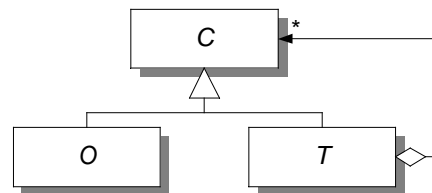


*A Conceptual Approach to
Teaching Object-Orientation
to C Programmers
COT/2-7-V2.0*



Centre for Object Technology

A CONCEPTUAL APPROACH TO TEACHING OBJECT-ORIENTATION TO C PROGRAMMERS

Ole Lehrmann Madsen, Henrik Røn, Kresten Krab Thorup & Mads Torgersen
The Devise Centre, Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Aarhus C, Denmark
Phone: (+45) 89 42 31 88, Fax: (+45) 89 42 32 55
{olmadsen,roen,krab,madst}@daimi.aau.dk

July 15, 1998

ABSTRACT

This paper describes experiences from a three-day course held as part of a collaborative research project, in which object-oriented programming was taught to a group of engineers who were all proficient C programmers. Our approach to teaching object-oriented programming focuses on conceptual modeling, emphasizing that object-orientation is not a bag of solutions and technology; rather, object-orientation is a means to understand, describe (model) and communicate. The paper describes a general approach to teaching object-oriented programming to C-programmers and reports on the experience from a concrete course.

1. INTRODUCTION

The Devise Centre at University of Aarhus is currently engaged in a large research project called Centre for Object Technology (COT), which is a joint project between industry and universities in Denmark. One of the research goals of COT is to investigate techniques for introducing object-orientation in companies. Despite the fact that there has been enormous interest in object-orientation for more than a decade, a large number of companies still do not use object-orientation.

Two of the companies participating in COT are using traditional methods like structural analysis and design (SA/SD) with C as the implementation language. One of the goals of COT is to replace SA/SD and C with object-oriented methods, tools and programming languages. To do this a number of activities including teaching, evaluation of tools, and pilot projects are carried out.

As part of COT, a three day course in the basic principles of object-orientation has been given to 24 experienced C programmers. The objective of the course

was to teach object-oriented programming as well as object-oriented modeling. The participants should be able to write simple programs during the course.

Aarhus University has more than 20 years of experience in teaching object-orientation starting with object-oriented programming using Simula [DMN68] and later Smalltalk [GR89], BETA [MMN93], C++ [Str86], Eiffel [Mey88], Scheme [AS85], etc. Later object-orientation was also introduced in other courses like system development, databases, and distributed systems.

Experiences from this have been presented in [KM88, KLMM93, KM96].

Object-oriented programming is taught with a focus on conceptual modeling, emphasizing that object-orientation is not a bag of solutions and technology, but a means to understand, describe (model) and communicate.

In this paper we first describe the general approach to teaching object-orientation and then relate our experience on using this approach with experienced C-programmers.

2. APPROACH TO TEACHING OBJECT-ORIENTATION

It is often discussed whether to teach object-oriented programming (OOP) before object-oriented methods (OOM) or vice versa. The argument for teaching OOP first is that the students should learn the basic principles of OOP before learning about analysis, design and modeling. The “bubbles and boxes” of OOM easily become too abstract if one does not know how they would be realized in an object-oriented programming language. The argument against teaching OOP first is that this easily focuses on technicalities of the programming language. Many courses in OOP are about teaching the syntax and semantics of an OO language and the students have difficulties in envisioning the potential of using OO concepts for modeling.

Some places teach OOP and OOM in parallel. This appears to be better than any of the above alternatives. In Aarhus we start with OOP but use a conceptual approach focusing on modeling. This is followed by a system development course based on OOM. In the following this approach is described in more detail.

2.1. The Scandinavian Tradition of Object-Orientation

Object-orientation started with programming, and the first OO language was Simula. Simula was originally designed as a means for writing simulation programs. When writing simulation programs, it is useful to have a language with good modeling capabilities. The notions of class, subclass and virtual function were discovered in order to describe complex real-world situations.

The first Simula language, Simula I, was designed as a simulation language, but it was soon discovered that the principles of Simula I could be used for general purpose programming. This led to the design of the programming language Simula 67, later renamed to Simula. The simulation origins of Simula have implied that modeling continued to be an important aspect of OOP with Simula. The first chapter in

[BDMN73] describes an approach to *system description*, which is the Simula term for modeling.

Although Simula 67 was a general purpose programming language, it continued to be used as a simulation language. Simulation was supported by “class Simulation”, which may be considered the first application framework. A simulation program is a model of a part of the application domain being simulated. One experience with using Simula as a simulation language was that the process of analysis, design, and implementation of a program (model) often was more beneficial for people than the actual simulation results, because the creative process of constructing the model gave a better understanding of the application domain. This led to the development of a system description language called Delta [HHN75]. Delta included elements such as assertions and time-consuming actions, and was not a programming language.

The Conceptual Framework. The modeling tradition from Simula and Delta was carried over to the development of the BETA language. The development of BETA consisted of two parts: development of a conceptual framework for object-orientation and development of the actual language. The conceptual framework [Chapter 18 in MMN93] was developed as a means to be used when modeling phenomena and concepts from the application domain. The BETA language is designed to support the conceptual framework. For a construct to be included in BETA it generally had to fulfill two criteria: (1) it should be useful for modeling and (2) it should add new technical expressiveness to the language. Although modeling was an important design goal, another goal for BETA was that it should also be attractive to programmers not interested in modeling.

An important goal for the conceptual framework was that it should be richer than the programming language. Designers often try to understand the application domain in terms of the actual language being used for analysis and design (or even the language

being used for implementation). In the OO world these languages often are formal graphical languages like OMT [RBPEL91] and UML [BJR97]. Such a formal language is only able to describe a small part of reality. With a conceptual framework richer than the formal languages, the designers have a good means for understanding the application domain. Of course, the designer eventually has to map concepts and phenomena into a formal language, but it is important that the designer is aware of the limitations of the formal languages, so he/she will not let his/her understanding of the application domain be restricted by these limitations. The designer should be aware of when a phenomenon or concept can be modeled directly and when it has to be changed a little in order to handle the shortcomings of the language.

A Conceptual Approach to Teaching Object-Oriented Programming. The tradition of emphasizing modeling in object-orientation has evolved at universities in Scandinavia over the last 30 years, and is reflected in our approach to teaching OOP.

The first thing we teach students is what we call the conceptual framework for object-orientation: Notions of concepts and phenomena, identification of objects, identification of classes, classification, generalization and specialization, multiple classification, reference- and part-of-composition, etc. This material is taught in a general setting not dependent on computer programming or a particular programming language. In this way, the mechanisms of the object-oriented way of understanding, i.e., the conceptual framework, are understood in their own right, and it is easier to separate the different levels of modeling that lie in the process of OO development. See [Chapter 18, MMN93] for details on this.

We have found that it is important to start out with assignments that lie within domains that the students are familiar with. By using problems from every-day domains, such as restaurants, vending machines, or bank accounts the students can more easily

start using the conceptual framework. With well-known problem domains, there is little or no need for knowing and using OOM. It is easy to start modeling, e.g., a vending machine; and it is important to do this early in order to realize that there are different levels of expressiveness: The actual real-world problem being modeled, the student's perception of what is being modeled, and various more or less formal ways of expressing this insight.

Learning a conceptual framework in a programming course prepares the students better for learning OOM later, because they have a framework for coping more easily with the less concrete programming languages herein. In general OOM is aimed at coping with the problems that lie in understanding a new domain, so it is of great value to the students if they are already familiar with the conceptual framework before they engage in learning methods.

Using the conceptual framework as the foundation for the object-oriented programming course gives the students a generally applicable knowledge, which is independent of any concrete programming language. This is very powerful and useful in many other settings because it gives the students a tool to critically understand and evaluate new object-oriented programming languages and methodologies.

System Development Course. The OOP course is followed by a system development course where emphasis is on the system development process. OOMs are an important part of the course, but at this stage the students are familiar with the basic principles of OOP and OO-modeling, and the system development course can therefore concentrate on teaching the students how to perform OOA and OOD in a new domain. Participatory design and rapid prototyping are taught as primary elements of the course. The students continue to use the same language and tools as in the programming course and they have to go through at least two cycles of analysis, design and implementation and develop prototypes together with real users. See

[KM96] for more details on usage of OO in this and other courses at Aarhus University.

3. THE SETTING FOR THE COT COURSE

The plan for the introduction of OO in the companies participating in the COT project was, for each company, to do a small pilot project followed by a larger main project. The industry employees have different backgrounds, but most have a B.Sc. or M.Sc. degree in Engineering. All participants are proficient C programmers, but the vast majority had no knowledge of OO, and the few that did, had no experience with using it in their everyday work.

Therefore a three-day course was organized in order to prepare the employees for the pilot project. The industry managers demanded that the focus should be on teaching the basic concepts of OO, rather than a specific programming language. Some of them had already had good experiences with this in courses at the Devise Centre, and it had not yet been decided which languages and tools to use in the subsequent projects.

The course should not deal with OOM specifically. Rather, this should be introduced on demand later on, during the pilot projects, and applied in close cooperation with very experienced OO specialists.

4. CHOICES

The course had three focal points: modeling and conceptual framework, OO programming language constructs, and OO programming. These were all influenced by our main goal of teaching general concepts and modeling rather than the syntax and semantics of a concrete OOPL.

4.1. Modeling and Conceptual Framework

Because we consider modeling and conceptual framework to be the most important aspects of teaching object-orientation, these were covered early in the course. The subject of the very first lecture was “The Conceptual Framework for

Object-orientation,” making it understood in its own right, as the foundation of object-orientation. In this way, the participants were equipped with a terminology for the central notions of modeling right from the beginning.

For each conceptual notion, examples of how to represent this notion in OO notations were given. When presenting, e.g., the class construct for representing concepts from the application domain, we used a graphical as well as a textual representation of the syntax. The graphical syntax was just some informal graphical notation, but could as well have been, e.g., UML. The textual syntax was very close to Java. In some cases we were presenting constructs that are not directly representable in Java. One example is part objects. The motivation for using a graphical as well as a textual syntax, was to emphasize that the main difference between OOA/OOD languages and OOP languages is a matter of syntax. We often experience that people have difficulties in realizing that OOA/OOD and OOP languages are based on (almost) the same set of abstract language constructs.

When the object-oriented constructs of a particular programming language are introduced later in the course, the conceptual framework will enable the participants to use them in a disciplined way for modeling, rather than arbitrarily exploiting the technical possibilities they present.

The conceptual framework might be taught on its own to the management and other non-technical personnel, giving them an abstract language to communicate with their engineers about object-oriented ideas; see e.g. [CCDHMMSST98].

4.2. Programming Language Constructs

Building on the conceptual framework, the participants should be taught the constructs that we consider the core of class-based OOPLs: object, class, method invocation, inheritance, polymorphism, genericity and virtual methods. These are the mechanisms that are generally available for describing

object-oriented designs. The idea was to teach these constructs as generally as possible in order to be language independent, but as mentioned exemplify using both informal graphical notations and syntax in a concrete programming language.

As one of our objectives was that the participants should write programs from day one, and as they all are experienced C programmers it seemed natural to choose a language with a C-based syntax. We had initially considered languages like Smalltalk or BETA, but feared that this might not work for a three-day course, as the participants would have to adapt not only to the new OO knowledge, but also to a new syntax, which would divert time and energy away from solving the assignments. When choosing a language with a C-based syntax, Java was preferred to C++ and Objective-C, since we think that Java is the simplest of these languages and, more importantly, the safest. A further argument for Java was that the participants might find it interesting to start writing Java Applets after the course and in this way continue programming in their spare time.

The course should not be a Java course, but primarily teach OO concepts. Some people from the participating companies had been on a Java course, which they described as a disaster since its focal point was teaching Java syntax and not OO concepts. To avoid this problem we were very careful only to teach the participants the most basic elements of the Java programming language and environment, just enough to be able to do the programming assignments. As examples of this, we did not teach the participants about such things as packages, exceptions, or any use of the standard Java libraries.

The participants are all working in the domain of embedded real-time systems. We thus found it important to teach concurrent object-oriented programming. Unfortunately there is no common agreement in the OO community on the central concepts and constructs of concurrent object-oriented programming, i.e., the conceptual model for concurrency so to speak. Because of that, this part of

the course was more Java specific (it was based on [Lea97]) than the part on the sequential OO language constructs.

4.3. The Programming Assignments

We wanted the participants to be able to start writing programs already on the first day, so the assignments should be doable with very limited programming skills. A problem with Java is that the use of standard libraries forces the programmer to deal with technicalities such as packages, visibility rules and exception handling. The exercises were designed so that the participants would not have to worry about all this, for instance by providing a tiny library of input/output routines that would not generate exceptions, which the participants could simply include directly in their program. This turned out to work very well.

The assignments used everyday concepts and phenomena from domains everybody can relate to, e.g., modeling a vending machine, or bank accounts and customers. We find that it is very important to use this style of assignments for introductory OO teaching, because it allows the participants to use an OOPL for describing phenomena and concepts in a domain they already know, thus eliminating the frustrating phase of analyzing an unknown domain. Using an OOPL as a language, in the sense of a vehicle for communication, the participants can focus on the modeling process thus being more aware of how closely their program actually models the real world than they would be able to if the problem domain was new to them. This also emphasizes to the participants that OO is not a bag of solutions and tricks, it is primarily a means to understand and describe.

In this particular course we also chose problem domains for the exercises that are far from those of the participants' everyday work. Although we have no empirical proof of this, we believe it to be a valuable approach when teaching programming to people who already have programming experience. With exercises from their usual problem domain, there might be a stronger

tendency to approach the problems using the same patterns or architectures as they are used to. Using a general domain has the further advantage that the teacher will know as much about the domain as the participants.

The assignments were designed to compensate for differences of speed and previous OO experience among the participants. Each practical session consisted of two exercises: a quite straightforward modification of existing code in order to get used to the syntax of the constructs in focus, and a larger modeling exercise which required a nontrivial design effort, and had to be coded from scratch. The required workload was carefully weighted, so that everyone should have time to complete the first exercise and carefully consider the modeling aspects of the second. In addition the second assignment contained enough separately formulated problems to keep everyone busy for the duration of the session.

The participants were asked to group themselves in twos according to their previous OO experience. Design discussions were encouraged, and at least two of the four teachers were present at all times to answer questions and discuss the participants modeling decisions, check on the progress of the participants and assist them in relating their problems to the terminology of the conceptual framework.

4.4. Onsite course organization

As the course was staged at the University of Aarhus the participants were forced to use The X Window System on UNIX workstations with Emacs as the editor. As more than half the participants were used to PCs running Windows, we chose a minimalistic programming environment consisting only of Emacs and a shell window in order to make things as simple as possible. We chose simplicity for several reasons. Firstly, we did not want to use an advanced programming environment that does advanced stuff behind the backs of the participants, as we wanted them to be in full control at all times. Secondly, getting

acquainted with an advanced programming environment is a time consuming task and could divert time, concentration and energy away from the main objective, namely solving the assignments by using the concepts taught. Given the limited size of the exercises, this lack of programming tools proved unproblematic.

5. CONTENT OF THE COURSE

As described above, the course entailed three different “teaching perspectives”: conceptual framework lectures, specific Java lectures and practice sessions. This does not mean, however, that the course was organized accordingly, in three big lumps. Rather, we focused on one major subject at a time, taking the participants all the way through the different perspectives.

First, the participants would get a thorough introduction to a subject from a conceptual, language independent, point of view. This involved a strong emphasis on the modeling capacities of the constructs explained, and often touched upon aspects not directly expressible in Java. Based on this, the specific Java features would then be presented. These lectures were very pragmatic, drawing heavily on the participants’ familiarity with C syntax, and focusing on the simple cases. Finally, the code writing exercises would allow the participants to toy with the new constructs, and explore the interplay between their perception of a problem and the object-oriented realization they were creating by means of the language features.

The philosophy behind this iterative structure was that, on the one hand, changing the *subject* too often tends to be confusing, while on the other, changing the *level of abstraction*, and the nature of the ongoing activity, has a refreshing and stimulating effect. Working with one subject in a number of different ways, both passively and actively, the new knowledge is allowed to “settle” in the mind of the participant, and in our opinion better enables him/her to autonomously relate to it in a critical and creative way.

The course consisted of four such iterations. On the first day, after a short introduction, the basic notions of object, class and attribute (field and method) were presented. On the second day, first subclassing and then virtual methods and genericity were introduced, and the last day dealt with concurrency constructs such as active objects, threads and monitors.

6. EVALUATION

The objectives of the course were to teach the basic principles of OO including object-oriented programming and modeling and to enable the participants to write simple OO programs during the course.

The course was based on the idea of a conceptual approach to teaching OOP. We think that the course succeeded in teaching OO programming as well as modeling. The participants will later, during the course of the COT project, learn about OOMs, so we are not yet able to conclude anything decisive about this.

There was a strong focus on the conceptual framework, but the participants generally did not seem to have any problems grasping it. A few parts were perceived as being too abstract, and upon reflection these seemed to be the ones that were not accompanied by programming examples. The lesson learned is that all conceptual elements should be exemplified.

The organization of the course as a number of subject-defined sessions, with different activities in each, seemed to provide a good balance between abstractions and modeling, language constructs/syntax and practical work. The volume of each subject seemed generally to be fitting, although the subclassing and virtuality sessions might have been joined without overloading the capacity of the participants.

Almost all of the participants managed to complete as much of the exercises as we had planned for. We discovered very few problems with writing Java programs. Actually one participant said that we used too much time on syntax. The C proficiency of the attendees was definitely

important here, saving us lots of time-consuming focus on irrelevant language details. Also this approach makes it easier to discern what language constructs are specifically object-oriented. One could argue that using a C-based language may keep them with their bad habits and that it would be better to use a completely different language like Smalltalk or BETA. This is hard to know, but using a non-C-based language would require more time.

The organization of the exercises also seemed to work well. By using well-known problem domains, it was easy for the participants to make models, and the simple exercise starting each session seemed to make the participants sufficiently familiar with the syntax to be able to cope with the more ambitious modeling exercise. Since there were always at least two teachers available during the exercises, the participants were never stuck with a problem. It is especially important that small problems like syntax errors can be solved immediately.

In general the course was well received. Most of the comments from the participants were positive.

One participant wanted more emphasis on design using graphical notation like UML. We are not convinced that UML should be introduced at this level, but we think that it might be useful to use some informal graphical notation to illustrate design. For the exercises we consider to ask the participants to make use of drawings to illustrate their solutions. At this level we think that it suffices to use an informal syntax for class hierarchies and associations. At a later stage when people are familiar with the basic concepts, a formal notation like UML can be introduced.

Although we attempted to minimize the focus on Java, it was impossible to avoid discussion of the applicability of Java for embedded real-time systems, since this is the application domain of the participants.

One problem with a course of this kind is that there are no good textbooks available. There are plenty of OOP books but almost all of them concentrate on learning a

specific OOL. Most OOP books do not present a conceptual framework and do not relate OOP to modeling. The OOM books are better on modeling, but lack OOP. We thus ended up using material from a number of different sources, and that was not satisfactory.

7. ACKNOWLEDGMENTS

Several participants in COT have made useful contributions in the preparation of the course. We would like to thank the participants in the course for their enthusiasm throughout the course and for their many constructive comments, and also Michael Caspersen for valuable comments on this paper. COT is sponsored by The Danish National Centre for IT-research (www.cit.dk) and the Danish Ministry of Industry.

REFERENCES

- [AG96] K. Arnold, J. Gosling: *The Java Programming Language*, Addison Wesley, 1996.
- [AS85] G. Abelson, G.J. Sussman with J. Sussmann: *The Structure and Interpretation of Computer Programs*, MIT Press, 1985.
- [BDMN73] G. Birtwistle, O.-J. Dahl, B. Myrhaug, K. Nygaard: *Simula Begin*, Studenterlitteratur (Lund), and Auerbach New York.
- [BJR97] G. Booch, I. Jacobsen, J. Rumbaugh: *Unified Modeling Language, version 1.1*. Rational Software Corporation, Santa Clara/CA, September 1997.
- [CCDHMMMSST98] M. Christensen, A. Crabtree, C. H. Damm, K. M. Hansen, O. L. Madsen, P. Marqvardsen, P. Mogensen, E. Sandvad, L. Sloth, M. Thomsen: *The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping*, ECOOP98, Brussels, Belgium, Springer Verlag, 1998
- [DBM68] O.-J. Dahl, B. Myrhaug, K. Nygaard: *Simula 67 Common Base Language*, Norwegian Computing Center, Oslo, 1968.
- [GR89] A. Goldberg, D. Robson: *Smalltalk-80, The Language and its Implementation*, Addison Wesley, 1989.
- [HFN75] E. Holbæk-Hanssen P. Håndlykken, K- Nygaard: *System Description and the Delta Language*, Norwegian Computing Center, Publ. #523, 1975.
- [KLMM93] J.L. Knudsen, M. Löfgren, O.L. Madsen, B. Magnusson: *Object-Oriented Environments – The Mjølner Approach*, Prentice Hall, 1993
- [KM88] J.L. Knudsen, O.L Madsen: *Teaching Object-Oriented Programming is more than Teaching an Object-Oriented Language*, ECOOP'88, Oslo, Springer Verlag, LNCS-322, 1988.
- [KM96] J.L. Knudsen, O.L Madsen: *Using Object-Oriented as a Common Basis for System Development Education*, OOPSLA'96 Educators Symposium, 1996.
- [Lea97] D. Lea: *Concurrent Programming in Java*, Addison Wesley, 1997.
- [Mey88] B. Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1998.
- [MMN93] O.L Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison Wesley/ACM Press, 1993.
- [RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall, 1991
- [Str86] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, 1986.