

The learners' environment in OO first: implications for sequence of teaching and tools¹

Jens Kaasbøll

Department of Informatics, University of Oslo

P.O.Box 1080 Blindern, 0316 Oslo

jensj@ifi.uio.no

Abstract

The sequence of instruction and appropriate tools for novice learners of object-oriented programming has been extensively debated. This paper analyzes the learning environment for OO programmers, and presents a series of studies aimed at finding the sequence of instruction and the corresponding tool that best supports the learning of object-oriented programming, starting with objects first. The studies indicate that starting out with a tool with visualization of object behavior and predefined classes in a toy domain is preferable. After having learnt the basic OO concepts, the learners can proceed to an environment that supports definition of classes for modeling a real world domain.

1. Introduction

Debates on the contents, sequence and tools when teaching object-oriented programming to novices have been going on for years, see e.g. [8], and the arguments have mostly been based on experience of teaching particular courses. Theories of learning seem to agree on that the learners' environment is of prime importance, regardless of whether it is called stimuli and reinforcement or social setting. In order to advance the knowledge of learning and teaching object-orientation, a systematic account of the learners' environment and how it may influence learning is therefore required.

There are many aspects of the learning environment that influences learning, like individual vs. group work, teacher interaction, books vs. computers, labs vs. real world, etc. These are general aspects of learning of any subject matter, so general pedagogical knowledge such as the usefulness of teacher intervention when the learner is stuck, will therefore probably be valid also in computer science learning. Research on pair programming [17] is an example of an enquiry into the individual vs. group aspect of the environment where the subject matter is narrowed down to programming and the group work is of a particular type.

Recognising the effect of the general aspects of the environment also on learning object-orientation, finding the specific ones for learning object-oriented programming is important for two reasons, being that these might be the easier ones to change and that they may have profound effects on the learning. The first research aim is therefore to determine the aspects of the environment that are specific when learning object-oriented programming and the possible effects of these aspects on the learning.

After a summary of previous research in section 2, three empirical studies are presented in section 3, indicating that reducing the number of aspects can be favorable for novice learners.

¹ This paper was presented at the NIK-2006 conference. For more information, see [//www.nik.no/](http://www.nik.no/)

Section 4 draws implications for teaching based on the studies and on general lessons from pedagogical research. Based on these implications, five teaching experiments were conducted, and these are reported in section 5. The results from these teaching experiments constitute the basis for implications for design of tools for teaching and for the sequence of teaching, given in section 6.

Each of the empirical studies has been published in theses or conferences, so their detailed descriptions are found in the publications referred to. However, the common background and the joint lessons learned have not been published before.

2. Previous studies

Booth [5] carried out a phenomenographic study of programming learners, and she identified four levels of skill mastery:

1. Expedient. Producing a complete program from the outset by making use of an existing program
2. Constuctional. Recognizing details of the problem in terms of features of the programming language
3. Operational. First interpretation of the operations that the program has to perform. Then writing the program.
4. Structural. First interpretation of the problem within its own domain. Then structuring. Finally coding.

The steps that Booth describe point to the environment of student programming, consisting of the teacher's examples, the programming language, operations of the specification, the domain to be represented, and the concepts of program structure.

Barnes et al [1] suggested a four-stage process of problem-solving which they have used in computer science teaching.

1. Understanding: structuring and dividing, clarifying, finding sample I/O.
2. Design: finding related problems and solutions, checking against I/O.
3. Writing: completing, adaptation to problem
4. Review: testing, summarizing lessons

The process 1-4 seems to resemble skill level 4 according to Booth's study. Stage 1 implies paying attention to the domain and the operations, and in addition to the aspects mentioned by Booth, also I/O is included here. Stage 2 hints at the teacher's and possibly other examples, while stage 3 refers to the code.

While the programming language provides structure at the micro level, other concepts, e.g. modules, associations, sets, are higher level constructs. Instead of using Booth's term "operation", the more common term "functionality" will be used to denote what the specification says that the program should be capable of doing. These will therefore be grouped together. Based on these studies, the following aspects of the learners seem to be present when learning programming:

- Related examples
- Programming language and higher order concepts

- Specification of functionality
- Domain to be represented
- I/O
- Code and program structure

An important feature of object-oriented programming is that during program execution, objects are created and their methods executed, and this dynamic model is supposed to represent the corresponding changes in the domain. While this way of comprehending the program execution is in the imagination of the programmer and is generally not directly observable at the interface of the computer, textbooks and teachers encourage the students to apprehend the execution in this way and also to draw static models representing a snapshot of the execution. When learning OO programming, a seventh aspect is therefore included,

- Program execution

These seven aspects are called the “OO learning environment.”

Barnes et al [1] suggested a sequence of four processes for students to follow. Based on experience with programming, students can be expected to follow any path in any direction between any of the seven aspects listed and in any order, e.g., coding based on requirements and testing code against requirements. The seven aspects thus provide the opportunity for 42 such subprocesses of programming.

3. Empirical studies

3.1 Initial observation

An initial observation of students in an introductory OO programming course was carried out in order to obtain a rough idea of how the OO environment influences their work. The students were observed in labs for 18 hours while the observers took notes [13]. The students mostly kept their attention on coding, while imagining the real world domain and at times, and, specifically when triggered by the tutors, draw figures of the program execution. Attention to the other aspects was hardly observed. This lack of observations triggered the need for a more thorough study of the students’ habits.

Distinguishing between when the students were occupied with the Code and with the Programming language was impossible, so these two aspects are joined into one called Code when programming.

3.2 Measurement of time spent on processes

The second study was carried out in order to measure how much time the students spent on each of the relations between the aspects [21]. Three pairs of first semester students were video recorded while solving programming problems for four hours, and both the students and their screens were captured. The recordings were transcribed.

During these observations, attention to user interface was not observed.

The processes between the aspects were named as follows:

Process	From	To
Modelling by coding	Code	Program execution
Design based on requirements	Specification	Code

Checking requirements against	Code	Specification
Adjustment of the model	Program execution	Code
Reality checking	Program execution	Domain

All three pairs spent most time on Modelling by coding. Design based on requirements and Checking against requirements were also observed a substantial number of times for all three pairs.

The quickest of the pairs spent less than half the time of the medium speed pair on the task, and the quickest pair also constructed the most complete program. This more advanced pair of students spent relatively much more time on Checking against requirements and relatively less time on Modelling by coding. The more advanced also carried out Reality checking and Adjustment of the model, while these processes of inference were not observed in the other groups.

3.3 Observation of class modelling

A third study was carried out on students learning class modelling prior to programming [13]. Four groups of approximately 20 students were observed during two sessions of 90 minutes, in which groups of 2-4 students were collaborating. One group of students at a time was audio recorded, and these recordings were transcribed.

Instead of Code and Programming language, the students were working with Class model and Modelling language. Program execution has its counterpart in the Instance level when modelling.

I/O was not considered here, since the tasks did not include programming or interface design. The semiotic analysis performed differentiated between Technical language and Everyday language used by the students; the latter used for talking about the Domain and the Requirements, while the Technical concerned the Class model. Also, the analysis distinguished between conversations about the Class level and the Instance level,

The students discussed between all these aspects, and inferences were drawn between them.

4. Conditions for learning OO

The second study indicates that more advanced programmers have a larger repertoire of aspects and processes than novices, who seem to be restricted to one or two aspects in their programming. The third one demonstrates that novices when modelling can handle the four aspects plus the modelling language.

While the modelling language is restricted to classes, associations, methods and attributes, the programming language contains the full set of concepts needed for instruction of computers. The class concept creates an additional topic to learn compared to procedural languages. Having relations to the other N concept of the language, the class concept thus does not just introduce one additional concepts, it also adds N relations to be grasped. The environment for programming therefore seems to be more complex than that for modelling.

In order for learning to take place, there has to be a limited number of new challenges [18]. Seven aspects and a full fledged OO programming language seem to be beyond the manageable size.

Given a real world problem and being asked to define classes is a challenge even when only one class is to be defined. Being given ready made classes for instantiation therefore constitute a smaller

first learning step. Creating subclasses of the ready made ones might provide an additional step between manipulating objects and defining classes from scratch.

General pedagogical research [17] shows that immediate feedback supports learning better than delayed, implying that programming tools that display the program execution immediately after coding is the better choice. Precise feedback is similarly better than the general one, meaning that a programming environment that displays the execution of each step and object in a program is a better learning tool than one which lumps steps or objects together. Visualizing phenomena that otherwise are hidden or imaginary enhances learning. The program execution with its objects, references and method calls is such a phenomenon which provides a consistent account of execution, but which remains imaginary when using professional programming tools.

Studies in education point to motivation as a crucial factor in learning. Learners are used to technology like computer games, so ASCII I/O will probably be boring, while tools that allow examples of more interaction and richer display might enhance motivation.

5. Tools

This section will assess available environments for OO training according to the conditions in the previous paragraph and present teaching experiments carried out by means of selected ones.

5.1 Minimizing the number of aspects

Karel J Robot [3] and Robocode [20] are programming environments that visualize the objects by graphical moving symbols, such that the learners do not have to imagine objects that are supposed to exist inside the computer. Also, there is no external Domain and Requirements can be omitted leaving the students to play around according to their own wish, thereby reducing the number of aspects to two only, Code and Program execution.

Both tools were used in experiments; Karel J for novice programmers and Robocode at the end of a first semester OO programming course.

Six novice students chose to attend a pre-university course of three days on object-oriented programming [7]. Karel J was used during the two first days, each lasting for six hours. Four of these students had never programmed before. Two observers took notes, and one of the groups was video recorded.

They all learnt to generate objects, extend the given robot class with their own methods, generate objects of this subclass and trigger the methods, during their first course day. These skills were elaborated with the basic algorithmic control structures on the second day.

Learning the basics of OO in one day without any programming experience is remarkable, so the immediate and precise feedback seems effective. Also, the visualization of objects and only two areas of attention contributed to establishing a simple platform for programming, allowing the students to concentrate on the basic OO concepts. The students also expressed that Karel J was motivating.

Robocode was taught to a group of three volunteer students who had completed their assignments of a 5-10 classes program in their introductory OO course [6]. The students were observed, and notes were taken. The students invented algorithms in order to improve the behavior of the robots. Achieving improvements was difficult, however; it seemed that the tool was more suited for learning sophisticated algorithms for movement and behavior in a war game than for learning basic OO concepts and programming. Due to the speed of execution and parallel processing, the learners could in general not

see the connection between their code and the movements of the robots. Although the students felt that Robocode was motivating, its lack of precise feedback and requirement of advanced algorithms makes it unsuitable as an OO learning tool for novice programmers.

Lejos [16] is a Java tool for programming Lego robots with classes Motor, Sensor, etc. No subclassing is needed. Similar to Karel J and Robocode, the tool requires two aspects: the code and the output, which in the case of Lejos consist of physical motors and sensors that control physical movement. However, the feedback from Lejos is somewhat delayed, as the learners have to code the program at the computer, transfer it to the robot, put the robot on the floor, and then start it. This may take a minute when everything runs smoothly. When the robot is running on the floor, it is impossible to watch the program at the screen simultaneously. The speed of the robot enables the learner to follow its movements when concentrating, but the operation of the light sensitive sensors is at times difficult to follow.

The Lego robots were used in order to boost motivation when teaching OO to a class of 28 children of age 14 (some were 15) in compulsory, lower secondary school [10]. The learners were observed, notes were taken, and three hours of video were recorded and later analyzed. The teenagers expressed that Lego was for kids; nevertheless they completed the assignments, and the girls seemed to get bored quicker than the boys. In a previous experimental teaching for 11 year olds with the same teachers using Lego and imperative style programming, motivation was on top for three days. Learning programming with Lego robots thus seem to fit better in elementary school.

After two days of building Lego robots and programming them with Lejos, the 14 year olds had learnt how to generate objects and trigger methods, showing the feasibility of the ready made classes approach coupled with the minimal number of aspects.

5.2 Real world problems with ready made classes

While the objects in Lego programs represent tangible phenomena, the 14 year olds were subsequently given the more demanding task of handling objects that had no observable physical counterpart.

An environment was built in Java for simulating guests ordering dishes from waiters, chefs cooking, and waiters delivering the dishes to the correct tables [11]. Class `Guest` had methods such as `chooseFromMenu`, and Class `Waiter` had the methods `placeByTable(guest,table)`, `receiveOrder(guest)`, `findCorrectGuest(table)`, etc. The environment opens for generating objects and writing method calls that closely resembles the business logic that the pupils have experienced when visiting restaurants. No visual representation of the objects appeared at the screen. When executing the programs, ASCII output such as the following appeared:

```
Hanne has received Ice cream as a desert. And this is correct.  
Per has received Salad as a main course. Incorrect. Per ordered Beef.
```

The learners thus had to relate to five aspects: the Requirements, Code, Domain, Program execution and Screen display.

The 14 year olds managed to learn manipulating objects and methods in this setting also, after they had performed the Lego Robolab tasks.

5.3 Visualization of program execution

BlueJ is an integrated development environment where the class structure is visualized graphically in an UML class diagram style [2]. When running programs made in BlueJ, the objects appear in a window, and learners can trigger the methods of the objects and view the object attributes and values. This

quality enables an immediate and precise experience of the relation between code and program execution. However, references are not visualized.

BlueJ has a few ready-made classes, from which the learners can create objects, some of which provide graphical output. Thus, the tool provides an environment both for using classes and constructing one's own. However, it is beyond the capability of the average beginner student to provide graphical views on completely new problem domains. In the object view, the user can click on an object to access the attributes of that object.

An introductory course in OO programming with BlueJ at the linguistics department at a university was observed, and teachers and a random sample of seven students were interviewed some months after the course had finished [12]. The interviews were audio recorded and notes were taken.

Although the purpose of BlueJ and teaching based on objects-first is to give the students an early understanding and appreciation of objects and their role in the design of programs and in program execution, many students took a long time to understand the difference between classes and objects. The difficulty may be explained in part by the class-oriented nature of BlueJ's graphical main window, which is based on a UML-notation, and in part by the teaching generally putting so much weight on programming as a process of class description. The BlueJ book and course material do not place an early emphasis on finding objects and designing the program on the basis of these, thus using a BlueJ presentation may appear to the student to be class-oriented rather than object-oriented. These conclusions support those of [19].

5.4 Implications for tool design

The qualities of the tools are summarized in Table 1.

Table 1. Educational qualities of tools for learning.

	Given classes		Learner defined classes	
	Artificial domain	Real world	Artificial domain	Real world
Immediate and clear feedback	Karel J	BlueJ		BlueJ
Delayed feedback	Lejos			
Unclear feedback	Robolab	Restaurant simulation		

Although the tools are aimed at visualization, the domain being visualized does not necessarily correspond to learning requirements. When running an OO program, objects are created and linked to each other, and method calls are propagated between the objects without being visualized, while what learners can experience is the output produced by specific output statements. Understanding how your code determines program execution is paramount for being able to create a proper OO program, so visualizing the objects and methods in the program execution should be a goal for a learning tool.

While Lejos and Karel J visualize the behavior of the objects involved, these tools cater only to the specific execution model of their predefined classes or subclasses of those. BlueJ is the programming environment that comes closest to visualizing program execution, but the objects shown lack references,

and in general, the visualization of program execution seems to have been of low priority during the design of the tool, compared to the display of the class structure and coding environment.

Developers of environments for learning OO programming should consider enabling the execution of OO programs with a small number of objects to be visualized sensibly.

6. Sequence of teaching

In OO programming, the learners have to understand how to define classes, yet this skill has been found to be difficult to acquire. BlueJ is the most promising tool for learning how to structure OO programs, so this programming environment is recommended. If the learning objective is to understand the fundamental OO concepts, Karel J is best suited as a starting vehicle. It visualizes program execution and removes the burden of defining classes, while strongly supporting the learning of subclasses and methods. The recommended sequence of teaching introductory OO programming is therefore as follows:

1. Teach OO concepts with Karel J or a similar tool that provides visual, precise and immediate feedback of program execution, which encourages definition of subclasses and methods, but which does not favor learner-defined classes.
2. Teach learners how to use predefined classes in BlueJ or a similar environment that supports visualization of program execution.
3. Teach learners how to construct their own classes in BlueJ or a similar tool that invites making methods in the objects.

After learners have gained some confidence in defining classes, they can switch to the command line mode of operating the computer and also to more professional tools.

Learning a new tool consumes time that could have been spent on the subject matter instead, so introducing more than two tools during a semester is not recommended. One way of minimizing the number of tools is to integrate the environments, e.g., to run Karel J in BlueJ. Professional tools normally have a complexity that draws attention from the learning of programming to learning the tool itself. If such tools had an amateur mode that would hide the functionality that is unnecessary for novices and also could visualize program execution like indicated in Section 5.4, they could be used throughout an introductory course, thereby saving the burden of having to learn more tools.

The sequence of teaching is recommended based on studies of its parts, and no research has been carried out to evaluate this suggestion. Studies of this sequence of teaching, preferably compared to other sequences, should be undertaken. A first project could be to let a small number of learners go through the sequence and observe the development of their competence intensively. Only if such a study should turn out favorably, a comparison of this sequence of teaching with another one should be undertaken with a larger number of students.

The observation in Section 3.3 on modeling points to another possible teaching sequence. Since modeling entails fewer concepts than programming, and thus provides a simpler learning environment, starting out with modeling constitutes another option. Previous research demonstrates that this is feasible [14] and gives priority to learning of object structures on the expense of algorithms. A recent implementation of such a model-first approach to introductory OO programming has improved the previous success rate of 52% in a programming first approach to 79% after three years of developing an refining the model-first approach [3]. Model-first thus also seems to be a path worthy of further research and development.

7. Acknowledgement

This research has been enabled by a grant from the Norwegian Research Council to the project Comprehensive Object-Oriented Learning.

8. References

- [1] Barnes, D. J.; Fincher, S.; Thompson, S.. Introductory Problem Solving in Computer Science. In Goretti Daughton and Patricia Magee (eds.) *5th Annual Conference on the Teaching of Computing*, Dublin City University, 1997, 36–39
- [2] Barnes, D.J. and Kölling, M. *Objects First with Java: A Practical Introduction using BlueJ*. Pearson/Prentice Hall, Harlow, 2003
- [3] Bennedsen, J. and Caspersen, M. E. Programming in Context: a Model-First Approach to CS1. *SIGCSE Bulletin* 36 (1), 2004, 477-481.
- [4] Bergin, J.; Stehlik, M.; Roberts, J.; Pattis, R. *Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*. Dream Songs Press, USA, 2005
- [5] Booth, S. *Learning to program. A phenomenographic perspective*. PhD thesis, University of Gothenburg, 1992
- [6] Borge, R. *Teaching OOP using graphical programming environments: an experimental study*. Master thesis, University of Oslo, 2003
- [7] Borge, R., Fjuk, A. and Groven, A.-K. Using Karel J collaboratively to facilitate object-oriented learning In Kinshuk, Looi, C-K, Sutinen, E., Sampson, D., Aedo, I., Uden, L., Kähkönen, E. (Eds.) *The 4th IEEE International Conference on Advanced Learning Technologies*, Los Alamitos: IEEE Computer Society, 2004, 580-584.
- [8] Bruce, K.B. Controversy on How to Teach CS1: A Discussion on the SIGCSE-members Mailing List. *SIGCSE Bulletin* 37 (2), 2005, 111-116
- [9] Fjuk, A., Karahasanović, A. and Kaasbøll, J. *Comprehensive Object-Oriented Learning: The Learner's Perspective* Informing Science Press Santa Rosa, California, 2006
- [10] Granerud, R., Kaasbøll, J., Borge, R., Holmboe, C., and Smørdal, O. *Children's Understanding of Object-Oriented Programming* [9], 27–48
- [11] Hegna, H. *Using Active Objects in a "Model First" Approach to the teaching of Object-Oriented Programming*. COOL seminar, Norwegian Computing Centre, Oct. 2003
- [12] Hegna, H. and Groven, A.-K. *A study of objects-first with BlueJ in a non-computer science context*. [9], 81–112.
- [13] Holmboe, C., and Knain, E. A semiotic framework for learning UML class diagrams as technical discourse. Accepted for publication in *Systems, Signs and Actions*.
- [14] Kaasbøll, J. Introduction to Computing. Master thesis, University of Oslo, 1980
- [15] Kaasbøll, J.; Berge, O.; Borge, R. E.; Fjuk, A.; Holmboe, C.; Samuelsen, T. Learning Object-Oriented Programming. *16th Annual Workshop of the Psychology of Programming Interest Group*. Institute of Technology, Carlow, Ireland, 2004
- [16] Lejos. *Java for the RCX*. Retrieved December 20, 2005, from <http://lejos.sourceforge.net/index.html>.
- [17] McDowell, C., Werner, L., Bullock, H.E. and Fernald, J. Pair Programming Improves Student Retention, Confidence and Program Quality. *Communications of the ACM*. 49(8), 2006, 90-95

- [18] Ormrod, J.E. *Human Learning*. Merrill, Englewood Cliffs, NJ, 1995
- [19] Ragonis, N and Ben-Ari, M. On Understanding the Statics and Dynamics of Object-Oriented Programs. *SIGCSE Bulletin* 37 (1), 2005, 226-230
- [20] *Robocode*. Retrieved 15. January, 2006, from <http://robocode.alphaworks.ibm.com/home/home.html>
- [21] Sigernes, I. L. *Learning object-oriented programming: work processes in problem solving*. Master thesis, Department of Informatics, University of Oslo, 2005