From Procedures to Objects: What Have We (Not) Done ?

Jorma Sajaniemi and Marja Kuittinen

University of Joensuu, P.O.Box 111, FI-80101 Joensuu, Finland
{saja|marja}@cs.joensuu.fi

Abstract. Programming education has experienced a shift from imperative and procedural programming to object-orientation. This shift has been motivated by educators' desire to please information technology industry and potential students; it is not motivated by psychology of programming nor by computer science education research—there are practically no results that would indicate that such a shift is desirable, needed in the first place, or even effective for learning programming. Moreover, there has been an implicit assumption that classic results on imperative and procedural programming education and learning apply to OO programming, also, but we argue that this is not the case and, therefore, call for systematic research into the fundamental cognitive and educational issues in learning and teaching OO programming.
In order to understand the huge shift from imperative and procedural programming to object-orientation, we compare these paradigms at three levels: notations of languages, the notional machine that describes how programs are executed, and the orientation of a paradigm describing what programs are for and what can be done with them. We will also review research literature and see how it supports our claims. Finally, we present a research agenda intended to improve the understanding of OO programming and OO programming education.

1 Introduction

During the last ten years, programming education has experienced a shift from imperative and procedural programming to object-oriented (OO) programming. This shift has been motivated by educators' desire to please information technology industry on one hand and potential students on the other hand. Object-orientation and Java have been spreading as the most important implementation platform for new, web-based applications with wide-spread visibility among computer users, which has created the illusion that programming equals to object-oriented Java programming. Thus, students want to learn Java from the very beginning of their programming studies, and teachers may fear that if an institute is not offering Java as the first programming language, students will go elsewhere. With the current fall in enrollments to academic computing programs [5] educators' thirst for pleasing potential students will probably even increase. Moreover, many companies want to hire students who know how to program in Java and educators may think that if an institute is not teaching Java, its reputation among those companies is gone.

It should be noted that the shift to object-orientation is not motivated by psychology of programming or computer science education research: there are practically no results that would indicate that such a shift is desirable, needed in the first place, or even effective for learning programming [32]. Yet, learning programming should be the most important issue—not learning the peculiarities of a single paradigm or a certain language. Note that “learning programming” does not refer to imperative[1] or procedural—neither functional nor logic—programming, but learning programming in a way that can be applied in many programming paradigms and many programming languages.

Indeed, we are surprised to find out that the cognitive consequences of the shift to object-orientation have been studied neither before the shift nor after it. This lack of research covers both comprehension of programming concepts and development of programming skills. There has been an implicit assumption that classic results on imperative and procedural programming education and learning (see [44] and [60] for reviews) apply to OO programming, also, but we fear that this is not the case. Object-oriented programming is so much more complicated than imperative and procedural programming—both at the concrete notational level and at a more abstract conceptual level—that there are good grounds to question whether the classic results can be generalized to object-orientation.

What this means in practice is that educational institutions around the world are using curricula and teaching methods that are not based on research, but on intuition. There are practically no theories on the development of programming skills or comprehension of programming concepts in the OO case. It is no wonder that educators are fighting against high drop-out rates (e.g., [26]) and poor learning outcomes (e.g., [36]) of programming courses. Research has offered them various pedagogic tricks (e.g., [2, 3, 8, 24, 25, 27, 33–35, 50, 56, 57]), but the lack of solid psychological and educational theories makes a holistic approach impossible.

This paper presents a case for systematic research into comprehension of programming and development of skills in the object-oriented paradigm. In order to understand the huge shift from imperative and procedural programming to object-orientation, we start by comparing these paradigms at three of the five domains that du Boulay [14] presents as issues that a learner must master: *notations* of the particular language, the *notional machine* that describes how programs in the particular language are executed, and the *orientation* describing what programs are for and what can be done with them. Differences between programming paradigms in du Boulay’s two remaining domains, *structures* and *pragmatics*, are not so clear and will not be treated in this paper.

This paper is structured as follows. First, we will look at the differences between imperative and procedural programming versus object-orientation with respect to notations (Section 2), notional machine (Section 3), and orientation (Section 4). Then, in Section 5, we will review research literature and see how it supports our claims. Finally, Section 6 contains a research agenda for OO programming and Section 7 concludes the paper.

[1] Imperative and procedural programming are often considered to be synonyms, but in this paper “imperative” refers to programming with variables, assignment and simple imperative control structures such as sequence, iteration and conditionals whereas “procedural” covers procedures, parameters and recursion, also.

2 The Notational Revolution

Notations needed in Java programs do differ remarkably from those of imperative and procedural programming[2]. This is partially due to the larger number of programming concepts needed, but also due to the structure of the Java language [42].

For example, consider the algorithm for simple user interaction in Figure 1, given in a natural language, English. The pseudo code version of this algorithm is given in Figure 2, and a Pascal program for the same task in Figure 3 (from a popular textbook of its time [7, p. 15]). Even though the notations differ in their level of formality, they look strikingly similar. When we compare the natural language version (that should be in a notation familiar to students) in Figure 1 to the Pascal version (that the students should learn to understand) the new notations and the related concepts are:

– "program", name of the program: program
– interaction ports needed: input/output
– "integer" and the variable name: variables
– "write", "writeln" and "readln": input/output
– "var", "begin", "end" and punctuation: language syntax

The first two of these are required by the language, but are simple to students (this *is* a program with input and output); the next two are just what the students are learning (the concepts of variable and input/output); the last one is something cryptic required by the language. Parts required by the language vary from one language to another; e.g., in Python there would be no special punctuation or statement brackets and the program line would not be needed.

Now, let us turn to the Java version of the same program given in Figure 4, which must be stored in a file with a certain name, Interactive.java. (We assume the existence of another class for user input stored in the file Input.java). Compared with Figure 1 the new notations and the related concepts are:

– "public": visibility
– "class", name of the class: classes and objects
– "static": access rights
– "void": return values
– "main": program
– method name and its argument: methods and their arguments
– "String", "[]", "System", "Input": predefined classes
– "int" and the variable name: variables
– "println", "readInt": input/output
– punctuation: language syntax

This list is much longer than the corresponding list for Pascal and, what is more important, it contains a large number of difficult concepts that are not required by the

[2] We are here interested in differences that are inherent to object-orientation and the way object-related concepts are implemented in Java. We do not treat Java problems that occur within imperative parts of Java, e.g., that using "=" as the assignment operator makes some students to confuse assignment with mathematical equality.

Tell the user that this is an interactive program.
Ask the user to enter an integer value.
Get the number from the user.
Tell the user what the entered number was.

Fig. 1. An example program in English.

```
write 'This program interacts with its user.'
write 'Please enter an integer value.'
read Number
write 'The number you entered was:'
write Number
```

Fig. 2. The example program in pseudo code.

```
program Interactive (input, output);
    var Number: integer;
begin
    writeln ('This program interacts with its user.');
    writeln ('Please enter an integer value.');
    readln (Number);
    write ('The number you entered was:');
    writeln (Number)
end.
```

Fig. 3. The example program in Pascal.

```
public class Interactive {
  public static void main(String[] args) {
    int Number;
    System.out.println("This program interacts with its user.");
    System.out.println("Please enter an integer value.");
    Number = Input.readInt();
    System.out.print("The number you entered was:");
    System.out.println(Number);
  }
}
```

Fig. 4. The example program in Java.

solution of the problem, but by the structure of the language: classes and objects, visibility, access rights, method definitions and calls, and return values.

One may argue that this example program favors imperative programming and that the first programs used in object-oriented courses do not contain this much input and output. Even if that were the case, the first Java program will contain almost all of the above concepts.

Thus, the shift to object-orientation and Java has made a revolution at the notational level even though this might not be obvious at first sight: the lengths of the programs in Figures 3 and 4 are practically the same; yet the number of new notations and concepts is remarkably higher in the Java case. This rise is not due to the programming problems that are solved, but due to the requirements of the language used.

3 The Notional Machine Revolution

In order to be able to understand what individual constructs of a programming language mean and how programs written in that language work, a student must understand how the notional machine [15] underlying that language works. Programs cannot be understood as strings of characters only, but students must understand, e.g., what a variable is and how it is affected by assignments. A more thorough understanding of programming includes, e.g., knowledge of typical uses of variables and control structures [13], which also relies on proper understanding of the notional machine. The machine needed for understanding the first programs should be simple as otherwise learning programming becomes hard [15].

In the procedural approach, instruction typically starts with the imperative constructs: variables, input/output, conditionals and looping constructs. The notional machine needed to explain these notions consists of:

– variable: location or slot with a name and contents
– input/output: two devices connecting variables to external world
– program execution: a program counter referring to a certain point at the program

A notional machine that consists of the above parts is clearly capable of executing the program in Figure 3 and can be used in teaching the first steps in imperative programming.

An extension to this notional machine is needed when pointers are included:

– pointer: contents of a variable may be the location of another variable

Further extensions are needed when procedures are introduced:

– procedure call: a call stack
– parameter: room for parameters in the call stack and parameter passing mechanisms
– return value: mechanism for return value, possibly with room for it in the call stack

It should be noted that these extensions are fully compatible with the initial notional machine and they can be introduced gradually along the introduction of new programming language constructs.

In contrast to the procedural approach, object-orientation requires a much larger and more complicated notional machine from the very beginning. A notional machine that is capable of executing the program in Figure 4 must contain all of the following parts (see the list of concepts of the program given in the previous section):

– object: a heap for objects
– method: a call stack
– parameter: room for parameters in the call stack and parameter passing mechanisms
– return value: mechanism for return value, possibly with room for it in the call stack
– variable: location or slot with a name and contents (in the call stack)
– input/output: two devices connecting methods to external world
– object reference: contents of a variable or a parameter may be the location of an object in the heap
– program execution: a program counter referring to a certain point at the program

Moreover, there are concepts that are needed even though they are not directly expressed in the notional machine: visibility and access rights concern validity of the program, and the relationship between classes and objects concerns the relationship between the program text and the object heap.

Compared with the notional machine in the procedural case, the difference is huge. The OO notional machine described above and needed for the simple program in Figure 4 is not only larger than the corresponding notional machine needed for the equivalent program in Figure 3, but it is much larger than the total notional machine in the procedural case. Furthermore, the notional machine described above does not even contain parts needed to describe other OO constructs that are typically introduced in the first programming course: subclasses and inheritance, implicit calls of superclass constructors, and polymorphism.

One might argue that there is no need for students to understand notations and the notional machine completely—students can simply put aside unnecessary parts as "boilerplate" when first learning. The problem with this thinking is that novices have no means to decide which issues are unnecessary and which must be taken care of when reading or writing programs. The use of "boilerplate" code mystifies programming and obscures concepts that should be learned. Programming should not be taught as a copy-paste art that only incidentally results in a correctly functioning program, but as a clearly defined activity that deals with unambiguous constructs. Otherwise, the central concepts remain blurred.

In summary, the shift to object-orientation and Java has made a revolution at the notional machine level. Not only is the size of the required notional machine much larger than in the procedural case, but the initial notional machine needed in order to understand the first programs is much more complicated, also.

4 The Orientation Revolution

Sajaniemi et al. [46] have studied example programs in elementary programming textbooks among three programming paradigms: procedural, object-oriented, and functional. They found major differences in the programming problem types used in different programming paradigms. The most important issue in procedural programming textbooks is the functionality of programs: example programs compute meaningful values based on input and print the results to users through simple output mechanisms. Object-oriented textbooks deal with data modeling on one hand and demonstrate specific language features on the other hand. Even though message passing structures may be complex, their net effects are trivial from the user's perspective. Finally, functional programming textbooks stress data manipulation techniques. Thus, the orientation, i.e., what programs are for, is very different in these paradigms.

This finding means that also students' tasks are different depending on the programming paradigm used for learning. In procedural programming, students try to write programs that *do* meaningful actions and computations whereas in OO programming students concentrate on creating conceptual models for (usually concrete) data. Détienne [12] notes that when novices design OO programs, the activity of finding classes consumes novices' attention, and they think about functionality only late in the design activity. Ebrahimi and Schweikert [16] found that students have problems in understanding object-orientation and incorporating OO concepts into problem solving. Students tend to spend more time trying to understand objects and less time on problem solving. Thus, the shift to object-orientation has made a revolution at the orientation and students' tasks in programming.

5 Research Support

In the previous sections we have seen that the shift from imperative and procedural programming education to object-orientation has denoted a revolution in the complexity of notations, concepts and the notional machine needed, and in the orientation and tasks carried out by students as programming exercises. In this section, we will look at research literature[3] and see what it says about this revolution.

Imperative and procedural programming: Classic works on programming education and psychology of novice and expert programming (e.g., [4, 10, 11, 20, 29, 39, 41, 43, 52]; see [44] and [60] for excellent reviews) are based on mostly imperative and to some extent also procedural programming—in many cases Pascal programming, which is why we used Pascal in Figure 3. It is evident from this literature that learning programming is hard even in the imperative case. Novices have problems in understanding basic concepts, such as variables and basic imperative control structures [1, 49, 53]—that is, they have problems in understanding the basic notional machine required for imperative programming.

Novices' knowledge about imperative parts of programming languages has been found to be at first fragile [41], such as inert knowledge that students cannot readily master, or misplaced knowledge migrated to inappropriate contexts. As a consequence, students have problems in applying their knowledge even though the knowledge itself may be correct. From a cognitive perspective, the causes of fragile knowledge include a sparse network of associations in long-term memory, i.e., weak connections between different concepts, and underdifferentiation of language commands. Yet, the hardest part of learning is not to learn the syntax and semantics of some language, but to learn how to construct larger program units that are needed to solve the problem at hand (see, e.g., [60]).

A specific source of problems is the limited capacity of working memory. Even when writing simple imperative programs consisting of a few lines only, expert

[3] In this literature review, we look at programming only. Thus, we do not include system design literature even though we do include program design literature.

programmers—let alone novices—cannot form a complete mental representation of the program in their working memory [21]. Highly economical chunking of knowledge is therefore crucial for good performance in programming. As novices' programming knowledge is fragile, efficient chunking is hard for them.

In summary, educational and psychological research into novice imperative and procedural programming tells us that even the simplest imperative notional machine is hard for students to learn, students' knowledge is fragile, and they have serious problems in combining basic constructs of a programming language to form larger, meaningful structures.

OO programming: For object-oriented novice programming, there exists very little psychological and educational research. Most papers (e.g., [2, 3, 8, 24, 25, 27, 33–35, 50, 56, 57]) introduce various pedagogic techniques and tips, such as visualization tools or curriculum changes, without consideration for educational or psychological theories. Only very few articles (see Table 1) analyze object-orientation from a cognitive or educational perspective, i.e., increase our understanding of OO programming learning and how it differs from the imperative and procedural cases. We will next review these results.

Corritore and Wiedenbeck [9] and Wiedenbeck et al. [59] have studied novices and experts comprehending short programs and found that in the OO case the overall function of programs is understood better than details of, e.g., control flow; with procedural programs, comprehenders' knowledge is more balanced. This indicates that programmers' mental representations of procedural and OO programs do differ qualitatively. As the nature of mental representations is strongly related with learning programming, this finding proposes the existence of fundamental differences between learning procedural programming and learning OO programming.

Eckerdal and Thuné [17] have studied novices' understanding of class and object and found several categories of conception of these concepts. Détienne [12], Holland et al. [23] and Teif and Hazzan [54] have found that students have severe misconceptions about fundamental OO concepts, such as classes and inheritance. Fleury [19] has found several misconceptions concerning the construction and use of objects in Java. In procedural programming, misconceptions about parameter passing [18] and recursion [30] have been found; in imperative programming only fragile knowledge instead of misconceptions has been reported. In consequence, problems in learning seem to have different roots in OO programming than in imperative programming.

Mead et al. [37] have compared cognitive problems in learning procedural and OO programming and developed a set of central concepts in the form of "anchor concept graph" for each paradigm. The two graphs differ considerably providing more evidence for the assumption that learning procedural programming and learning OO programming are very different in nature.

Thomas et al. [55] found that students did not perform better in tracing OO code fragments when they were provided with ready-made partial object diagrams, nor did they draw their own diagrams more often in a follow-up test. On the other hand, Lister et al. [31] found that many students were able to track values of numeric variables on paper, and Vainio and Sajaniemi [58] found that students were able to draw values of primitive types, but not object references. Taken together, these results imply

Table 1. Psychological and educational research on OO programming.

Topic of investigation	Expert performance	Novice performance	Cognitive development	Programming education
Mental representations				
Notional machine: structure				
Notional machine: detailed contents				
Notional machine: misconceptions				
OO programs: structure				
OO programs: detailed contents	[9]	[59]		
OO programs: misconceptions				
OO programming: structure				
OO programming: detailed contents		[17]		[37]
OO programming: misconceptions		[12, 19, 23, 54]		
Skills and strategies				
Program comprehension				
Tracing and debugging		[31, 58]		[55]
Program design	[12, 28, 40, 45]	[12]		

that students have more problems in making external representations of OO parts than imperative parts of the notional machine, i.e., the OO notional machine is even more poorly understood by students.

In her state-of-art review of empirical research on object-oriented design, Détienne [12] examined the processes involved in designing in the OO paradigm and in the procedural paradigm. Among other things, she reports on findings of Lee and Pennington [28], Pennington et al. [40] and Rosson and Gold [45] concerning the differences between OO designers and procedural designers. OO designers seem to base their solutions on the problem domain itself, whereas procedural designers use generic programming constructs for structuring their solutions. Thus, the overall approach in program design differs between procedural and OO programming, and their teaching should acknowledge this difference.

Discussion: Even though studies into OO programming are few, the above results make it clear that both OO programming itself and learning OO programming are very different from their imperative and procedural counterparts: mental representation of programs is different, problems have different roots, conceptual contents of knowledge is different, the level of understanding the underlying notional machine is different, and the overall approach to program design is different. These differences are so fundamental to learning processes that we dare to claim that the classic educational and cognitive results of novice imperative and procedural programming should not be used in the OO context.

Furthermore, the number of educational and cognitive studies of learning OO programming is small. Lister et al. [32] studied several popular claims about learning OO programming and found practically no evidence for them in scientific literature. Neither do we know of any results that would provide evidence for the desirability or efficiency of replacing imperative/procedural programming education by object-orientation. On the contrary, Chen et al. [6] found no effects of the first programming paradigm and later design skills; Détienne [12], Pennington et al. [40] and Sharp and Griffyth [51] found positive transfer effects of traditional structured and procedural approaches to OO design.

6 Proposal for Research Agenda

Table 1 collects together research on OO programming described in the previous section. We have tabulated research articles according to two dimensions: the first describing the cognitive content or skill targeted in an investigation; the second telling whether the investigation deals with experts' performance, novices' performance or problems, development of novices' mental representations and skills, or ways to improve this development with educational techniques. The table makes it clear that large areas are totally neglected; even the most researched areas—novices' misconceptions in OO programming knowledge and experts' program design processes—have been both studied in few papers only.

If novices are to be helped in their struggles when learning OO programming, we need to know their problems and misconceptions as well as what experts know and how they apply their knowledge. Only then can we devise efficient teaching methods and contents that have a strong cognitive basis. Many studies in traditional programming have compared expert and novice performance and mental representations, thus providing information on what distinguishes experts from novices. In the OO domain, such studies are rare; all but a single study in Table 1 cover both experts and novices. We therefore suggest that research into *expert and novice differences* should be carried out in all cognitive aspects listed in the table.

A notable gap in Table 1 covers the OO notional machine. There are no studies on experts' or novices' understanding of the notional machine behind OO programming; neither are there studies on teaching a viable notional machine to students. There are some suggestions for visualizing OO program execution (e.g., [22, 38, 47]) but their correspondence to experts' or novices' mental representations or their efficiency in providing a mental model of a correct notional machine has not been studied in detail. In a recent study [48], students were found to be poor in visualizing relationships between objects and method calls during program execution and students' understanding of these relationships, i.e., the structure of the notional OO machine, was found to contain many errors. We therefore suggest that *experts' mental representations of the notional OO machine* should be studied in detail. Moreover, effective *ways to convey this knowledge to novices* should also be investigated.

Another gap in Table 1 is the lack of studies into cognitive development of novices' mental representations and skills. In order to support learning by teaching, steps in cognitive development must first be known. Basic cognitive activities—such as chunking—do, of course, appear in the context of OO programming also, but the building of the notional machine, construction of OO programming knowledge, and detailed development of OO programming skills and strategies presumably have components that are specific to OO programming. We therefore suggest that *novices' cognitive development in OO programming* should be studied.

Investigations of mental representations of OO programs [9, 59] have probed participants' knowledge with yes/no questions divided into categories determined by the researchers *a priori*. Such a method reveals whether participants do possess knowledge in those categories but it does not reveal what other types of knowledge they might have. As a consequence, exact contents of experts' mental representations of OO programs are largely unknown and teachers have only vague ideas of how to best explain important program elements and their relationships to students. We therefore call for *exploratory research into experts' mental representations of OO programs*.

Studies in cognitive processes, i.e., skills and strategies, cover mainly experts' program design. In imperative programming, research into experts' and novices' program comprehension has increased our understanding of the comprehension processes and, moreover, of the mental representations of imperative programs and imperative programming knowledge. The structure of OO programs differs so much from imperative and procedural programs that one may assume that their comprehension processes do also differ considerably. Again, some elements (e.g., hypothesis driven comprehension) are the same, but issues related to program structure can be assumed to differ. We therefore suggest research into *experts' and novices' OO program comprehension processes*.

Finally, results of the research suggested above should be utilized in devising effective methods for teaching OO programming. However, we do not include this work in the research agenda proposal for two reasons. Firstly, the right time for such educational-oriented research will come only after there is a large body of results obtained from the research agenda. Secondly, it may well be that effective ways to transfer experts' mental representations, skills and strategies are at least partially revealed during the earlier research covered by the agenda.

7 Conclusion

The question in the title of this paper was *"What Have We (Not) Done?"* To please industry and students, *we have* shifted from imperative and procedural programming education to object-orientation without studying its necessity or consequences and without studying how OO programming education should be carried out. Moreover, we have used classic results from imperative and procedural programming even though their applicability in the OO case can be questioned. The shift from imperative/procedural programming to object-orientation in elementary programming education is so revolutionary that the use of research results obtained in the imperative and procedural cases is doubtful in the OO case. The amount of notations and concepts needed, the size of the notional machine required, and the whole orientation of programming are so different that the assumptions of imperative and procedural programming research do not necessarily hold for object-orientation.

What *we have not* done is systematic research into the fundamental cognitive and educational issues in learning and teaching OO programming. Lister et al. [32] conclude their paper by noting that "our community needs to discuss—and debate—this issue", but we claim that the computer science education research community and psychology of programming community need to rigorously *study* these issues. For that purpose, we have presented a research agenda comprising:

– experts’ and novices’ differences in mental representations, skills and strategies in OO programming
– novices’ cognitive development in OO programming
– experts’ mental representations of the notional OO machine
– ways to convey the notional OO machine to novices
– exploratory research into experts’ mental representations of OO programs
– experts’ and novices’ OO program comprehension processes

Only with rigorous research into the psychological and educational issues we can attack the problems in learning OO programming.

References

1. Y. Ben-David Kolikant and B. Haberman. Activating ”black boxes” instead of opening ”zippers” - a method of teaching novices. In *Proceedings of the Sixth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE’01)*, pages 41–44. ACM Press, 2001.
2. J. Bennedsen and M. E. Caspersen. Programming in context: a model-first approach to CS1. In *SIGCSE ’04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 477–481, New York, NY, USA, 2004. ACM Press.
3. K. Bierre, P. Ventura, A. Phelps, and C. Egert. Motivating OOP by blowing things up: an exercise in cooperation and competition in an introductory Java programming course. In *SIGCSE ’06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 354–358, New York, NY, USA, 2006. ACM Press.
4. R. E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:534–554, 1983.
5. L. B. Cassel, A. McGettrick, M. Guzdial, and E. Roberts. The current crisis in computing: what are the real issues? In *SIGCSE ’07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 329–330, New York, NY, USA, 2007. ACM Press.
6. T.-Y. Chen, A. Monge, and B. Simon. Relationship of early programming language to novice generated design. In *SIGCSE ’06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 495–499, New York, NY, USA, 2006. ACM Press.
7. D. Cooper and M. Clancy. *Oh! Pascal!* W. W. Norton & Company, New York, 1982.
8. S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. In *SIGCSE ’03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 191–195, New York, NY, USA, 2003. ACM Press.
9. C. Corritore and S. Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50:61–83, 1999.
10. C. L. Corritore and S. Wiedenbeck. What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, 3(2):199–222, 1991.
11. S. P. Davies. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2):237–267, 1993.
12. F. Détienne. 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, 1997.
13. F. Détienne. *Software Design—Cognitive Aspects*. Springer-Verlag, 2002.

14. B. du Boulay. Some difficulties of learning to program. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, pages 283–299. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
15. B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14:237–249, 1981.
16. A. Ebrahimi and C. Schweikert. Empirical study of novice programming with plans and objects. *SIGCSE Bulletin*, 38(4):52–54, 2006.
17. A. Eckerdal and M. Thuné. Novice Java programmers' conceptions of "object" and "class", and variation theory. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education ITiCSE'05*, pages 89–93. ACM, 2005.
18. A. E. Fleury. Parameter passing: The rules the students construct. In *Proc. of the 22nd SIGCSE Technical Symposium on CS Education*, volume 23(1) of *ACM SIGCSE Bulletin*, pages 283–286, 1991.
19. A. E. Fleury. Programming in Java: student-constructed rules. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 197–201, New York, NY, USA, 2000. ACM Press.
20. D. J. Gilmore and T. R. G. Green. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21:31–48, 1984.
21. T. R. G. Green, R. K. E. Bellamy, and J. M. Parker. Parsing and gnisrap: A model of device use. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 132–146. Ablex Publishing Company, 1987.
22. P. Gries and D. Gries. Frames and folders: A teachable memory model for Java. *The Journal of Computing in Small Colleges*, 17(6):182–196, 2002.
23. S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. *SIGCSE Bulletin*, 29:131–134, 1997.
24. M. A. Holliday and D. Luginbuhl. CS1 assessment using memory diagrams. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 200–204, New York, NY, USA, 2004. ACM Press.
25. J. I. Hsia, E. Simpson, D. Smith, and R. Cartwright. Taming Java for the classroom. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 327–331, New York, NY, USA, 2005. ACM Press.
26. P. Kinnunen and L. Malmi. Why students drop out CS1 course? In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, pages 97–108, New York, NY, USA, 2006. ACM Press.
27. M. Kölling and P. Henriksen. Game programming in introductory courses with direct state manipulation. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 59–63, New York, NY, USA, 2005. ACM Press.
28. A. Lee and N. Pennington. The effects of programming on cognitive activities in design. *Int. J. Human-Computer Studies*, 40:577–601, 1994.
29. S. Letovsky. Cognitive processes in program comprehension. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79, NJ: Norwood, 1986. Ablex Publishing Company.
30. D. Levy. Insights and conflicts in discussing recursion: A case study. *Computer Science Education*, 11(4):305–322, 2001.
31. R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4):119–150, 2004.

32. R. Lister, A. Berglund, T. Clear, J. Bergin, K. Garvin-Doxas, B. Hanks, L. Hitchner, A. Luxton-Reilly, K. Sanders, C. Schulte, and J. L. Whalley. Research perspectives on the objects-early debate. *SIGCSE Bulletin*, 38(4):146–165, 2006.
33. R. E. Lopez-Herrejon and M. Schulman. Using interactive technology in a short Java course: an experience report. In *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 203–207, New York, NY, USA, 2004. ACM Press.
34. Q. H. Mahmoud, W. Dobosiewicz, and D. Swayne. Redesigning introductory computer programming with HTML, JavaScript, and Java. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 120–124, New York, NY, USA, 2004. ACM Press.
35. W. Marrero and A. Settle. Testing first: Emphasizing testing in early programming courses. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 4–8, New York, NY, USA, 2005. ACM Press.
36. M. McCracken, T. Wilusz, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. Ben-David Kolikant, C. Laxer, L. Thomas, and I. Utting. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education ITiCSE'01*, pages 125–140. ACM, 2001.
37. J. Mead, S. Gray, J. Hamer, R. James, J. Sorva, C. S. Clair, and L. Thomas. A cognitive approach to identifying measurable milestones for programming skill acquisition. *SIGCSE Bulletin*, 38(4):182–194, 2006.
38. A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004, Gallipoli (Lecce)*, 2004.
39. N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113. Ablex Publishing Company, 1987.
40. N. Pennington, A. Lee, and B. Rehder. Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction*, 10(2&3):171–226, 1995.
41. D. N. Perkins and F. Martin. Fragile knowledge and neglected strategies in novice programmers. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 213–229, NJ: Norwood, 1986. Ablex Publishing Company.
42. A. Radenski. "Python first": a lab-based digital introduction to computer science. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 197–201, New York, NY, USA, 2006. ACM Press.
43. R. S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.
44. A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13:137–172, 2003.
45. M. B. Rosson and E. Gold. Problem-solution mapping in object-oriented design. Technical report, 1989. IBM Research Report RC 14496.
46. J. Sajaniemi, M. Ben-Ari, P. Byckling, P. Gerdt, and Y. Kulikova. Roles of variables in three programming paradigms. *Computer Science Education*, 16(4):261–279, 2006.
47. J. Sajaniemi, P. Byckling, and P. Gerdt. Metaphor-based animation of OO programs. In *Metaphor-Based Animation of OO Programs (Extended Poster abstract). Proceedings SOFTVIS 06 ACM Symposium on Software Visualization*, New York, NY, USA, 2006. ACM Press.
48. J. Sajaniemi, M. Kuittinen, and T. Tikansalo. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. Submitted.

49. R. Samurçay. The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, pages 161–178. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
50. V. Shanmugasundaram, P. Juell, and C. Hill. Knowledge building using visualizations. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 23–27, New York, NY, USA, 2006. ACM Press.
51. H. Sharp and J. Griffyth. The effect of previous software development experience on understanding the object-oriented paradigm. *Journal of Computers in Mathematics and Science Teaching*, 18(3):245–265, 1999.
52. E. Soloway and J. C. Spohrer, editors. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
53. J. C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy Pascal programs. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, pages 355–399. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
54. M. Teif and O. Hazzan. Partonomy and taxonomy in object-oriented thinking: Junior high school students' perceptions of object-oriented basic concepts. *SIGCSE Bulletin*, 38(4):55–60, 2006.
55. L. Thomas, M. Ratcliffe, and B. Thomasson. Scaffolding with object diagrams in first year programming classes: Some unexpected results. In *Proc. of the 35th SIGCSE Technical Symposium on CS Education*, pages 250–254, 2004.
56. N. Truong, P. Bancroft, and P. Roe. Learning to program through the web. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 9–13, New York, NY, USA, 2005. ACM Press.
57. I. Utting. Problems in the initial teaching of programming using Java: the case for replacing J2SE with J2ME. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 193–196, New York, NY, USA, 2006. ACM Press.
58. V. Vainio and J. Sajaniemi. Factors in novice programmers' poor tracing skills. In *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM Press, 2007.
59. S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. L. Corritore. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11:255–282, 1999.
60. L. E. Winslow. Programming pedagogy — a psychological overview. *SIGCSE Bulletin*, 28:17–22, 1996.