Learning Object-Oriented Programming

Jens Kaasbøll[1] Ola Berge[2] Richard Edvin Borge[1] Annita Fjuk[2] Christian Holmboe[1] Terje Samuelsen[1]

1) University of Oslo, P.O.Box 1080 Blindern, N – 0316 Oslo, Norway. Tel: +47 22 85 24 10
{ jensj, richared, christho, terjesam}@ifi.uio.no
2) InterMedia, P.O.Box 1161, N – 0318 Oslo, Norway. Tel: +47 22 84 07 00
{ola.berge, annita.fjuk}@intermedia.uio.no

Keywords: POP-I.B. Barriers to programming, POP-II.B. Problem comprehension, POP-IV.A. Object-oriented design, POP-V.A. Attention investment, POP-V.B. Phenomenography, POP-VI.E Computer science education research

Abstract

Loud discussions concerning various ways of teaching object-orientation have taken place without much empirical evidence for any position. This paper reports qualitative observations of learning of object-oriented programming in an introductory course. The students were found to cope reasonably well with the object-oriented concepts, and they had learnt procedural programming first. However, when modelled the real world domain to be represented in the program, they imagined the model and coded it without explicit analysis and design.

Their problems may be attributed to the high complexity generated by the five different areas of attention the students have to cope with. In addition to representing the problem domain in the program execution, they have to design the other components of the program, like user interface and file handling, and relate these to the reality model.

Three ways of improving teaching are suggested, making the areas of attention and the ways to relate them more explicit for the students, forcing modelling by means of a tool, and reducing complexity by means of programming environments that visualize objects and their behaviour.

Introduction

Teaching programming and modelling by means of object-oriented methods has become a common mode of introductory computer science training during the last ten years. Some research indicates that learning object-orientation is more difficult than coming to grips with other paradigms (Vessey & Conger, 1994), while another recent study of programming learning showed no significant difference between procedural and OO learners (Wiedenbeck & Ramalingam, 1999). High drop out and failure rates in introductory programming at universities world wide also add to the impression of a subject difficult to teach and learn.

Previous studies of novices learning programming have investigated students learning procedural (Soloway & Spohrer, 1989) and logic programming (Booth, 1992). They conclude that after an introductory period, most students have learnt the syntax and semantics of the programming language, but they struggle with composing their programs. This would imply that courses which go beyond the simplest programs will constitute a much larger challenge to the students, regardless of the programming paradigm and the sequence of instruction.

When learning object-oriented programming, the language comprises more concepts than previously used languages, which could imply that learning this paradigm becomes more difficult. On the other hand, the object-oriented language includes the concept of class, which is supposed to ease design and structuring of programs (Coad & Yourdon, 1991; Nygaard, 1986). Accordingly, in courses with larger programming assignments, object-orientation should be an advantage.

This study aims at finding out how the students come to grips with the object-oriented concepts, how their understanding influence their learning of programming, and point to issues that students are struggling with.

Background

The students who are studied attend an introductory course in object-oriented programming with Java at university level. The sequence of instruction is variables, control structures, methods, arrays, file operations, graphical interfaces, classes, objects and pointers, object-sets, inheritance, sub- and superclasses, abstract classes and exception handling.

The course is scheduled to 50% workload during a semester, with four hours of lecture, two hours of classroom problem solving and two hours of supervised labs per week. The students are given five mandatory exercises during the semester.

Only a minority of the students has any prior programming experience.

Method

Since little is known concerning learning of programming, devising categories that lend themselves to quantitative studies are difficult, so a qualitative approach was selected. In order to elicit how the students understand programming, their conversations and programming were observed. Observation has long been used for studying learning of a diverse set of subjects, logic programming included (Marton & Booth, 1987).

The students were observed for eight hours in the lectures. A request for permission to observe the students in the classrooms and the labs was sent to the instructors, Three out of 11 instructors were positive, and the students in these groups were observed for a total of 20 hours, 18 of which were done in the lab.

During the observations, the observers also asked the students some questions, mostly concerning what they experienced as most difficult.

Notes were taken during the observations. Audio tape recording was not considered giving sufficient added value, since much of the conversation was concerned with code on the computer screen, and understanding the recording without the programs being discussed would be difficult. Recording the program code with video was considered impractical due to the small fonts and flickering screen on the computer terminals.

On some occasions, the observers also guided the students, taking over the role of the instructor. This happened when there were more students requesting assistance than the instructors could handle. Acting as an instructor provided opportunities for getting the students to talk about their programs, but the interaction implied that fewer comments from the students were noted down. The interactions were not planned interviews, but they might have focussed more on the issues that the researchers were interested in than would have happened if carried out by an instructor.

Results and analysis

Areas of attention

The tasks in the education studied concerned developing applications for domains that the students should be somewhat familiar with (i.e spectators in a movie theatre). When developing object-oriented applications, the students have to consider several domains. First, the students have to consider the real world domain that the application is supposed to serve (i.e the movies, the theatre with its rows and seats, and the audience). Second, the students have to consider the objects that the program will generate during run-time, and in what manner these objects as defined by the the classes “Customer,” “Movie” and “Seat” represent the actual customers, movies and seats. People can imagine that these objects exist in the computer during program execution, but they do not lend themselves to experience. Third, the interaction with the running program catches the students’ attention when running a syntactically correct program, and they can experience their own input as it appears in the windows on the screen. Fourth, the code with its class descriptions has to be written and discussed,

and this area tends to be in the focus of the students' attention during their labs. The classes constitute the prescriptors of the objects, the descriptors of the real world concepts, and the organizing units for the user interface.

Keeping these different areas in mind when programming may be trivial for the experienced programmer, who may consciously choose to disregard interface and objects for periods, and only concentrate on constructing class descriptions that represent generalized real world phenomena. But some of the students observed did not have any clear conceptions of objects during program execution, so they stored every change of the program on file. Another student struggled for two hours with obtaining consistency between the data entered at the user interface and what her program stored on file (JK 03.04.03)[1]. She could specify what the file should look like, but she was incapable of sketching the objects generated during program execution and the pointers, which kept the objects coherent. Five areas of attention that the students should consider are illustrated in Figure 1.

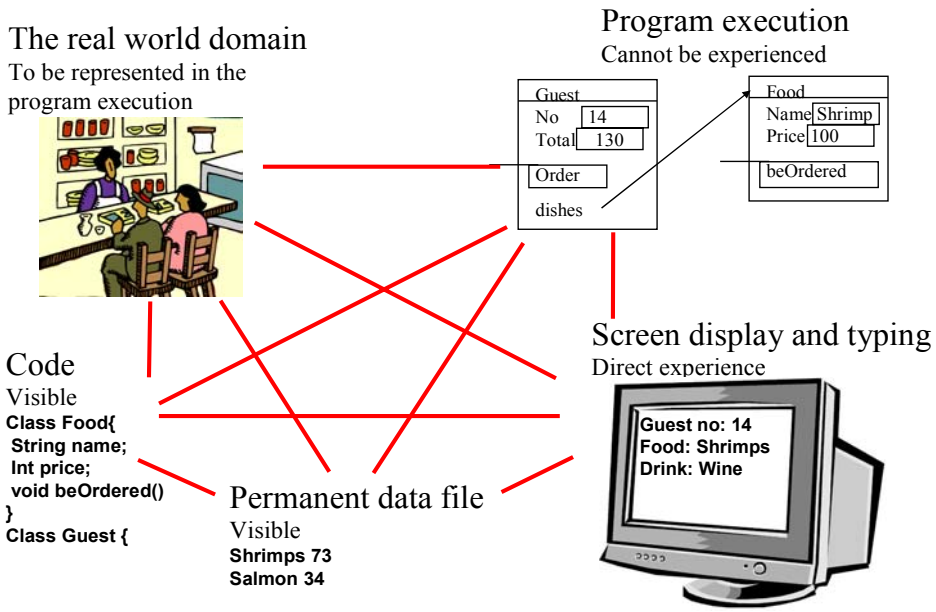

Figure 1. Five areas of attention for the programmer when constructing an object-oriented program.

Three of these areas are visible for the programmer - the code, the file, and the user input. The student has to recall the real world domain from her memory, which is probably trivial when the student is familiar with it. In addition, the problem specification as a discursive tool might function as a reminder (Säljö, 1999). The program execution with its objects is something that the student has to imagine, and the teachers advice the students to draw sketches like those depicted in the upper right corner of Figure 1

When the novices have to grapple with five areas of attention and the 10 relations between them, it should not come as a surprise that most students experience problems.

The student who struggled with getting the correct data on file was talking about her program in the following way: "the program reads from the keyboard and writes on the file," indicating that she regarded the world of computing as "input → processing in program → output," see Figure 2, and not as a set of interlinked objects that represent a real world problem domain. We would say that this student had a procedural and not an object-oriented conception (Détienne, 1997) when programming.

[1] Observer initials and date of observation.

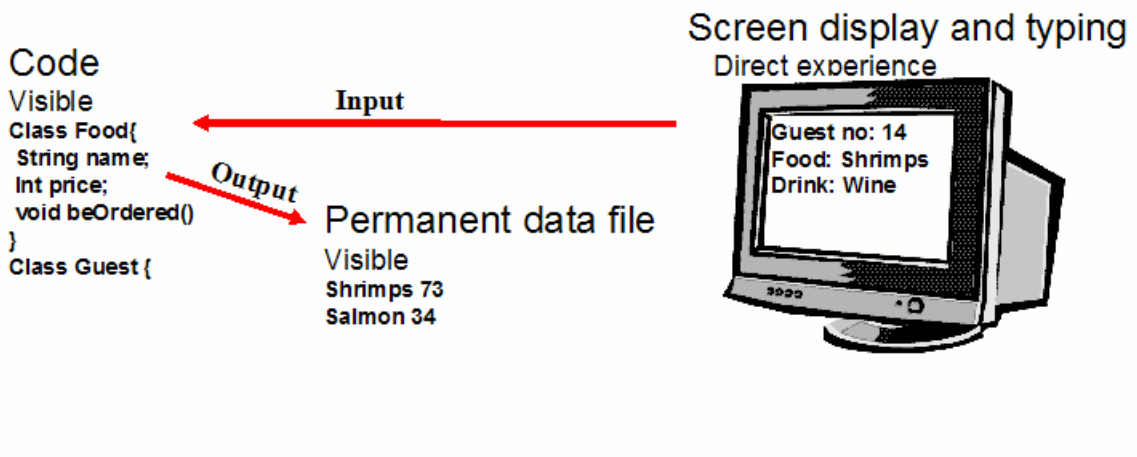

Figure 2. One student's conception of her program.

Most students had developed a more comprehensive framework of understanding for their programming. During conversations about the cinema program, the students referred to a "show" of a specific movie at a specific time as the object in the program execution, while never discussing the actual cinema that the program was supposed to manage. The instructor (I) reinforced this understanding in her conversation with the students (S) (JK 20.03.03):

> I: … similarities between shows … they contain a theatre. So we make an object for each show. Each show has a theatre, in a way, or our seats.
>
> (The instructor draws object structures)
>
> I: One like this which controls whether we have this one already.
>
> S: And that is the HashMap?
>
> I: Yes

If they were talking about the real world cinema, they would hardly speak of a show as something that contains a theatre, but rather about a theatre that hosts a show. Assuming that all students are familiar with going to the movies, they probably use this experience as a frame of reference when discussing the objects, even if their conversations do not mention the real world domain. Their framework of understanding can therefore be illustrated as in Figure 3.

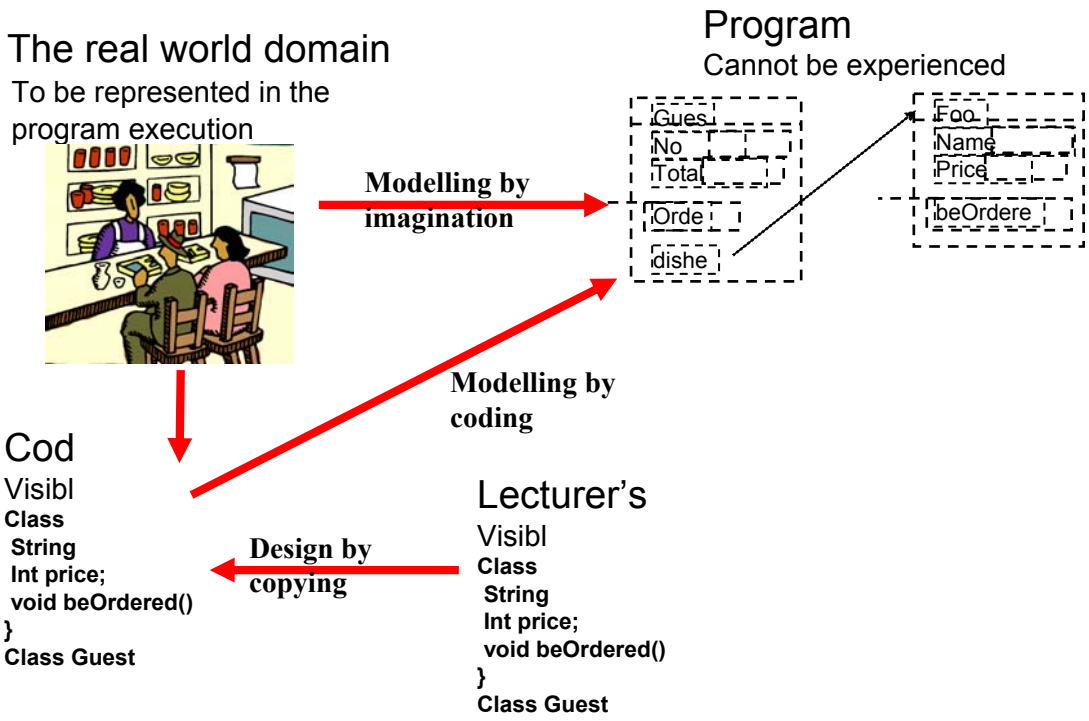

Figure 3. The students' areas of attention and directions of inference. The Program execution is dotted due to the students' habits of only imagining this area.

Relating the areas

The movies task studied was the students' first assignment of size more than a page of code and a somewhat complex domain. This increase in problem complexity implies two different skills to be appropriated by the students. First, they need to be able to model the problem domain (i.e. part of reality) by means of object-oriented concepts. We will refer to this activity as "modelling." Second, they have to master the structuring of their object-oriented program by means of classes and methods for file handling, user interface, etc., and we call this process "program design."

The concepts that the students had been introduced to – and thus had available for modelling – were objects, pointers, arrays, and HashMaps. These concepts were clearly implementation oriented, and no analysis concepts like associations and cardinalities, were known to the students. The observation that the students made design models rather than models of the real world domain, is therefore not surprising.

Moreover, only occasionally did the students draw models on paper, and even if they did, these models "disappeared" under other papers, or the students were not able to translate them into code. Their conversations about arrays and pointers were not anchored in these drawings, but rather in the visible code on the computer screens or code that could easily become visible through scrolling. Referring to Figure 3, they imagined the two upper areas rather than visualizing or experiencing them (Petre & Blackwell, 1999), so we can call their modelling activity "modelling by imagination."

When students modelling by imagination asked the instructor for assistance, she often replied by drawing what she perceived as the student's model, and commented upon it, e.g. (TS 27.03.03):

> I: Isn't it easier to put the seat into the theatre class instead of 100 seat objects?

The students who were unable to code based on the drawn model could only work in the upward direction from Code to Program execution in Figure 3, so we will call this activity "modelling by coding." The opposite direction, which the more advanced students could handle, would be "coding the model."

The students arrived at their program design by copying a program made by the lecturer and modifying it, we call this strategy "design by copying." (Booth, 1992, p 207) observed that students use the same strategy when writing their first program. Even when having imported the program structure from the lecturer, some students were unsure about how to proceed (TS 27.03.03):

> S: We are sure about what to do, but we don't know where.

This statement indicates that they have a certain mastery of the details without a complete understanding of the big picture.

Structuring the program

Being novices, the students needed lots of assistance in structuring their program, e.g. (JK 20.03.03):

> S: Is this OK?
>
> I: What are you gonna make? Have you thought about how you will structure your program? I don't think that I would have had that HashMap there. It is correct the way you have made it, but I would have had it another place. Smart to think about how you will structure your program.
>
> (The instructor draws in the student's book)
>
> S: No, I don't know what I thought.

Again, the student is imagined a model, and the instructor provides visualisation, forcing the student to rethink his design.

The students were not familiar with the power of pointers, objects, arrays and HashMaps, e.g., the instructor comments upon a student's array for pointing to seat objects in the theatre (JK 20.03.03):

> I: But it is three-dimensional!

Even though the students struggled with putting the pieces together, they all tried to use the set-concepts when storing many objects of a kind and never when creating only one object. Also, no students were reported mixing up Class with Array or HashMap.

When designing by copying, the students were told to use a Database class and a Control class, but these classes were frequently misunderstood (RB 24.03.03). Since the students had no intuition concerning them, they merely followed the lecturers' examples, hence there is ample room for misunderstanding.

Object-oriented concepts

Personal experience shows that the students' speed of learning in programming courses differ vastly. One example of the slow learners was a student who should program a class that took care of customer information, and he made his program generate a new customer-object for every piece of information for each customer (RB 31.03.03). Most of the students observed had a more sophisticated understanding of objects. They knew that many objects could be generated for a class, that each customer should have one object. Further, they knew how to declare private variables and public methods in the classes. Some students also demonstrated a good understanding of the difference between object and class (`static`) variables.
Given the code:

```
class Rectangle
{
  private double heigth;
```

```
  private double width;
  private static String colour;

  Rectangle(double h, double w, String c)
  {
    heigth = h;
    width = w;
    colour = c;
  }
}
...

Rectangle r1, r2;
r1 = new Rectangle(2,3,"Red");
r2 = new Rectangle(4,6,"Yellow");
```

The students were asked what height, width and colours r1 and r2 would have, and they said (RB 31.03.03):

S1: 3, 4 and yellow. And the other 4, 6 and yellow

I: Any other suggestions?

S2: Red and then yellow

I: Any other?

S3: Yellow and yellow.

I: Why

S: The colour is static

I: Yes, static means that the colour is the same for all the objects of this class.

Subclasses were introduced in the lectures during the time of observations, but most of the students had not started to use them in the programs. Questions during the lecture indicated a good understanding among some of the students (OB,AF,RB 31.03.03)

S: Do you need a constructor in the sub-class?

I: Well, yes, we do so here, due to the parameter

However, other students had not grasped the idea (TS 04.04.03):

S: Do all variables have to be declared twice in sub-classes?

Students also seemed to have grasped the generality of the class concept, e.g. (JK 20.03.03)

S1: Arrays are actually objects, see

S2: So what applies there applies here as well

S1: Yes

Pointers were also problematic in some instances. The students knew that they should create a pointer and assign it to the new object when created, but pointer assignments in general was more difficult (TS 27.03.03):

I: Why do you say (while pointing at the code, which was like):

```
Seat S = new Seat();
Seat T = new Seat();
S = T;
```

The student was not aware that the first Seat object would be lost.

Questions concerning most of the concepts in Java were also raised by the students. However there were few syntax related ones, and none of the observers overheard any questions concerning the basics of variables, if and while. While teachers have reported that parameters are on the top of the list of difficult to learn programming concepts, parameters were not mentioned more frequently than other concepts by the students observed.

Discussion

Coping with five areas of attention

Previous studies of learning procedural and logic programming have pointed to that most students learn syntax and semantics within reasonable time, and thereafter they struggle with learning how to design their programs. This study shows a similar pattern when learning object-oriented programming. However the students had learnt the basics of procedure oriented programming first, so a certain correspondence with the previous findings should be expected. A study of students who learn the object-oriented concepts first is needed in order to find out whether the pattern also applies in the pure object-oriented case.

Widenbeck, and Ramalingam (1999) distinguish between what they call *program model* and *domain model* in their study of novice OO programmers. The program model covers the elementary operations and the control flow, while the domain model concerns to the problem situation referred to in the program, specified as data flow and function information. In addition, they have studied students' understanding of program state. Their elementary operations seem to correspond to what is regarded as code here, while control flow, data flow and state are aspects of the program execution. Their functions are abstractions of input/output, referring to the screen display and typing in the areas presented here. While other studies (Détienne, 2002; Holmboe, 2003; Sharp, 1991) focus on the importance of knowledge of the problem domain, there is no specific mentioning of issues of the real world domain in (Wiedenbeck & Ramalingam, 1999). This might be due to the fact that the categories used were based on previous studies of procedural programming. A basic OO perspective of the real world domain described in code and represented in the program execution constitute a perspective that does not easily apply to the procedural approach. The procedural bias of the study is shown in its view of how programs work: "The transformation of data from an input state to a desired output state is the fundamental purpose of a program…" (Wiedenbeck & Ramalingam, 1999). This furthermore resembles the perception of one of our students as demonstrated above.

Object-orientation pose two challenges in learning program design. First, the students have to learn how to make a model of reality that their objects are going to represent, and second, they have to learn how to design the other components of the program, like user interface and file handling, and how these components should be linked to the reality model. The students studied were working on their first assignment where both kinds of structuring were necessary.

Concerning the modelling process, they worked with mapping implementation concepts like HashMap to the program execution that they imagined. Hence, we call this process "designing by imagination." Few signs of explicit checking against reality were performed. A possible way of strengthening the learning of modelling would be to introduce modelling of reality explicitly by means of object

diagrams that illustrate cardinalities. An additional modelling step may, however, also be counterproductive to student learning since it entails more to be learnt.

A study of professionals doing ER modelling showed that they did not consider the ER diagrams as a conceptual model of reality but as database design (Hitchman, 2003). If professional modellers do not distinguish between analysis and design models, there might not be any point in introducing this distinction in introductory courses. However, the professionals consistently checked their model with the reality, while the students seemed to have their attention mostly at the code and the program execution. A similar characteristic of novice expert differences have been documented in a recent study by Holmboe and Knain (Holmboe & Knain, 2004).

When structuring the rest of their programs, the students copied solutions presented by the teachers or found in the course material. Copying and modifying is a well known first step in learning programming, and it seems to work in this setting also. Therefore, the program design that the teachers preach should be tuned to the students' tasks at the novice level, being general enough to a large set of tasks, without bringing in unnecessary sophistication. The concrete model-view-control pattern that was applied in the course seemed to fit the students' tasks.

The students demonstrated a seemingly hazardous programming development by ignoring reality checking and by coding without visual models. While the instructors repeatedly assisted the students in drawing models, the students did not take up this habit, something that might have happened because they were unable to base their coding on the models. This inability may stem from the way of teaching, or novices may in any case find it easier to code based on an example and their conception of the domain of the program rather than on models. Explicit teaching of the areas of attention and the ways programmers think when making inferences from one area to the other (see Figure 4) might strengthen the students' abilities to use models more actively.

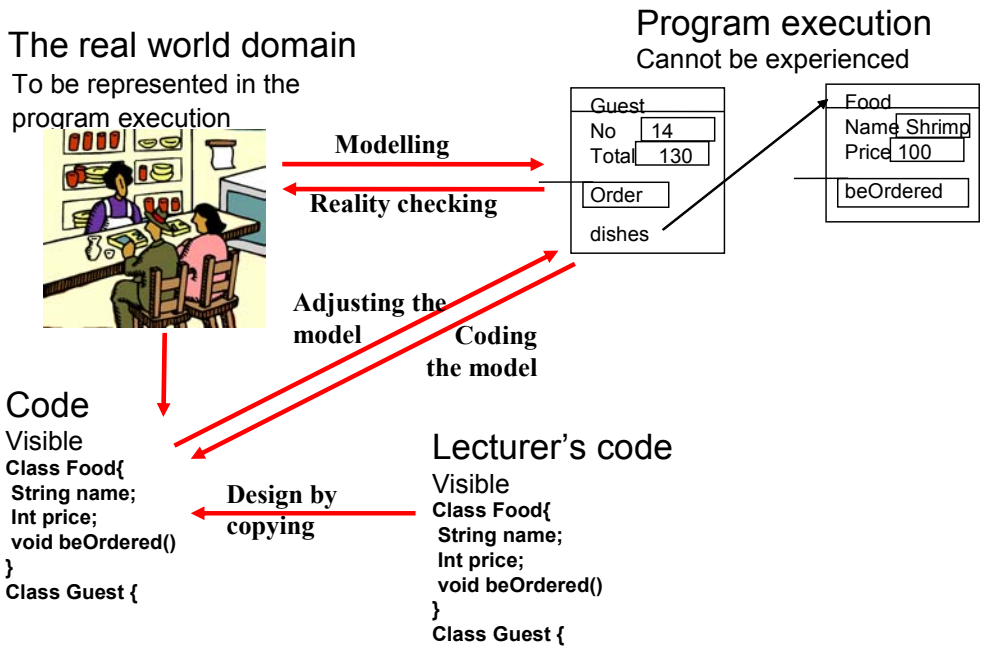

Figure 4. Areas of attention and ways of programming

The empirical study made by Widenbeck and Ramalingam (1999) shows that the worst performing OO novices make more errors on operation and control flow, while the procedural learners fail more on data flow and function. More advanced beginners do not exhibit these differences. They conclude that learners need both domain and program knowledge in order to boost their comprehension.

Although they use basically procedural categories of analysis, pointing to that students need to understand relations between several ways of regarding a program fits well with our observations of OO learners.

Simplifying the programming environment

An alternative approach is to reduce the number of areas of attention from the four in figure 4, and possibly five when bringing in user interface and files like in Figure 1, to only two, by means of special programming environments.

Karel J Robot (Bergin, Stehlik, & Pattis, 2003), Robocode (2003) and Alice (Cooper, Dann, & Pausch, 2003) are programming environments where the objects generated in the programs become visible, such that the students do not have to imagine objects that are supposed to exist inside the computer. Also, the students do not have to consider relations between phenomena outside the computer and the objects inside, thereby reducing the number of areas of attention to two only, see Figure 5.

Objects or procedures first

The students studied here learnt procedural programming first, which could provide some indication as to whether starting with the procedural paradigm has had any effect of the students' understanding of the object-oriented paradigm.

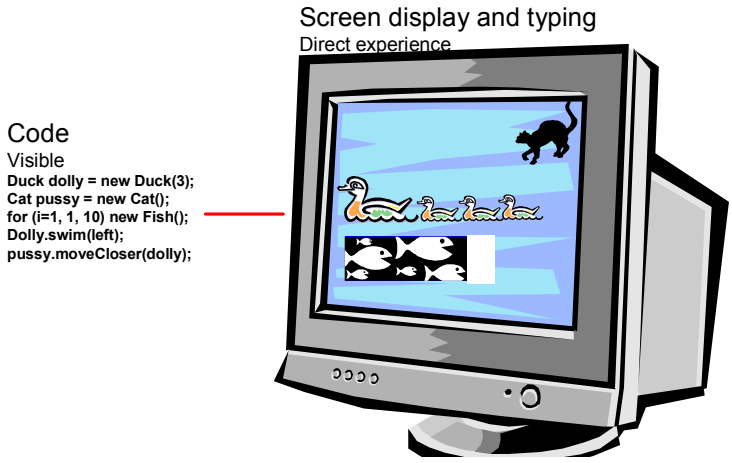

Figure 5. Areas of attention in special programming environments.

One student regarded the operation of her program as an input-transformation-output sequence, see Figure 2. This sequence corresponds to the procedural view and contradicts the object-oriented paradigm. No other student seemed to have such a conception, however, so teaching the procedural way first did not seem to have much impact in this respect.

Another interpretation of the students' poor performance of modelling is that modelling is not encouraged in procedural programming, so that they have learnt coding first. Instead of modelling being helpful for programming, it becomes an additional burden that has to be learnt together with the object-oriented concepts.

There are also programming environments that force modelling, and at the same time generate code skeletons from the models drawn (BlueJ, 2003). Such environments may also be useful for students who have started out with procedural programming, since they may turn "modelling by imagination" into "graphical/visual modelling" and "Modelling by coding" into "Coding the model."

References

Bergin, J., Stehlik, M. R., & Pattis, R., (2003). Karel J, Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java http://www.csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html

BlueJ, (2003). The Interactive Java Environment http://www.bluej.org

Booth, S., (1992). Learning to Program: A phenomenographic perspective, (Vol. 89). University of Göteborg, Gothenbourg.

Coad, P., & Yourdon, E., (1991). *Object-Oriented Analysis*, (2nd ed.). Prentice-Hall, Englewood Cliffs, NJ.

Cooper, S., Dann, W., & Pausch, R., (2003). Teaching objects-first in introductory computer science. *SIGCSE Bulletin*. 35(1), 191-195.

Détienne, F., (1997). Assessing the cognitive consequences of the object-oriented approach: a survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers*. 9, 47-72.

Détienne, F., (2002). *Software Design - Cognitive Aspects*, (F. Bott, Trans.). Springer, London.

Hitchman, S., (2003). An interpretive study of how practitioners use entity-relationship modelling in a ternary relationship situation. Communications of the Association of Information Systems. 11, 451-485.

Holmboe, C., (2003). Conceptualisation and Labelling as Linguistic Challenges for Students of Data Modelling. Submitted to *Computer Science Education*.

Holmboe, C., & Knain, E., (2004). A semiotic framework for learning UML as technical discourse. Submitted to *International Journal of Human-Computer Studies*.

Marton, F., & Booth, S., (1987). Learning and awareness. Lawrence Erlbaum, Mahwah, N.J.

Nygaard, K., (1986). Basic concepts in object oriented programming. *SIGPLAN-Notices*. 21(10), 128-132.

Petre, M., & Blackwell, A. F., (1999). Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*. 51, 7-30.

Robocode, (2003). http://robocode.alphaworks.ibm.com/home/home.html http://robocode.alphaworks.ibm.com/home/home.html

Sharp, H., (1991). The role of domain knowledge in software design. *Behaviour & Information Technology*. 10(5), 383-401.

Soloway, E., & Spohrer, J. (Eds.). (1989). *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Säljö, R., (1999). Concepts, cognition and discourse: from mental structures to discursive tools, in: W. Schnotz, S. Vosniadou & M. Carretero (Eds.), *New Perspectives on Conceptual Change*. Pergamon, Amsterdam, pp. 81-90.

Vessey, I., & Conger, S. A., (1994). Requirement specification: learning object, process, and data methodologies. *Communications of the ACM*. 37(5), 102-113.

Wiedenbeck, S., & Ramalingam, V., (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*. 51, 71-87.