concept, i.e. some concepts are constructed from the same set of concepts. E.g. Leg, Body, Head and Tail may be used to aggregate Dog as well as Cat.

- *Decomposition*: A concept may be decomposed into more than one set of concepts, i.e. for some reason we may want to understand the same concept in terms of several decompositions. E.g. Dog may be decomposed into Leg, Body, Head and Tail but it may as well be decomposed into Hair, Meat, Bone etc.

- *Specialization*: The same concept may be specialized in several ways. E.g. Animal may be specialized to both Dog and Cat. The same effect is obtained by seeing the general concept as a common generalization of the specializations. E.g. Dog and Cat are both specializations of Animal and Animal is a generalization of both Dog and Cat.

- *Generalization*:

  The same concept may be generalized in several ways. Eg. Pitbull may be generalized into Pet as well as Predator; also Cats and Canaries are Pets, while Tigers and Sharks are Predator. The same effect is obtained by seeing the special concept as a common specialization of the generalizations, cf. [Kristensen, 1989].

4 Examples

To illustrate our approach to programming languages, we will in this section examine two existing abstraction mechanisms known from the Smalltalk language [Goldberg and Robson, 1983], blocks and messages. The basis for our analysis is our conceptual understanding of abstraction; we believe that all concepts can be aggregated/decomposed and specialized/generalized. Neither blocks nor messages are subject to these kinds of abstractions. In that sense the examples illustrate how our approach can be used to generate new ideas for more powerful language constructs. While our analysis primarely focuses on the machanisms as found in Smalltalk, we compare possible extensions to similar mechanisms found in other languages.

4.1 Blocks

The block construct from Smalltalk (and its sibling the lambda expression from Scheme [Keene, 1989]), is a powerful abstraction mechanism with roots in function theory. We will in this section examine this abstraction mechanism from a conceptual perspective. The purpose is to illustrate the perspective of conceptual understanding on programming languages.

Traditionally the block construct is seen as an abstraction mechanism. We view it as a linguistic representation of a concept. In order to do that the extension and intention must be identified.
The extension of a block is a set of executions. Its intention captures important properties of those executions.

The properties of a block are:

Parameters. As is normally the case with concepts, the concrete phenomena in the extension has values of properties that are unique to each individual, yet the phenomena are classified as belonging to the same concept. The parameters of the block leaves open the actual values while stating that a value must exist. For some languages the types of the values can be specified.

Return value. Same comment as for parameters.

Exceptional returns. Some languages includes exception handling. In that case an important property of a block is what kinds of exceptions it can raise.

Free variables. Besides parameters and local variables a block can refer to free variables which are not bound in the block. The set of free variables and their binding mechanism are important properties of a block. Especially if the language

uses dynamic scoping, free variables are a property of great concern.

Body. The actual body of the block prescribes the constituents of the executions in the extension of the block.

Given that the extension of a block is the set of possible executions, and that the intention is described using the above set of properties, we can examine what kind of abstraction mechanisms are supported for the block–concept.

Exemplification corresponds to invocation with actual parameters, and is clearly supported. Classification would be to take a execution, and form a block–concept based on that. Aggregation of blocks is surely supported, in that functional composition corresponds to block aggregation. The inverse, decomposition is not supported; if we are given a block, there is no way we can decompose it into smaller blocks within the programming language. Specialization and generalization is not supported either.

So while the block is generally seen as very powerful abstraction mechanism, it seems rather poor from a conceptual perspective.

In the rest of this section, we will speculate on what specialization could mean for blocks.

If we start by looking at the parameters and return values of blocks. A specialization of a given block should have an extension which are a meaningful subset of the general block. One way of reducing the extension is to further refine or extend the intension. For parameters this means to constrain the set of acceptable values for parameters. The limit of this approach is to constrain it to a specific value. Therefore one way to create a specialized block, is by parameter reduction. If a block takes three arguments, we can create a specialized block of only two arguments by fixing one parameter value.

For free variables, we can create a specialization of the block by restricting free variables. The subconcept becomes a block whose extension is the subset of the general block where the given variable has the specific further binding. For dynamically scoped languages one could easily imagine a language construct for binding a free variable in order to create a specialized block.

In Simula [Dahl *et al.*, 1970] and Beta [Madsen *et al.*, 1993] there exists a special *inner* statement. In connection with specialization the inner statement can be bound to some sequence of actions. The complete action becomes the general action, with the specialized actions substituted for the inner statement. This could clearly be done for blocks also. Part of the body could have open inner points which could be filled out in specializations. The inner-mechanism is seen as a most–general–execution, having all possible executions in its extension; that is, inner is seen as an execution equivalent to the general class "object" found in some programming languages. Thus when we create a block with inner, we say explicitly that here we allow some execution to take place.

4.2 Message

The message as known from Smalltalk is not a very well supported abstraction mechanism. The following examination of the message mechanism will be based on our conceptual understanding.

We must establish the extension and intention of the message concept. The extension of the message concept is all the messages sent during a program execution. The intention of the message concept include such properties as:

- The receiver of the message.
- The name of the message.
- The arguments; the number of arguments, their values and their types.

In a program execution there are no actual message objects available because of optimizations. When the method lookup fails however, the runtime system will combine information to exemplify Message and present the message to the programmer as part an error notification. Classification is not interesting as long as there is only a single Message concept.

In the following we will speculate on the usefulness of aggregation and specialization in connection with messages.

Aggregating messages would mean to combine several messages to form a compound message. A very simpleminded use of this could be to view this as a sort of transaction or atomicity. Beta has unified routine invocation and compound assignments. This enables us to invoke two routines in one statement as:

```
(15+2, 100-1) -> (a.foo, b.bar)
```

which corresponds to invoking (sending message) `foo` with `a` as the receiver, and 17 as argument, and `bar` with `b` as receiver and 99 as argument. We might view this as an aggregated message. In Beta, this means that the `foo` routine is invoked first, and the `bar` routine next. But the arguments for both are computed *before* any message is sent. In some early concurrent object–oriented languages [Yonezawa *et al.*, 1986; Tokoro and Ishikawa, 1986] messages were manifest in the language. In ABCL/1 [Yonezawa *et al.*, 1986] messages could be classified as either ordinary or express, and as either *Past, Now,* or *Future*. In both languages, however, the user could not them selves describe new types of messages.

With the above understanding of messages, it is difficult to see how messages should be specialized. The problem is that messages are very intangible objects in must programming languages, and do not have any first order status. To support specialization for messages we must be able to create message concepts. In languages supporting a distributed runtime system, messages are tangible at least at the runtime level. For such languages, we could elevate messages to first class abstraction mechanisms.

Both aggregated and specialized messages might require new forms of exemplification, as is the case with the Beta compound evaluation. In some reflexive architectures Message is available at the meta-level, where messages are therefore subject to specialization and aggregation.

5 A Framework

In this section we present a framework for the analysis of object–oriented programming languages based on conceptual understanding. The framework is based on conceptual understanding and on pragmatic issues of language design in general. The framework is organized as a number of aspects, namely

- *The perspective of abstraction*
- *The functions of abstraction*

- *The aspects of functions of abstraction*

Our framework will allow us to measure to what extent a given programming language supports conceptual understanding. It is important to distinguish between language support of some given concept and simulation of the concept in some given language. We shall use the definition given in [Stroustrup, 1987]: "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."

We give examples of the use of the framework on the object–oriented languages Smalltalk, Beta, C++ [Ellis and Stroustrup, 1990], Eiffel [Meyer, 1992], CLOS [Keene, 1989], and Self [Ungar and Smith, 1987]. Object–oriented languages may also be discussed and analyzed without involving conceptual modeling cf. [Bobrow and Stefik, 1985] and [Wegner, 1987]. We examine object–oriented languages because these fit excellent, not only with classification and aggregation, but also with specialization. While not complete, it is our intention that the framework serves to illustrate the manner in which language analysis can be performed with the outset in concept understanding.

In order to discuss how the example languages supports the abstraction mechanisms of our conceptual understanding, it is necessary to state what is seen to model a concept in each of these languages. In general classes are modeling concepts and objects (as instances of classes) are modeling phenomena. In addition for Smalltalk, C++ and Eiffel, the properties of a concept are represented as instance-variables and methods.[1] For Beta the linguistic form of a concept is a pattern. The properties of the concept are represented as both reference and pattern attributes. In CLOS the properties are represented by slots of the classes, and by those generic methods which takes objects from the extension of the class as arguments. Self has the notion of prototype objects. But as there is no linguistic distinction between prototypes and normal objects, we choose to view all objects as prototypes, and as such representing a (unique) concept. The properties of the concept is given by its slots, and its extension by the objects that delegates to the prototype.

5.1 The Perspective on Abstraction

This aspect states whether a language has the *Aristotelian*, *fuzzy*, 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 part of this aspect.

	Eiffel	Self
Perspective	Aristotelian	Prototypical
Properties	Mandatory	Characteristic

The meaning of Aristotelian is here limited to mandatory properties only. The meaning of prototypical is here limited to implicit implicit description of concepts only; given the use of prototypes and delegation in Self we have characteristic properties only.

[1] Instance variables and methods are here used generically, and covers members, features and attributes.

5.2 The Functions of Abstraction

This aspect enumerates which abstraction functions are supported; also the different kinds of properties in relation these functions are part of this aspect:

Classification & Exemplification: In programming languages classification is not supported very well. Typically the object is classified as an example of the class from which it is instantiated, it keeps this classification throughout its lifetime, and it possesses only the properties of that class. Exemplification is only supported through instantiation of an object given a class.

Aggregation & Decomposition: Typically aggregation is supported in object-oriented languages in the form of instance variables. If a concept X is aggregated from two Ys and one Z, the class X will have three instance variables qualified as two Y type objects, and one Z type object. All object-oriented languages supports this structural aspect of aggregation. Decomposition is supported only through aggregation.

Specialization & Generalization: Specialization is supported by means of various forms of inheritance. The inheritance mechanisms differ especially for modified properties. An additional way to distinguish between the languages may be to clarify whether or not the languages support the refinement of all the aspects of concepts, i.e. not only variables and methods but also an action part. [Pedersen, 1989] presents a generalization mechanism, which has not yet found its way into any mainstream languages.

Function	Beta	Self
Classification	Supported	Not Supported
Exemplification	Supported	Not Relevant
Specialization	Supported	Supported
Generalization	Not supported	Not supported
Aggregation	Supported	Supported
Decomposition	Not supported	Not supported

Remarks:
Self. New objects 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; hence there is no denomination of a class and any object can be used as prototypical. Self only *enables* classification. For the same reason exemplification is not relevant. Specialization and aggregation is supported by Self even though there is no single prototypical element available. The parent mechanisms available for inheritance is seen as modeling specialization for any object used as prototypical element."

Beta allows instance specific properties as well, thus allowing objects to have a richer set of properties than their concept (pattern); however, Beta does not support dynamic classification.

Properties

For aggregation and specialization we can also look at the relationship between the properties of the involved concepts. For specialization it is: *Inherited -, Modified -,* and *Additional property.* For aggregation this is: *Emerging -, Hereditary -,* and *Concealed property.*

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

Modified. Modified properties are found in many forms. A property being a refinement of another property must have

at least the same meaning as the property modified. This implies that a completely redeclared property is not considered a modified property.

Additional. The only restriction is that the property must have a unique name in the context in which it appears. In some languages behavioral properties must be explicitly marked as virtual or not. The conceptual interpretation of this is that if a virtual property is redefined in a sub–class, it is seen as modifying the property; if a non-virtual property is redefined it is seen as a additional property which accidentally has the same name as a property in the general concept.

Emerging. Emerging properties are properties of the aggregate that that does not stem from any single component, but from the interaction of a number of properties from several components. In programming such interactions are not accidental, but explicitly programmed. We view methods of the aggregate that explicitly combine properties of the components as an emerging property. In that sense all object-oriented languages supports emerging properties.

Hereditary. The property is hereditary only if it can be specified as such through some simple notation, i.e. without being redeclared. We are not aware of any languages supporting hereditary properties.

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

Properties	ST80	C++
Specialization		
Inherited	Supported	Supported
Modified	Supported	Supported
Additional	Supported	Supported
Aggregation		
Emerging	Supported	Supported
Hereditary	Not supported	Not supported
Concealed	Supported	Supported

Remarks:
Smalltalk. All properties are always inherited. Modification is only supported for methods. Both instance variables and methods can be added.

C++. Selected properties can be annotated as "private", and they will not be inherited to subclasses. A conceptual interpretation can be given for this: A private property is a property representing pragmatic issues of implementation. They are therefore not part of our conceptual model building. Member functions must be declared virtual to support modification in the subclasses. Only function members can be modified.

5.3 Aspects of Functions of Abstraction

For each of the functions the following aspects are valid:

- *Form*: The form of the function, i.e. it is implicit or explicit described in the program.
- *Time*: the time of the use of function, i.e. it is dynamic (run–time) or static (program–description time).
- *Order*: The order of the function, i.e. it is possible to define the function as single (only) or multiple valued.

These aspects can be discussed in relation to all the abstraction functions:

Classification: The distinction between types and classes may be seen as a way of supporting implicit classification. Also languages without types supports implicit classification to some extent. By Time we mean whether the object–class relation is given at program description time or it can be changed at run–time. The predicate classes proposed in [Chambers, 1993] addresses the need for dynamic classification in a quite direct manner, allowing an object to be classified according to its current state and values.

Exemplification: Single is understood as the possibility of having singular objects, e.g. in the form of prototypes as in Self.

Specialization: Single is an uninteresting special case only. This should not be confused with *singular* specializations, i.e. one–of–a–kind specializations.

Generalization: Form and Time are as for specialization. Multiple generalization is in practice supported by various forms of multiple inheritance and is thus seen as some kind of specialization. However generalization is a function, that given some concept defines some more general concept from this. Multiple generalization means that this may be done several times for the same (more special) concept producing several generalizations of this. In contrast the multiple inheritance mechanism is trying to express such multiple independent generalizations in one description

Aggregation, Decomposition: An additional distinction is whether or not a reference can be to a constituent object (in the sense being an integral part of) or an autonomous object (in the sense being independent of).

Aspects of Functions	ST80	C++	Self
Classification			
Form	Explicit	Explicit	Not Relevant
Time	Dynamic	Static	Not Relevant
Order	Single	Single	Not Relevant
Exemplification			
Form	Explicit	Explicit	Not Relevant
Time	Dynamic	Static	Not Relevant
Order	Multiple	Multiple	Not Relevant
Specialization			
Form	Explicit	Explicit	Explicit
Time	Static	Static	Dynamic
Order	Multiple	Multiple	Multiple
Generalization			
Order	Single	Multiple	Multiple
Aggregation Decomposition			
Form	Explicit	Explicit	Explicit
Time	Static	Static	Static
Order	Single	Single	Single

Remarks:
Smalltalk. We have chosen not to take the meta-aspects of Smalltalk into account; We therefore chose to ignore that classes can be created and modified at runtime. The method "changeClassToThatOf:" will change the class of an object; so we put dynamic under the time aspect of classification. Static is supported as well.

C++. No further remarks.

Self. Changing the parent reference is seen as as dynamic specialization; the parent link is explicit. There can be multiple parent links.
For all three languages, aggregation is supported the same way.

6 Conclusion

We have presented an approach to programming illustrated by the modeling diagram. We have argued that this approach may imply certain important qualities for the program de-

velopment process and product. For the elements in the diagram we have presented some of the knowledge presently available. Also we have illustrated how this knowledge may be used to analyze existing languages. Concerning conceptual understanding we have presented well–established material only in sections 3.1 and 3.2. But various other proposals exist: A study of parts and wholes are presented in [Østerbye, 1990], the concept of aspects is presented in [Østerbye and Kristensen, 1994] and properties, relations etc are discussed in [P.J.Braspenning *et al.*, 1989].

We have presented a framework for analyzing object-oriented languages on the basis of concept understanding. The power of the framework lies in its language independence; independent both of specific languages, and independent of concrete language mechanisms. It is noteworthy than the framework has pointed out a number of issues that *none* of the languages support: generalization, decomposition, hereditary aggregation, and implicit classification.

It appears to be straightforward to use the framework, however

- The conceptual understanding from section 3 supports only directly the construction of parts of the framework
- Many exceptions and remarks in relation to the interpretation of the aspects and in relation to the mechanisms of the languages are necessary to make the example illustrative and understandable

References

[Bobrow and Stefik, 1985] Daniel G. Bobrow and Mark J. Stefik. Object-oriented programming: Themes and variations. *The AI Magazine*, pages 40–62, 1985.

[Chambers, 1993] Craig Chambers. Predicate classes. In *Proceedings of ECOOP*, 1993.

[Dahl *et al.*, 1970] Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard. *Simula 67, Common Base Language*. Norwegian computing center, Oslo, 1970.

[Ellis and Stroustrup, 1990] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ Refererence Manual*. Addison-Wesley, 1990.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80 - The language and its implementation*. Addison-Wesley Publishing Company, 1983.

[Keene, 1989] S. E. Keene. *Object–Oriented Programming in Common Lisp*. Addison Wesley, 1989.

[Kristensen, 1989] Bent Bruun Kristensen. Multiple specialization and generalization in concept modeling (modeling concepts by multiple inheritance). In *Proceedings of the 1989 European Simulation Multiconference*, 1989.

[Kristensen, 1993a] Bent Bruun Kristensen. Transverse activities: Abstractions in object–oriented programming. In *Proceedings of International Symposium onObject Technologies for Advanced Software (ISOTAS'93), Kanasawa, Japan*, 1993.

[Kristensen, 1993b] Bent Bruun Kristensen. Transverse classes & objects in object–oriented analysis, design, and implementation. *Journal of Object–Oriented Programming*, 1993.

[Kristensen, 1994] Bent Bruun Kristensen. Abstraction mechanisms for object-oriented modeling of the organization and cooperation of classes and objects. Technical Report R-94-2001, Aalborg University, 1994.

[Lieberman, 1986] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA'86 Proceedings*, 1986.

[Madsen *et al.*, 1993] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison Wesley, 1993.

[Meyer, 1992] Bertand Meyer. *Eiffel, The Language*. Prentice Hall, 1992.

[Pedersen, 1989] Claus H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*, 1989.

[P.J.Braspenning *et al.*, 1989] P.J.Braspenning, J.W.H.M.Uiterwijk, and L.C.J. van Leeuwen H.Bakker. Conceptual tools for modelling complex objects. In *Proceedings of the 1989 European Simulation Multiconference*, 1989.

[Stroustrup, 1987] Bjarne Stroustrup. What is "object–oriented programming"? In *European Conference on Object–Oriented Programming*, 1987.

[Thomsen, 1986] Kristine S. Thomsen. Multiple inheritance, a structuring mechanism for data, processes and procedures. Technical Report DAIMI PB–209, Department of Computer Science, Aarhus University, 1986.

[Tokoro and Ishikawa, 1986] Mario Tokoro and Yotaka Ishikawa. Orient84/k: A language within multiple paradigm in the object framework. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, 1986.

[Ungar and Smith, 1987] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA'87 Proceedings*, pages 227–242. ACM, 1987.

[Wand, 1990] Yair Wand. A proposal for a formal model of object. In Won Kim and Fredrick H. Lochowsky, editors, *Object-Oriented Concepts, Databases and Applications*, chapter 21. Addison-Wesley/ACM Press, 1990.

[Wegner, 1987] Peter Wegner. Dimensions of object–based language design. In *OOPSLA'87 Proceedings*, 1987.

[Yonezawa *et al.*, 1986] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object–oriented programming in abcl/1. In *OOPSLA'86 Proceedings*, 1986.

[Østerbye and Kristensen, 1994] Kasper Østerbye and Bent Bruun Kristensen. Aspects of objects - the analysis, design, and implentation of a new language construct. In Preparation, 1994.

[Østerbye, 1990] Kasper Østerbye. Parts, wholes, and subclasses. In Bernd Schmidt, editor, *Proceedings of the 1990 European Simulation Multiconference*, pages 259–263, 1990.