Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages

Jørgen Lindskov Knudsen and Ole Lehrmann Madsen
Computer Science Department, Aarhus University,
Ny Munkegade 116, DK-8000 Aarhus C, Denmark.
E-mail: jlk@daimi.dk -and- olm@daimi.dk

Abstract

One of the important obligations of an expanding research area is to discuss how to approach the teaching of the subject. Without this discussion, we may find that the word is not spread properly, and thus that the results are not properly utilized in industry. Furthermore, discussing teaching the research area gives additional insight into the research area and its underlying theoretical foundation. In this paper we will report on our approach to teaching programming languages as a whole and especially teaching object-oriented programming.

The prime message to be told is that working from a theoretical foundation pays off. Without a theoretical foundation, the discussions are often centered around features of different languages. With a foundation, discussions may be conducted on solid ground. Furthermore, the students have significantly fewer difficulties in grasping the concrete programming languages when they have been presented with the theoretical foundation than without it.

Introduction

Most text books on programming languages describe the *technical differences* between various language constructs. This implies that emphasis is often concentrated around *features* of one language compared to features of another language. This makes it difficult to discuss the *qualitative difference* between languages. The well-known "Turing Tarpit"* states the fact that, on theoretical basis, any computation which can be expressed in one of the familiar programming languages can also be expressed in any of the others — including Turing machines. Thus comparison of features should be more than a discussion about whether or not a given construct may be *simulated* in another language. Furthermore, "a technical discussion" of programming languages is often lacking arguments about the programmers *perspective*† on programming. (One illustrative example of this approach can be found in [17].)

*According to W.A. Wulf[41] the "Turing Tarpit" was originally formulated by Alan Perlis.

†Please note, that others use the phrase *paradigm* instead of perspective here, but the use of paradigm in computer science has been questioned from several different sources; see e.g. [26].

Instead of technical details it is often much more fruitful to discuss requirements for supporting one or more perspectives. However, there are books that discuss languages relative to one perspective. The perspectives are usually based on mathematical models. (One illustrative example of this approach can be found in [37].) Few books are devoted to the object-oriented perspective. This may be due to the fact that the foundation/basis of object-oriented programming has not yet been very well formulated.

The purpose of this paper is to describe how programming languages are being taught at the Computer Science Department, Aarhus University. This teaching is highly influenced by 15–20 years of research in programming languages and system development in Scandinavia, mainly in Oslo and Aarhus. For more than 10 years, the teaching of programming languages at the Computer Science Department, Aarhus University has been heavily influenced by the object-oriented perspective. The approach to teaching object-oriented programming as well as the structure of the present courses will be described.

First a description of the overall approach to teaching Computer Science at the Department will be given. This is followed by a description of the objectives for teaching the subject of programming languages, leading to a description of the approach to teaching object-oriented programming. Finally, the courses that have been given over the last few years, both in the department and to industry are presented.

1 Background

When planning s course it is important to be conscious of the prerequisites of the students in order to design the most effective course. Our courses have primarily been given to students at the Computer Science Department and the overall approach to teaching computer science at the department will therefore be described.

The Computer Science Department at Aarhus University has grown out of the Institute of Mathematics. This has resulted in a strong influence of theoretical approaches to subjects. That is, the prime emphasis in teaching computer science is put on teaching theories and perspectives. Teaching concrete techniques and methods are considered as utilizations of the theories and perspectives. This implies that the students are trained in handling abstract notions besides being able to utilize these abstract notions in approaching concrete problems. That is, they are taught the abstract notions in order to make them capable of (relatively easy) learning techniques and methods for applying the abstract notions on concrete problems.

A study for the Master's degree is supposed to take 5 years. Most students, however, take considerable longer time to complete their degree. During the first 3 years approximately half of the time is devoted to computer science. The other half is usually mathematics and statistics. After 3 years the students are at the level of a Bachelors degree. The last 2 years are full time computer science, including a thesis.

With respect to programming languages, the students are trained during the first 2 years of study

in using traditional procedural languages, such as Pascal, Modula-2, and Concurrent Pascal. This implies that their perspective on programming is highly influenced by the procedural programming perspective.

The courses described here are given on year 3–5 of the study.

2 Objectives

The fundamental principle is that teaching concrete programming languages should be a subordinate objective in teaching the subject of programming languages. There is a number of reasons for this:

- It is very difficult to predict which programming languages will be the most influential in industry 10–20 years ahead (unless we settle with the good old workhorses Cobol and Fortran). Furthermore, we *have to* make sure that the students of today are able to access the programming languages in the 21'st century (it's only 11 years ahead). By teaching them concrete languages of today, we are liable to make it difficult for them to access the languages of the 21'st century.

- In any teaching situation, it is most important to emphasize the principles and utilize this insight to access concrete examples of the principles. If you e.g. teach people object-oriented programming by just giving a course on Smalltalk, they may have difficulties in understanding the basic principles of object-oriented programming. They will very likely equalize object-oriented programming with programming in Smalltalk. Furthermore they often have difficulties in actually learning Smalltalk.

- By learning principles, techniques and concepts, the student will be able to evaluate different programming languages on basis of the principles, and not on basis of more or less important concrete differences (such as syntax). Furthermore, with well-chosen principles, there is a better chance of the evaluation being fair to all languages under consideration, and not being in favor of one specific language. For any concrete programming language, it may be difficult for the student to distinguish the important and general constructs of the language from the always present idiosyncrasies. State-of-the-art in programming language design has not yet reached a level where it is possible to design a language that does not end up having some poorly designed features, even if the overall principles are good and sound. Simula 67[3] and Smalltalk-80 are good examples of this. The basic principles behind these languages were excellent at the time of invention. Still a user of these languages is confronted with a large number of poorly designed edges.

- The students must be made able to consider using different languages for different programming tasks. In this case it is important that the student (when he later acts as a system developer) is aware of the perspectives which underlie the specific languages (or rather, is able to identify the underlying perspective of different languages) since the underlying perspective

of a programming language in a sense outlines the borders of the application areas for which the particular programming language is well-suited, and therefore will have an impact on the programming process.

We also strive towards avoiding the discussion of *features* of programming languages, and stress that in order to make languages accessible it is very important that the concepts *simplicity*, *consistency* and *orthogonality* are the primary guidelines — features will never be fully utilized or understood if they are nothing but features. Simplicity, consistency and orthogonality of language constructs are what makes a language accessible, irrespective of which programming perspective the language supports.

The quantitative approach to evaluating programming languages has some serious defects, the "featurism" mentioned above being one of them. Without more abstract notions of what constitutes important aspects of a programming language, one is seriously in danger of the "Turing tarpit". This may stop any serious discussion of different programming languages, since it does not make any distinction between *supporting* and *simulating* a particular language construct. One very good example of not making this distinction clear is the discussion by Per Brinch Hansen of selecting language constructs to be included in the Edison programming language[6]. One of these discussions is about whether it is necessary to include both the **repeat**- and the **while**-statements. P.B. Hansen argues that since **repeat** may be expressed in terms of **while** there is no need for the **repeat**. This is the "Turing tarpit" since applying the argument repeatedly reduces any control statement to being only specific **goto** structures, and since not having the **repeat** in the language places the burden on the programmer to implement the **repeat** *each* time he finds a need for it. In this case we will say that the Edison language *simulates* the **repeat** concept.

The question is now: What do we demand of a language in order for it to *support* a given concept? Let us use the **repeat** example again. If the **repeat** were present in a language, we would of course state that the language supports the **repeat** concept, but more generally we would say that a language supports the **repeat** concept if there exists a mechanism in the language that makes it possible to state the **repeat** concept as an abstraction which may then be used on equal terms with the built-in concepts. In this way it is possible to create new abstractions that can be safely implemented once, and then securely utilized over and over. That is, the consistency of the abstraction is expressed once in the implementation and not scattered all over the program as with simulated concepts discussed above.

Returning to object-oriented programming, it is important to be aware that object-oriented programming is a lot more than inheritance, objects, and message passing or member function calling, very much the same way as structured programming is a lot more than goto-less programming.

3 Approach

The approach to teaching programming languages and especially object-oriented programming is very much influenced by the perspective you have on the role of the programming language in the system development process. In fact this role is a three-way role: as a means for expressing concepts and structures (*conceptual modeling*), as a means for *instructing* the computer, and as a means for *managing* the program description. Just focusing on the role as a means for instructing the computer is far to narrow. In the role for conceptual modeling, the focus is on constructs for describing concepts and phenomena. In the role for instructing the computer, the focus is on aspects of the program execution such as storage layout, control flow and persistence. Finally, in the role for managing the program description, focus is on aspects such as visibility, encapsulation, modularity, separate compilation, library facilities, etc.

Some of the success in teaching programming languages can be traced back to the emphasis that is put on using these roles as the foundation of the approach. Here the roles as means for conceptual modeling and prescription have proven very effective, and to some extent this makes the approach to teaching programming languages novel. It has been found that restricting the discussion of programming languages to the role of instruction (or coding) is far to restrictive, primarily because the end-product of a programming process (the program) cannot (and should not) be viewed in isolation from the programming process and thereby the application domain.

3.1 Perspectives

Teaching the perspective of object-oriented programming cannot (or should not) take place in isolation from other perspectives. Extensive parts of the courses are therefore devoted to programming perspectives as such, and presentation of various different programming perspectives.

Procedural programming[‡] is taken as the starting point for the discussion. Functional/logical programming and object-oriented programming are then described as two different reactions to several problems related to the concept of state in procedural programming. In functional programming, the approach has been to eliminate the concept of state, whereas the approach taken in object-oriented programming has been to treat the concept of state as a first-class citizen. In addition various other perspectives such as the process perspective, the type system perspective and the event perspective are treated. The latter three perspectives are not treated extensively but primarily in the context of the other perspectives.

Below a short formulation of the perspectives are given.

Procedural Programming

A program execution is regarded as a (partially ordered) sequence of procedure calls, manipulating

[‡]To ease the writing we will use the phrase "... programming" interchangeable with the phrase "the ... programming perspective".

data structures. This perspective is the most common and supported by languages like Algol, Pascal, C and Ada. Procedural programming has the prime focus on the instructive role of the programming language and very little support for the other roles of the programming language, and is therefore not sufficient.

The courses we give do only treat procedural programming on the level of perspective since the students in their previous courses have been trained extensively in procedural programming. We do, however, cover Ada as a representative of state-of-the-art within procedural programming languages.

Functional Programming

A program is regarded as a mathematical function, describing a relation between input and output. In functional programming, the concept of state or variable is eliminated entirely. I.e functional programming is variable free programming.

Lisp is often mentioned as the most prominent "functional programming" language. It is well-known that most Lisp variants also have variables and thereby state. For this reason it is important to stress that instead of classifying a given programming language as either a "functional programming language", a "procedural programming language", etc., it is often more useful to discuss to what extent a given programming language has support for functional programming, object-oriented programming, etc. There are few programming languages that are based purely on one perspective.[§]

In the courses, functional programming is treated on the level of perspective. The students are trained in functional programming using the Scheme programming language[27]. In another part of the department, a course is devoted entirely to the subject of functional programming using the Miranda language[39]. This course is complementary to the course described here, since its main emphasis is on the theoretical foundation for functional programming. Students are advised to take that course if they have special interest in functional programming.

Constraint-Oriented (logic) Programming

A program is regarded as a set of equations, describing relations between input and output. As in functional programming, the concept of state is eliminated in constraint-oriented programming. Prolog is the most dominant example of a language supporting the constraint-oriented perspective.

In the courses we treat constraint-oriented programming on the level of perspective and exercises the constraint-oriented programming perspective using the Prolog language. In another part of the department, a course is devoted entirely to the subject of logic programming using the Prolog language and again students are advised to take that course if they have special interest in constraint-oriented programming.

[§]To ease the writing, we will however use the phrase "... programming language" to mean "programming language with primary support for ... programming".

Object-Oriented Programming

A program execution is regarded as a physical *model, simulating the behavior of either a real or imaginary part of the world.* The object-oriented perspective on programming is in contrast to the above perspectives that are focusing either on manipulations of data structures or on mathematical models. The object-oriented perspective is closer to physics than mathematics. Instead of describing a part of the world by means of mathematical equations, a *physical model* is literally constructed. This means that elements of the program execution are regarded as models of *phenomena* and *concepts* from the real world. The part of the world being modeled is described by the program. Some of the well-known examples of languages supporting this perspective are Smalltalk-80, Beta and C++.

This "definition" cannot be seen in isolation but must be understood in a broader context (this applies for the other perspectives as well.) In the courses we elaborate extensively on this broader context as described in section 3.2.

The Process Programming Perspective

A program execution is regarded as consisting of a set of processes, each involved in their own activities, and communication with other processes.

The process perspective is focusing on structuring the transformations on state. Some of the well-known examples of languages supporting this perspective are CSP[7], Concurrent Pascal, and Ada.

In the courses we treat the process perspective at the level of perspective and study CSP and the process aspects of Ada.

It is also discussed to what extent the process perspective is at the same level as some of the other perspectives. You may e.g. view the process perspective as a development of the procedural perspective or as subordinate to the object-oriented perspective.

Concurrent programming in languages like Concurrent Pascal and Ada is in our view mainly carried out as a generalization of a "sequential" procedural perspective.

The modeling of objects with individual action sequences is a fundamental part of the Scandinavian tradition for object-oriented programming. Simula 67 has support for coroutines and Beta[16]. has support for coroutines and concurrency. For this reason it is natural to view the process perspective as subordinate to the object-oriented perspective.

Again we must stress (and this is of course also done in the courses) that there is no objective way of defining what is right and wrong with respect to the different perspectives on programming.

The Type System Perspective

The type system perspective may be viewed as a subordinate perspective of the constraint-oriented perspective, where the relations are described by means of type structures. Since type systems are an integrated part of many procedural languages, these perspectives co-exist harmonically in the

same programming language. Many type systems have been proposed in the past, most notably the Pascal type system, the Ada type system, the ML type system (incorporating type inference and polymorphism), and the Cardelli and Wegner type system (incorporating hierarchical types).

Type systems are treated at the perspective level, and are related to the object-oriented perspective by studying the relations between hierarchical type systems and classification hierarchies. We focus on the Ada type system and on the Beta type system.

The Event Perspective

A program execution is regarded as a (partially ordered) set of events. The event perspective is a theoretical approach to the process perspective. The most notable representatives of the event perspective are Petri net models[28], Calculus for Communicating Systems (CCS)[24], CSP-85[8].

The event perspective is only treated at the perspective level, and as with the functional and constraint-oriented perspective, students with special interest in the event perspective are advised to follow the Petri net course, given by others in the department.

As indicated above we do not pretend to be objective in our teaching in the sense that we make it clear that our perspective on programming is mostly object-oriented. This implies that we (in the interest of honesty) inspire the students to take the specific courses directed towards the other perspectives if they find special interest in them.

3.2 Teaching Object-Oriented Programming

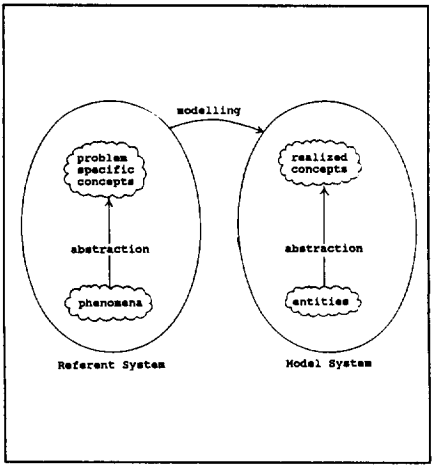
Having described the context of object-oriented programming, we will now present the actual subjects treated in our courses on object-oriented programming.

3.2.1 Theoretical Foundation for Object-Oriented Programming

As stated above, the object-oriented perspective must be accessed on basis of a theoretical foundation and not on basis of specific language constructs. The theoretical understanding of object-oriented programming which will be outlined in the following is among others a result of research activities that the authors have carried out together with a number of other people. It is important to stress that the teaching has influenced the research too. A large number of students have treated many of the subjects in their thesis work. A more detailed description of the issues discussed may be found in [13] and [16]. The foundation is highly influenced by the work reported in [9,22,32,25].

Modeling

In order to clarify the different roles that the programming language plays in the programming process, we have to look more closely at that process. The programming process may be described as a modeling process in which several sub-processes take place. The figure illustrates the



programming process as a modeling process between a referent system and a model system. The *referent system* is part of the world that we are focusing on in the programming process, and the *model system* is a program execution modeling a part of the referent system on a computer. The referent system is the concrete physical world or some imagination of a future physical world, and as such it consists only of phenomena. As a characteristic human activity, we create concepts in order to capture the complexity of the world around us — we make abstractions. That is, in the referent system, both phenomena and concepts are important. In the model system, we find elements that model phenomena and concepts from the referent system. Objects in a Smalltalk-80 program execution are typically models of physical phenomena in the referent system and the sequence of events generated by the execution of a method is typically a model of a sub-process going on in the referent system. Concepts in the referent system are modeled by abstractions such as classes, types, procedures and functions. The program text is a *description* of the referent system and in addition it is a *prescription* that may be used to generate the model system.

The programming process can now be described in terms of this figure. During the programming process, three sub-processes are taking place: abstraction in the referent system, abstraction in the model system, and modeling. Please note that intentionally we do not impose any ordering among the sub-processes. *Abstraction in the referent system* is the process where we are perceiving and structuring knowledge about phenomena in the referent system with particular emphasis on the problem domain in question. We say that we are creating *problem specific concepts* in the referent system. This process is an integrated part of the system development process. *Abstraction in the model system* is the process where we build structures that should support the model we are intending to create in the computer. We say that we create *realized concepts* in the model system. Finally, *modeling* is the process where we connect the problem specific concepts in the referent system with the realized concepts in the model system.

Concepts and Abstraction

As discussed above, concepts and abstraction are the key notions in our understanding of the programming process. It is therefore necessary to discuss subjects like the notion of concepts and their relations to phenomena, concept understanding, and important aspects of the abstraction process.

A *phenomenon* is something in the world that has definite, individual existence in reality or the mind; anything real in itself. What constitutes a phenomenon is to some degree dependent on the view of the observer. A *concept* is a generalized idea of a collection of phenomena, based on knowledge of common properties of the phenomena in the collection. Concepts may be characterized

by three aspects: the designation, extension and intension. The *designation* refers to the collection of names under which the concept is known. The *extension* refers to the collection of phenomena that the concept somehow covers, and the *intension* refers to the collection of properties that in some way characterize the phenomena in the extension of the concept.

These definitions are deliberately somewhat vague since there are (at least two) different ways to understand concepts: the Aristotelian view and the prototypical (or fuzzy) view. Space does not allow an extensive discussion of these two views — just a short characterization. In the *Aristotelian* view, the concepts are rigidly defined, leading to *sharp concept borders* and *relatively homogeneous* phenomena in the extension. The Aristotelian view is the view that can be mechanized without human interaction. The *prototypical* view, on the other hand, is characterized by *blurred concept borders*, phenomena of *varied typicality* in the extension, and *decision-making/judgement* when a phenomenon is considered for inclusion in the extension. The prototypical view is the view that best describes human concept understanding.

As it can be seen above, the programming process is faced with the problem that not only do we restrict the precision of our model by only considering a part of the world (this is a problem studied in system development courses), but equally important, the modeling process *has* to take into account the restrictions imposed by modeling a possible prototypical concept structure in the referent system into an Aristotelian concept structure in the model system.

In both the referent system and the model system, concept structures are created. This implies that we have to discuss the process of producing and using knowledge, i.e. issues related to epistemology. Part of this discussion includes an introduction to some of the work of Marx, who has split the process of knowledge into three levels:

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

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

3. *The level of thoughtconcreteness.* The understanding corresponding to the abstract level is further developed to obtain an understanding of the totality of the referent system. By having organized the phenomena of the referent system by means of concepts we may be able to

understand relations between phenomena that we did not understand at the level of empirical concreteness. As well as we may be able to explain why things happen and may be able to predict what will happen.

In the process of creating concepts it is useful to identify the three well-known sub-processes of abstraction: classification, aggregation and generalization. To *classify* is to form a concept that covers a collection of similar phenomena. To *aggregate* is to form a concept by describing the properties of the phenomena by means of other concepts. And finally, to *generalize* is to form a concept that covers a number of more special concepts based on similarities of the special concepts. All three sub-processes have an inverse process, called *exemplification*, *decomposition* and *specialization*, respectively.

In general the process of creating new concepts cannot just be explained as consisting of the above sub-functions. In practice the definition of concepts will undergo drastic changes. This is similar to the situation with top-down and bottom-up programming. It is realized by most people that pure top-down or bottom-up development of programs is rarely possible. The understanding obtained during the development process will usually influence previous steps. It is however useful to be aware whether a problem is approached top-down or bottom-up. In the same way it is useful to be aware of the above mentioned sub-functions of abstraction.

The word abstraction may be used to characterize a process, and the sub-functions of abstraction were explained as processes going on with the aim of creating concepts. On the other hand the word abstraction may also be used in a static or descriptive way. A concept is an abstraction. Given a number of concepts, their structure may be described in terms of classification, aggregation and generalization. It is e.g. possible to describe a given concept as a generalization of a number of other concepts.

In teaching it is important that the students are aware of this distinction. When evaluating a given language they might consider to what extent the language support abstraction and its sub-functions as a process and to what extent the language supports abstraction and its sub-functions as a means for describing concept structures.

Information Processes and Object-Oriented Programming

Having discussed concepts and abstraction we turn our attention towards characterizing the part of the world we are interested in creating model systems for, and then characterize object-oriented programming in greater detail.

The kind of model systems we are interested in, are those that model information processes. An *information process* is regarded as a system, developing through transformations of its state. The *substance* of the process is organized as objects. The *state* of the substance may be measured upon through *measurable* properties, and the state of the substance may change as an effect of *transformations* on the substance. Substance is physical matter, characterized by a volume and a position in time and space. Substance have certain properties that may be measured. E.g.

measurements may be compared with other measurements. Transformations are partially ordered sequences of events that change the substance and thereby its properties. Note that by focusing on information processes, concepts exist that cannot be captured, e.g. "God", "good", "bad", etc.

In object-oriented programming, an information process is modeled by organizing the substance of the program execution as a number of *objects*. The measurable properties are modeled as *state of objects*, and transformations are organized as *action sequences* performed by objects. An object is furthermore characterized by a set of *attributes* that may be either *measurable properties*, *part-objects*, *references to objects*, *procedures*, or *classes*. Finally, an object may have an *action-sequence* associated with it. Every object has at any given point in time a state. States are changed by objects performing actions that may involve other objects. Actions may in addition be involved in the production of measurements. A *program execution* consists of a collection of objects. Objects are classified into classes, and classes may be specializations of more general classes.

3.2.2 Study of Object-Oriented Languages

Having set the scene for object-oriented programming, we turn to the study of concrete examples of programming languages supporting object-oriented programming. Here we focus on the languages Simula 67, Smalltalk-80, Beta, C++ and LOOPS, and discuss hierarchical type systems[2] and delegation systems[18]. The study of the languages Smalltalk-80 and Beta is extensive and includes training in actual programming using the systems, whereas Simula 67, C++ and LOOPS are only evaluated theoretically. Here we have found that the theoretical foundation for approaching programming languages really has paid off. The students have very little problems in grasping the concrete language constructs presented in the different languages when they utilize the theoretical foundation as the basis for the learning process. They find that the notions that are handed to them in the theoretical part of the courses are in fact useful (although they might doubt it in the beginning of the courses).

It is not possible to discuss the application of the foundation in full details in this paper but to illustrate the issues, we discuss classes as models of concepts, class/subclass hierarchies as models of generalization hierarchies, and classes as aggregations.

3.2.3 Applying Object-Oriented Theory to Traditional Languages

Besides utilizing the theoretical foundation for studying object-oriented programming languages, we apply the foundation to traditional programming languages, such as Pascal, Modula-2 and Ada. In this way, we are able to gain additional insight into the foundation, but also to study the limitations of the support for object-oriented programming in the traditional languages. As an example, we discuss the relations between (generic) packages in Ada and abstraction (especially generalization).

3.2.4 Persistency

One of the major drawbacks of present object-oriented systems is that they do not support multi-user usage very well — they are essentially single-user systems (e.g. the Smalltalk-80 system). In order to support multi-user projects, persistent objects and shared program libraries must be supported within the object-oriented framework (i.e. object-oriented databases). The subject of object-oriented databases and persistence is presently not discussed in detail. Only a discussion of the underlying ideas and the reasons for the presently growing interest in the area is included. It is however mandatory that this subject is included in our courses in the near future.

3.2.5 Integration of Perspectives

The discussions of the various perspectives on programming give rise to discussions of possible ways for integration of the perspectives in one language. As already indicated, we find that some integration is both possible and fruitful. Since our perspective on programming is centered around object-oriented programming, we discuss how the other perspectives can be integrated in an otherwise object-oriented programming language. For this discussion the Beta programming language has been chosen.

As mentioned in section 3.1, the Beta language integrate the procedural, process, object-oriented and type system perspectives in one unified language. The limitations of these perspectives, and the elegance of certain solutions using the functional and constraint-oriented perspectives are constant inspirations to the discussions. We find that integration can be utilized to specify purely functional transformations on the states of objects, and to specify constraints on the interrelations between objects. Although not implemented, it seems to be possible to specify a purely functional subpart of Beta such that parts of a running Beta system is specified using the functional perspective. With respect to integration with the constraint-oriented perspective, various different approaches are being considered. The proof-of-existence can be found in the Smalltalk/V system[43] that contains a Prolog subsystem, but this system has not been found sufficient. With respect to the event perspective, we have found that the process perspective should be chosen as the pragmatic approach to multi-sequential programming, using the event perspective as the inspiring theoretical foundation.

3.2.6 Integration with Related Subjects

Traditionally the study of programming languages have been integrated with aspects of compiler construction, formal language theory, machine architecture and mathematical semantics. The courses described here are also concerned with integrating with aspects of system development. As indicated above, the approach to system development and the approach to programming languages are related in such a way that selecting a programming perspective will have an impact on the system development process as a whole and to some degree on the resulting product. We discuss those relations and their impacts as an integrated part of the courses.

4 Courses

Several different courses based on the above premises have been given. The present line of courses at the department is described together with two courses given to people from the industry.

At the Computer Science Department we are offering two courses on programming languages. The two industrial courses are one on object-oriented programming and one on Smalltalk/V.

Bachelor level

The first course is at the third (and last) year of the Bachelor level program. The course is partly on systems development and partly on programming languages. The course is occupying 1/3 of the student program for a whole year of which the programming language part is apx. 1/2.

The part on systems development, includes the Jackson System Development Method(JSD)[11] and various approaches to prototyping. The programming language part covers the following topics: The programming process and conceptual modeling. Presentation of different perspectives on programming. Introduction to and practical training in Smalltalk-80, Scheme and Beta. The introduction to JSD is related to object-oriented programming where it is emphasized that JSD in fact is very close to an object-oriented methodology. It is discussed to what extent JSD may be strengthen by using an object-oriented language. Smalltalk-80 is also used as an example of an environment that supports fast prototyping.

As mentioned, the important issues in teaching is not teaching the actual languages. This makes it difficult to find good textbooks on the subject. Take for example Scheme. In the courses we want to demonstrate to what extent Scheme supports the various perspectives on programming. That is the teaching is concerned with features for supporting procedural programming, features for functional programming and features for object-oriented programming. The book by Abelson and Sussman[1] is an excellent book for a course on introduction to programming. However it is far to big to be used by students already familiar with programming. Other books on Scheme (and most books on Lisp in general) introduces the language feature by feature, and does not relate it to perspectives. The material used in the course includes (parts of) [4,19,16,42,30,13,20].

The reasons for using Smalltalk-80, Scheme and Beta are: Smalltalk-80 and Scheme are representatives of flexible, dynamic languages without static typing. This make them well-suited for exploratory programming. Beta on the other side is a language with a static type system and intended for production programming. Smalltalk-80 and Beta are representatives of the two major directions in object-oriented programming. Finally Scheme and Beta are languages that are not solely based on one perspective. This is in contrast to Smalltalk-80 that has little support for other perspectives than object-orientation.

Master's level

The second course is at the Master's level with a bachelor degree (including the above course) as the only explicit requirement. The course is on advanced features of programming languages with

most emphasis on programming language support of object-oriented programming. The course is occupying 1/3 of the student program for one semester.

The course covers the following topics:

- The programming process and conceptual modeling[13].
- Types, packages, generics, tasks from Ada.
- Various definitions of "object-oriented programming"[20,40,36].
- Inheritance, delegation and enhancement[5,18,29,23,10].
- Multiple inheritance[35,38,14].
- Modularization[21,33].

Furthermore, the Beta programming language is used extensively throughout the entire course. The references are indications of material used. The course is organized mainly as a seminar course where the students are giving oral presentations of selected topics with the purpose of opening discussions on the topics. At the end of the course, the students are asked to write a small report on a selected topic within the course.

At the end of the course it is evident that the students are very able in programming perspectives, their relative merits and application areas, and object-oriented design and programming. Finally, they are able to evaluate particular languages with respect to their relation to specific application domains. Their ability to actually construct programs using some specific programming language is not the subject of this course, but experience has shown that the students are becoming accustomed to learning new programming languages and use them effectively after a short learning period. We find this to be a contribution of our theoretical approach to learning the subject of programming languages.

Industrial Courses

In the industrial environment we have been given two courses on object-oriented programming. The first industrial course was called "Object-Oriented Programming" and has been taught twice. The courses were arranged by "Datalogforeningen" (an association of Danish computer scientists, mainly in industry). The courses were limited to members of the association, and in effect this meant that the attendants were all having a Master's in computer science from Aarhus University. They had at least 4 years experience in industrial settings and had only very limited previous experiences with object-oriented programming.

The course was two-days with lectures and discussions. The course material was journal and conference papers, and language descriptions (almost identically to the material used in the department courses). The subjects discussed were object-oriented programming as discussed above and the Smalltalk-80 and Beta programming languages with a brief discussion of the C++ language. The

second course had attached to it a one-day workshop one week after the course, covering exercises in actual programming using the Smalltalk-80 and Beta systems.

Since the attendants were computer scientists with extensive training in handling theoretical approaches to problems, we found that stressing the theoretical approach to object-oriented programming was very fruitful. The attendants were very active during the course and very many of the discussions were centered around applying object-oriented programming in real-life industrial settings. The following workshop showed that they were able to handle object-oriented design very well, and their primary problems could be traced down to problems in expressing these designs in the concrete languages and to problems in handling the systems.

The second industrial course was given as an in-house course. The course was on object-oriented programming using the Smalltalk/V system on IBM personal computers. The attendants were mostly engineers with no previous experiences with object-oriented programming but experienced in traditional procedural programming languages (Pascal and Modula-2) and assembler programming.

The course was two-days with lectures, discussions and class-room problem solving followed by an one-day workshop with one week delay, covering exercises in actually using the Smalltalk/V system working on relative small programming tasks in groups. The course material was the tutorial and reference book accompanying the Smalltalk/V system plus some articles from the Byte issue on Smalltalk-80[44].

Having no previous experience in object-oriented programming and most importantly not being used to extensive theoretical approaches, the attendants were in the beginning rather confused (Question: *When do we start learning something about Smalltalk/V programming?*). In the last part of the course they realized the importance of taking the broader view (Reaction: *Oh, that's why I need to think differently!*). Finally, in the workshop the attendants had their major problems in handling the Smalltalk/V system (they had nearly no previous experience using window/mouse based interaction), whereas they were able to take the initial steps in the direction of creating classification hierarchies. In time of writing it is known that at least two of the companies having representatives at the course are in the process of experimenting with object-oriented programming using either Smalltalk-80, Smalltalk/V or C++.

5 Final Remarks

Thus ended the story about teaching object-oriented programming at the Computer Science Department at Aarhus University. Computer scientists in the 21'st century will be forced to master several programming perspectives and a magnitude of languages supporting those perspectives. Without the theoretical understanding of the underlying perspectives they may run into difficulties. Mastering concrete programming languages and literacy in features will not be sufficient.

The most important single pay-off of the theoretical approach to the subject has been the ease with which the students have been able to access the concrete programming languages. When approaching new languages they have proven to be able to characterize the language in question

on basis of the theoretical foundation, and thus avoiding discussing features of one language compared with features in another language. That is, they make qualitative evaluations on basis of the theoretical foundation and not quantitative evaluations on basis of features.

Looking back, we have found that we as teachers (and to some respect as researchers) are lacking some profound knowledge within other research areas. Some of our work have strong connections to philosophy, linguistics, philosophy of science, and psychology. The danger is that we in those disciplines are somewhat amateur researchers. We have therefore been very conscious to tell the students that we are not experts in all these disciplines, and that our approach might have some defects if seen from those research disciplines. However, we hope to improve our knowledge in those areas. We are also aware that many of the interdisciplinary issues mentioned here are well-known to people working with knowledge representation within artificial intelligence and data bases, where modeling of real world phenomena are central issues. Our approach is not primarily directed towards issues of knowledge representation, but on issues of software construction. This implies, that we are primarily discussing the various aspects of utilizing object-oriented design principles in the context of software construction and not knowledge representation. This is the reason why we are discussing the object-oriented perspective in this very broad context of system development, program description, and program prescription as well as in the context of traditional languages like C, Pascal, Modula-2 and Ada.

6 Acknowledgements

The motivation to write this paper came from discussions in an ad-hoc working group at ECOOP'87 in Paris. Here it became evident that there is a great need for discussing how people teach object-oriented programming.

This paper reports on the experiences of teaching programming languages at the Computer Science Department at Aarhus University for more than 10 years. Many people have been involved in this process. Furthermore, the approach have been influenced by the last 15–20 years of research in system development and programming languages in Scandinavia. Proper acknowledgement of all these people is impossible. However, Kristine Stougård Thomsen must be mentioned for taking very actively part in the design and implementation of several of the courses. A special thanks must go to the large number of students who have actually taken the courses.

7 References

1. H. Abelson, G.J. Sussman & J. Abelson: *The Structure and Interpretation of Computer Programs*, MIT Press, 1985.

2. L. Cardelli & P. Wegner: *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, **17**(4), 471–522 (December 1985).

3. O.-J. Dahl, B. Myhrhaug & K. Nygaard: *Simula 67, Common Base Language*, Norwegian Computing Center, 1970.

4. A. Goldberg & D. Robson: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, 1983.

5. D.C. Halbert & P.D. O'Brian: *Using Types and Inheritance in Object-Oriented Programming*, IEEE Software, 4(5), 71–79 (September 1987).

6. P. Brinch Hansen: *The Design of Edison*, Software — Practice & Experience, **11**, 363–396 (1981).

7. C.A.R. Hoare: *Communicating Sequential Processes*, Comm. of the ACM, **21**(8), 666–677 (August 1978).

8. C.A.R. Hoare: *Communicating Sequential Processes*, Prentice-Hall, Inc., 1985.

9. E. Holbaek-Hanssen, P. Haandlykken & K. Nygaard: *System Description and the DELTA Language*, publication no. 523, Norwegian Computing Center, February 1977.

10. C. Horn: *Conformance, Genericity, Inheritance and Enhancement*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87), Paris, France, June 1987.

11. M.A. Jackson: *System Development*, Prentice-Hall Inc., 1983.

12. J. Lindskov Knudsen & K. Stougård Thomsen: *A Taxonomy for Programming Languages with Multisequential Processes*, Journal of Systems and Software, **7**(2) (June 1987).

13. J. Lindskov Knudsen & K. Stougård Thomsen: *A Conceptual Framework for Programming Languages*, Computer Science Department, Aarhus University, DAIMI PB-192, 1985.

14. J. Lindskov Knudsen: *Name Collision in Multiple Classification Hierarchies*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'88), Oslo, Norway, August 1988.

15. B. Bruun Kristensen, O. Lehrmann Madsen, B. Møller-Pedersen & Kristen Nygaard: *Syntax Directed Program Modularization*, in P. Degano & E. Sandewall (eds.): *Integrated Interactive Computing Systems*, North-Holland Publishing Company, 1983.

16. B. Bruun Kristensen, O. Lehrmann Madsen, B. Møller-Pedersen & K. Nygaard: *The Beta Programming Language*, in [31].

17. H. Ledgard & M. Marcotty: *The Programming Language Landscape*, Science Research Associates, Inc., 1981.

18. H. Liebermann: *Using Prototypical Object to Implement Shared Behavior in Object Oriented Systems*, Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), Portland, Oregon, September 1986.

19. B.J. MacLennan: *Principles of Programming Languages: Design, Evaluation, and Implementation*, CBS College Publishing, 1983.

20. O. Lehrmann Madsen & B. Møller-Pedersen: *What Object-Oriented Programming may be — and what it does not have to be !*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'88), Oslo, Norway, August 1988.

21. O. Lehrmann Madsen: *Block Structure and Object-Oriented Languages*, in [31].

22. L. Mathiassen: *Systemudvikling og systemudviklingsmetode* (in Danish), Computer Science Department, Aarhus University, DAIMI PB-136, 1981.

23. B. Meyer: *Genericity versus Inheritance*, Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), Portland, Oregon, September 1986.

24. R. Milner: *A Calculus of Communicating Systems*, Springer Lecture Notices in Computer Science, Vol. 92, Springer Verlag, 1980.

25. K. Nygaard: *Basic Concepts in Object Oriented Programming*, Sigplan Notices, **21**(10), 128–132 (October 1986).

26. K. Nygaard & P. Sørgaard: *The Perspective Concept in Informatics*, in G. Bjerknes, Pelle Ehn & Morten Kyng (eds.): *Computers and Democracy — A Scandinavian Challenge*, Avebury, 1987.

27. J. Rees & W. Clinger (eds.): *Revised*^{3} *Report on the Algorithmic Language Scheme*, Sigplan Notices, **21**(12), 37–79 (December 1986).

28. W. Reisig: *Petri Nets — An Introduction*, Springer Verlag, 1985.

29. D. Sandberg: *An Alternative to Subclassing*, Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), Portland, Oregon, September 1986.

30. B. Sheil: *Power Tools for Programmers*, Datamation, **29**(2) (February 1983).

31. B.D. Shriver & P. Wegner (eds.): *Research Directions in Object-Oriented Programming*, MIT Press, 1987.

32. J.M. Smith & D.C.P. Smith: *Database Abstractions: Aggregation and Generalization*, ACM TODS, **2**(2) (June 1977).

33. A. Snyder: *Inheritance and the Development of Encapsulated Software Components*, in [31].

34. B. Stroustrup: *An Overview of C++*, Sigplan Notices, **21**(10) (October 1986).

35. B. Stroustrup: *Multiple Inheritance for C++*, Proceedings of EUUG Spring '87 Conference, 1987.

36. B. Stroustrup: *What is "Object-Oriented Programming"?*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87), Paris, France, June 1987.

37. R.D. Tennent: *Principles of Programming Languages*, Prentice-Hall Inc., 1981.

38. K. Stougård Thomsen: *Inheritance on Processes, Exemplified on Distributed Termination Detection*, International Journal of Parallel Programming, **16**(1), 17-52 (1987).

39. D. Turner: *An Overview of Miranda*, Sigplan Notices, **21**(12), 158–166 (December 1986).

40. P. Wegner: *Dimensions of Object-Based Language Design*, Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Orlando, Florida, October 1987.

41. W.A. Wulf: *Languages and Structured Programs*, in R.T. Yeh (ed.): *Current Trends in Programming Methodology*, Vol. I, Prentice-Hall Inc., 1977.

42. *Scheme Manual (Seventh Edition)*, MIT, September 1984.

43. *Smalltalk/V: Tutorial and Programming Handbook*, Digitalk Inc., 1986.

44. *Special Issue on Smalltalk-80*, Byte, Aug. 1981.