A Study of the Development of Students' Visualizations of Program State during an Elementary Object-Oriented Programming Course

JORMA SAJANIEMI, MARJA KUITTINEN and TAINA TIKANSALO
University of Joensuu

Students' understanding of object-oriented (OO) program execution was studied by asking students to draw a picture of a program state at a specific moment. Students were given minimal instructions on what to include in their drawings in order to see what they considered to be central concepts and relationships in program execution. Three drawing tasks were given at different phases of an elementary OO programming course where two animation tools were used for program visualization. The drawings were analyzed for their overall approaches and their detailed contents.

There was a large variability in the overall approaches and the popularity of various approaches changed during the course. The results indicate that students' mental representations of OO concepts and program execution not only grow as new material is covered in teaching, but they also change. The first drawings treat methods as having primarily a static existence; later methods are seen as dynamic invocations that call each other. The role of classes in program execution fluctuates during learning, indicating problems in locating the notion of class with respect to, for example, objects. Two major sources of problems that manifested in many different forms were the relationship between object and method, and the role of the main method with respect to program state. Other problems were caused by overly simplistic understanding of object identification and improper use of application domain knowledge.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*; D.m [**Software**]: Miscellaneous—*Software Psychology*

General Terms: Human Factors, Experimentation

Additional Key Words and Phrases: CS1/2, mental representation, object-oriented programming, program state, visualization

Author's address: J. Sajaniemi, University of Joensuu, Department of Computer Science and Statistics, P.O.Box 111, FI-80101 Joensuu, Finland; email: Jorma.Sajaniemi@Joensuu.fi.
This paper originally appeared in the Proceedings of the 3rd International Workshop on Computing Education Research, 2007, Atlanta, Georgia.

ACM Reference Format:
Sajaniemi, J., Kuittinen, M., and Tikansalo, T. 2008. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. ACM J. Educ. Resour. Comput. 7, 4, Article 3 (January 2008), 31 pages. DOI = 10.1145/1316450.1316453. http://doi.acm.org/10.1145/1316450.1316453.

1. INTRODUCTION

Students learning object-oriented (OO) programming have problems not only in developing program writing, comprehension, and debugging skills but also in understanding basic OO concepts [Ebrahimi and Schweikert 2006; Eckerdal and Thuné 2005; Holland et al. 1997; Lewandowski et al. 2005; Ragonis and Ben-Ari 2005; Robins et al. 2003; Schulte and Bennedsen 2006; Teif and Hazzan 2006; Winslow 1996]. Research into novices' understanding of OO has revealed several misconceptions and problems that concern the notions of class and object—concepts that form the basis for structuring data and actors within a program. Another central aspect of OO programs is their execution: Sequence of method calls and their effects on objects. However, novices' comprehension of these dynamic aspects of methods and method calls has not been studied in detail. Execution of an OO program requires the notion of program state with dynamic creation of objects and nesting of method calls in order to be understood. Du Boulay [1989] refers to such an execution mechanism with the term notional machine and suggests that a learner must master it in order to be able to understand program behavior.

Teachers use pictures to visually display OO concepts and program execution to their students. However, in many novice-level programs the set of objects remains the same throughout the program and is described with a static picture with no reference to method execution. Visualization and animation tools have also been developed to visualize OO concepts but there is little research into the effectiveness of such tools [Gross and Powers 2005], and it seems that the type of engagement, that is, what a student does with the visualization, is far more important than the visualization itself [Hundhausen et al. 2002]. Moreover, students' ability to take advantage of visualizations is weak: Teachers' drawings are poorly understood by students, students are unable to draw their own pictures, and students' abilities to make inferences about program execution based on drawings depicting program state are poor [Thomas et al. 2004; Vainio and Sajaniemi 2007]. These findings cast doubt on teachers' positive attitudes on the effectiveness of visualizations on students' learning of the notional OO machine, and on assumptions that students do easily assimilate the same drawing styles.

In order to study students' own visualizations of OO program state (i.e., the notional OO machine) and their (mis)conceptions of OO concepts, we asked students to draw pictures depicting program state at a specific point of program execution. These drawings were then analyzed by looking at what types of elements they contained and what the relationships were between these elements; correctness of drawings was considered less important unless errors occurred systematically in similar contexts. In order to see how students' visualizations of program state evolved during learning, several drawing tasks were given along an elementary OO programming course.

Students were given minimal instructions on the form or content of the drawings. As a result students were guided by pictures they had seen earlier during lectures and in textbooks, and by their own mental representation of program state and OO concepts—what they considered to be central elements and relationships in programs and program execution. Thus, drawings do not only report the way students wanted to draw things; they also reveal students' understanding of OO programs and the notional OO machine as well as their conceptions and misconceptions of dynamic aspects of OO programs.

The rest of this article is organized as follows. Section 2 gives an overview of related research on visualization and OO misconception analysis. Section 3 describes the study; its results are presented and discussed in Section 4. Finally, Section 5 concludes the article.

2. RELATED RESEARCH

In this section, we will give an overview of literature on students' visualizations of program state, other uses of students' visualizations in programming, and students' misconceptions about OO concepts.

2.1 Visualization of Program State

Ford [1993] asked students that had been taught OO design and programming for one semester to design and implement animations that exemplified features of the C++ programming language. Students worked in teams of two to four persons. They used special animation software to implement the animations with the view that animations were to be recorded on video and used the following year to demonstrate language features to new students. Ford used exploratory data collection followed up by interviews with the students. His findings concern students' ideas of programming concepts as well as specific visualization techniques used because of system limitations. An example of the latter category is the visualization of a large array with only some of its elements to save screen space. More important for our study are findings in the first category, for example, the use of a box with a slightly open lid for a variable to show that the variable can receive a new value or pass a copy of its value any time.

Ford focused on dynamics involving variables, classes, choice statements, iteration, and functions, and noted that these show how students visualized the notional machine behind the language. Students visualized variables typically with boxes or, in cases where the type of variable was important, with a different shape or color that identified the type. Variable identifiers were most often located outside the box but some students placed the identifier inside the box, separated with a bar from the value. Choice statements and iteration were animated with various forms of flowcharts and different colors; in interviews some students explained how they had selected the form of the flowchart so that it corresponds to the "meaning" of the structure, for example, that the else-part is the default and the then-part is just a deviation from the main path. Nearly all students used rectangles for functions and coloring for function calls with parameters floating in and return values floating out of the rectangle. Ford did not report whether a new function rectangle was created for each function call but it seems that a single rectangle for each function was typically present through the whole animation.

Classes were usually depicted in the same way as in lectures but some students used various techniques to show that private member variables (i.e., attributes) are not visible outside the class and can be accessed by public methods only. Method calls were animated by arrows pointing to, or color changes to, method names. When methods were executed, animations used arrows and color to show which member variables were being accessed.

In Ford's study, students used special animation software to make a dynamic representation of program execution. Students' visualizations were analyzed in an exploratory way and only part of the findings were reported with the goal of illustrating the variety of student visualizations. Notably, visualization of OO concepts was treated neither systematically nor extensively. This study is closest to our study, but we use pen and paper instead of laborious animation, our students give a static visualization of program state instead of a dynamic animation, our students work alone and not in groups, we gather data at several points during a programming course, we look at OO constructs only, and we analyze students' visualizations systematically and in detail.

Teif and Hazzan [2006] studied junior high school students' perceptions of class and object using questionnaires, interviews, videotapes, and a researcher diary. The questionnaires included drawing tasks where students were asked to draw the class-object relation; later, students were interviewed and could explain their drawings. Example drawings in Teif and Hazzan [2006] depict concrete things such as the face of a cat with ears, eyes, mouth, etc. Students' drawings and explanations reveal several misconceptions (set-subset and class-object confusion, set and object confusion, part-whole and object-class relations confusion, part and attribute confusion). Teif and Hazzan categorized these misconceptions as confusions in taxonomic or partonomic hierarchies, where taxonomic refers to kind-of relation and partonomic to part-of relation. In this study, students' drawings do not depict a specific program or program state but are pictures of everyday things intended to visualize class-object relation at a more general level; students' drawings are analyzed for OO misconceptions.

Lister et al. [2004] conducted a multinational, multiinstitutional study on novices' tracing skills by giving students program fragments and asking them to predict the outcomes of those fragments. Some of the fragments were non-idiomatic, that is, their outcomes were not what one would expect by a quick look, but rather step-by-step tracing was needed. Students answered multiple-choice questions and think-aloud protocols were collected from some participants. Students were allowed to draw pictures or perform calculations while answering; such doodles were collected and analyzed to see if certain types of doodles appeared more often with correct answers. Doodles were grouped into 11 categories such as underlined (i.e., some part of the question was underlined or shaded for emphasis by the student), computation (e.g., "3 + 5"), trace (consecutive values of a variable), synchronized trace (trace of several variables so that all are rewritten whenever any of them changes), etc. Even though the program fragments were written in Java, they were imperative by nature, for example, no objects other than arrays were used. Consequently, the doodle categories do not cover OO concepts. Lister et al. [2004] found that doodles indicating close tracking of code appeared more often with correct answers. They concluded that while the results will surprise few teachers, they are very useful for teachers to quote to their students. In this study, students' drawings depict various aspects of imperative program execution, but no OO concepts; students' drawings are analyzed to see if certain element types occur with good tracing performance.

Holliday and Luginbuhl [2004] developed a diagram type, a so-called memory diagram, to describe objects and method invocations, used it in programming education, and studied students' use of the diagram. A memory diagram describes the state of program execution at a certain point of time. Variables are represented as rectangles, objects as circles, classes as diamonds, object references as arrows, and method calls as wavy arrows labeled with parameters. Method call arrows start at the variable that contains reference to the called object and end at the method name. A double arrow starting at the method name points at the return value. Holliday and Luginbuhl used memory diagrams extensively in lectures. In class interaction, they asked students to draw the memory diagrams associated with their solution to an exercise, and used these diagrams to identify which OO concepts students have problems with within that specific exercise. Holliday and Luginbuhl also conducted a study where students were asked to draw simple memory diagrams in a normal examination. They found a positive correlation between students' ability to construct memory diagrams with their performance on the rest of that exam and on the course overall. Of course, correlation does not imply causality and it may simply be that students who are good at OO programming are also able to draw diagrams. In this study, diagrams depict program state with the help of objects and method invocations. The form of diagrams is fixed by the teachers; students' diagrams are used in a classroom setting to find OO misconceptions so that teachers can give appropriate feedback to students.

Thomas et al. [2004] studied first-year programming students' understanding of program behavior and especially of object referencing. They used object diagrams in many examples when tracing code during lectures. They then conducted a test consisting of multiple-choice questions on program tracing. In the test, half of the students were given ready-made partial object diagrams. Several weeks later a new test was conducted with no ready-made diagrams, and students' scratch sheets were collected and analyzed for their use of object diagrams. Thomas et al. found that students did not perform better in tracing OO code fragments when they were provided with the ready-made diagrams, nor did they draw their own diagrams more often in the follow-up test. Thomas et al. concluded that the inability to use diagrams in program tracing is a widespread phenomenon among beginning, weaker students. In this study, diagrams depict program state by describing existing objects at a certain point of program execution. Execution history is visible through past object references represented by crossed arrows. The form of diagrams is fixed by the teachers; students' diagrams are not analyzed in detail.

In the previous studies, researchers collected drawings made by students and analyzed their contents or counted their frequencies. Another approach to program state visualization is to build tools that visually display programs to students. The assessment of the impact of such tools is rare and based on course grades or drop-out rates, attitude surveys, and tool usage statistics or observation rather than analysis of students' drawings of program state [Gross and Powers 2005]. There are also tools where the purpose of student-written programs is to produce animations of, for example, people and animals and where program state and visualization are practically unified [Kelleher and Pausch 2005]. These tools are based on icons for programming constructs, direct manipulation, and textual programming and they are usually meant to be used by children.

2.2 Other Uses of Visualizations

Visualization and animation have also been used to make algorithms more understandable to students. In algorithm visualization, program constructs are only partially represented at the notional machine level; data structures and high-level operations are more important and hence visualizations concentrate on them. Researchers have also asked students to produce visualizations of algorithms (e.g., [Douglas et al. 1995; Hübscher-Younger and Narayanan 2003]) in the hope of finding visualizations that match better students' mental representations. Because the level of these visualizations is not the level of the notional machine, and because such algorithms rarely contain OO concepts, we shall not consider them further in this article.

A very different approach to visualization and programming has been taken by Tolhurst et al. [2006], who conducted a multinational, multiinstitutional study to see whether novices' map-drawing styles predict success in elementary programming courses. The rationale behind this study is the finding that program code can be characterized as a virtual space where programmers must navigate and that different navigation styles result in different views of a program; thus programs can be understood as maps and navigation strategies may be reflected in map drawing strategies and therefore in programming performance. Students were first asked to draw a map of a given physical area to be used for getting from one location to another, then they were asked to annotate the map with key decision points, explaining what a person using the map would need to do at those points. These maps were classified into three categories depending on their overall approach: (1) "landmark" maps whose focus is on key visual landmarks along the route, (2) "route" maps whose focus is on the actual route taken, and (3) "survey" maps that give an overview of the area where the route is located. Tolhurst et al. [2006] found a general trend for students who drew a survey map to be more successful in introductory programming courses than route map students, who performed better than students drawing landmark maps. There was an interaction effect between students' country and mapping style that could not be explained. In this study, students' drawings are physical maps that have no relation to programming; students' drawings are analyzed for their overall approach in order to see if it correlates with programming course performance.

2.3 Misconceptions about OO Concepts

Novices' misconceptions about OO concepts have been studied also without drawings. Holland et al. [1997] registered students' problems while teaching two OO courses and identified several misconceptions: object equals to variable; objects are simple records; methods' work is done by assignments only; object equals to class; object identity confused with attribute value; and textual representation of object is its reference.

Fleury [2000] studied novices predicting outcomes of short Java programs. She identified four student-constructed rules, e.g., the only purpose of invoking a constructor is to initialize the instance variables of an object. These rules do not lead to errors, but limit the way Java constructs can be used.

Or-Bach and Lavy [2004] studied students' answers to a design task that required the use of inheritance and polymorphism and found three categories that describe how students perceive abstract classes: (1) a collection of attributes; (2) a collection of attributes and implemented methods; or (3) a collection of attributes, implemented methods, and abstract methods. Eckerdal and Thuné [2005] studied students' conceptions about object and class and found several categories that describe how students perceive these concepts. For example, an object can be experienced as a piece of code; as a piece of code that is active in the program; or, in addition, as a model of some world phenomenon.

Garner et al. [2005] recorded students' problems during an introductory Java course using a predefined list of problem types. Even though some of the problems may be the result of misconceptions, the data collecting method used in the study does not support analysis of causes for problems. Thomasson et al. [2006] studied students' answers to an OO design problem and found several problems with students' design skills. These problems are, however, not misconceptions of OO concepts but represent students' incapability to use OO concepts in a design task. Sanders and Thomas [2007] gave an overview of the previous six studies and reported on a study in which they identified some of the above-listed misconceptions in novices' programs.

Ragonis and Ben-Ari [2005] followed 10th-grade students on an elementary OO programming course and found several misconceptions (object state is unchangeable, methods can be called only once, parameters must be given manually, return values are shown on the screen, no need for input statement, objects are created manually, lack of understanding of the sequence of method calls) and attributed many of these to the use of static visualization and interactive features of the used programming IDE (BlueJ [Kölling 2007]). They suggest that dynamic visualization is needed so that students can coordinate the static and dynamic aspects of OO programs.

3. THE STUDY

In order to study the development of students' visualizations of OO program state, we gave students program visualization tasks three times during an elementary programming course. Students were given a short program and were asked to draw a picture depicting the state of the program at a given point of execution. Drawing instructions were minimal in order to encourage students to draw what they found most important. During the course, the lecturer used standard visualization techniques, e.g., rectangles for objects, and in lab exercises students used two program visualization tools giving them examples of what a visualization might contain and look like.

3.1 Method

The study was conducted in lab exercises of an elementary Java course that lasted ten weeks with four hours of lectures, two hours of exercises in normal classrooms, and two hours of lab exercises each week. Due to the Christmas holiday, there was a two-week break after the sixth week of the course. For programming assignments, students used both BlueJ [Kölling 2007] and command line interface. In the lab exercises, two program animation tools—Jeliot [Moreno et al. 2004] and a metaphor-based tool [Sajaniemi et al. 2006]—were used to animate two Java programs; first with one tool and later in the course with the other tool. In two of the four lab groups Jeliot was used first, whereas the other two lab groups started with metaphor animation. Due to external conditions, two teachers were involved; each Java program was, however, always animated by the same teacher.

The three drawing tasks were made during lab exercises so that the first task was given a week after the use of the first tool, the second task a week after the second tool, and the third task two weeks after the second task. The drawing tasks were part of a longer questionnaire that was presented in various forms four times during the course. No drawing task was given prior to the first visualization tool use because students' knowledge of OO concepts was too weak at that time. Table I contains a summary of the course schedule and lists main topics covered by lectures each week.

The Jeliot program animation tool [Moreno et al. 2004] visualizes objects as rectangles containing names and values of attributes (or member variables) in smaller rectangles. Method calls are visualized with similar rectangles containing local variables; nested method calls are visualized with a stack where method rectangles are piled. Object references are visualized with tightly aligned arrows. The overall appearance of Jeliot is technical: A collection of rectangles and arrows.

The metaphor animation tool [Sajaniemi et al. 2006] visualizes objects as watch panels with monitors for attributes. Method calls are visualized with workshops attached to the watch panel depicting the appropriate object; parameter passing is depicted with a flying envelope containing parameters; and method nesting is depicted with arrows that appear when the envelope is passed. Object references are depicted with pennant pairs. Other metaphoric elements include workbenches for return values, blueprint books for classes,

Table I. Course Summary

Week	Study activities	Topics in lectures
1		Notion of algorithm, overview of OO concepts
2		Structure of Java programs, basic data types, variables and their roles
3	First questionnaire; Animation of program A (first tool, teacher X)	Expressions, arrays
4	Animation of program B (first tool, teacher Y)	Control structures
5	Second questionnaire (first drawing task)	Attributes, methods, recursion
6	Animation of program B (second tool, teacher Y)	Class libraries, strings
7	Animation of program A (second tool, teacher X)	Exceptions, Vector
8	Third questionnaire (second drawing task)	Enumeration, I/O
9		Testing, documentation
10	Fourth questionnaire (third drawing task)	GUI, event-driven programming

a garbage vehicle for garbage collection, and role metaphors [Sajaniemi and Kuittinen 2004] for attributes and variables. The overall appearance of metaphor animation is figurative: pictures of workshops, envelopes, etc.

Both program animation tools contain visual elements that are not obvious to novices. Students were given a one-page description of both visualizations but, due to time constraints, the visualizations were not explained in detail and their terminology was not used in lectures. Each animation session lasted about 30 minutes. First, the teacher presented the Java program to be animated and then animated it and explained its behavior. Then, students animated the same program with different input values and, if time allowed, with their own input values. When a tool was used for the first time, the teacher explained the most important visual elements of the tool.

3.2 Participants

The course was started by 59 students, 13 of them female. Those who had no previous programming experience had attended a short introduction to imperative programming with Python. The four questionnaires were a part of course duties and were answered by 48, 41, 29, and 30 students, respectively. The number of students attending all three drawing tasks was 22; 12 attended two tasks and 10 students attended a single drawing task. Students were allowed to freely choose among the four lab session groups.

3.3 Materials

The questionnaires were copied so that each question was on its own page. The first page contained a self-assessment of understanding of nine programming concepts that were partially varied between questionnaires. The second page asked the student to describe in his or her own words the concept of object and encouraged to draw a picture; the third page did the same for the concept of class. This was followed by the drawing task. In the last questionnaire, the class description task was dropped and the drawing task was followed by an evaluation of the two visualization tools used. The questionnaires were anonymous so answers of the same student could not be tracked. In this article we analyze the results of the drawing tasks only.

For the three drawing tasks, two short Java programs were prepared. The second program, given in Figure 1, was used in the second and third drawing tasks. The first program was simpler consisting of a single class and having no local variables in methods. In both programs, a method call in the "main" method and a point of execution within a (possibly nestedly called) method were marked (see Figure 1).

3.4 Procedure

Participants were allowed a certain amount of time for each task within a session: 1:30 minutes to self-assessment, 3 minutes to each concept definition task, and 15 minutes to the drawing tasks. The time was limited because we were not so much interested in the completeness or correctness of answers, but in the contents of the answers: what issues students considered most important, that is, what were the central contents in their mental representations of the concepts and program state.

Pennington [1987] has probed expert programmers' mental representations of specific programs by asking them to write a program summary and then analyzed the contents of those summaries. Later, the same technique has been used by several researchers (see, e.g., [Good and Brna 2004]). In order to not direct people producing program summaries, it is customary to use nonspecific instructions, for example, "write a summary of the program" [Good and Brna 2004]. The wording of the drawing tasks in our study was (translated from Finnish): "Draw a picture that includes existing objects and methods and their relationships when the program . . . ". An unspecific instruction would ask to "draw a picture of program state at . . . ". Our participants were, however, novices and we expected that the concept of program state was not familiar to them; therefore we used a more specific wording that mentioned objects, methods and their relationships. Moreover, only a few participants limited themselves to objects and methods only; most of them included attributes, local variables, parameters, classes, etc. in their drawings. Thus, we may assume that the drawings cover issues considered to be salient to program state by the participants.

3.5 Analysis Method

Because we were interested in the contents of mental representations, we made an *element analysis* based on a scoring form where the existence of various elements could be coded. An element was coded as existent even if all of its occurrences in the program did not appear in a drawing. For example, the element "object" was coded to exist even if a single object out of several objects of a program appeared in a drawing. This decision was

```
public class Town {
  public static void main (String[] args) {

    Owner owner_a = new Owner("John",50);
    Owner owner_b = new Owner("Pete",20);

    Pet dog1 = new Pet("Cody", owner_a);
    Pet dog2 = new Pet("Duke", owner_a);
    Pet dog3 = new Pet("Toby", owner_b);

    owner_b.increaseFood(100);
    dog1.hunger(5);  // --- During this method call
    dog3.hunger(4);
  }
}

// ----------------------------------------------

class Pet {

  private String name;
  private Owner owner;

  public Pet (String n, Owner ow) {
    name = n; owner = ow;
  }

  public void hunger (int wish) {
    int given;
    given = owner.feed(wish);
    System.out.println(name + ": " + given);
  }
}

// ----------------------------------------------

class Owner {

  private String foreName;
  private int food;

  public Owner (String n, int f) {
    foreName = n; food = f;
  }

  public int feed (int wished) {
    int portion = wished;
    if (wished > food) portion = food;
    food = food - portion;
    // -- at this point
    return(portion);
  }

  public void increaseFood (int increment) {
    food = food + increment;
  }

}
```

Fig. 1. The Java program used in the second and third drawing tasks (translated from Finnish).

justified for two reasons: (1) because of the time limit, the student might not have had enough time to draw all objects and, (2) on the other hand, the student perhaps did consider that a single object represented the other objects in the picture. In any case, when drawing an object, the student did consider that objects are important in representing program state, that is, objects were part of the student's mental representation of the notional OO machine.

The first version of the analysis form was created by one researcher and it was independently applied to a small number of drawings by three researchers. The three scorings were compared and the analysis form was revised. The revised form was used by two researchers independently for all drawings of a drawing task. The researchers then discussed differences and agreed on the final scoring. This was repeated for all three drawing tasks.

The analysis looks at three aspects of a drawing: (1) impact of previously seen visualizations, (2) form of the drawing, and (3) the contents of the drawing. In the *impact of visualizations* part, we scored the possible impact of Jeliot, metaphor animation, UML, or some other known visualization (see Table III; the results included in the tables will be treated in more detail in the next section). A score of 1 was used if any impact was found even in a small part of a drawing.

The *form* of a drawing was coded by the type of visual elements it contained (Table IV): text, rectangles, etc. The score for each visual element type was the number of different subtypes, e.g., the number of visually different rectangle types or the number of different special characters in the drawing.

The *contents* of a drawing indicated the existence of OO-related concepts (Table V) like classes, methods, etc. Existence was scored as incorrect or correct depending on whether the concept was mostly used incorrectly or correctly. The scoring makes a distinction whether a visual element represents static or dynamic existence: for example, a rectangle representing some method is coded as C16 if it represents the existence of the method in the program code (static existence), and as C17 or C18 if the rectangle represents an execution of that method—C17 for an ongoing method execution and C18 for a past method execution.

Note that the contents scoring tells only whether some concept is being used—not how many elements of that type a drawing contains. For example, Figure 2 contains a drawing made by a student in the second task. This drawing was scored as containing the following correct content element types:

—C2 (data type): the word "int"
—C5 (class): for example, "class Owner"
—C6 (object): object rectangles
—C9 (object reference, id of the referring element visible): for example, "owner owner_a" where "owner" is the id of the referring element
—C12 (dynamic existence of an attribute, id visible): as C9
—C15 (attribute's belonging to its object): for example, "name" inside the dog1 object rectangle
—C16 (static existence of a method): all methods are listed under the corresponding class and method call arrows point to these methods; the listing thus represents existence of the methods within the program
—C17 (dynamic existence of an existing method): arrows pointing to the methods "feed" and "hunger"

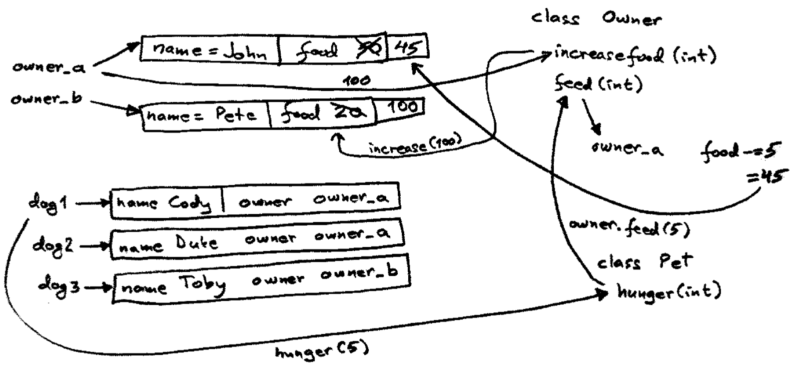

Fig. 2. A student's drawing of the program state in the second drawing task. (Translated from Finnish and redrawn.)

—C18 (dynamic existence of an exited method): arrow pointing to the method "increaseFood"

—C21 (method's or constructor's belonging to its class): for example, method "increaseFood" drawn as subordinate to the class "Owner"

—C22 (method's or constructor's belonging to its object): arrow from method "increaseFood" to object "owner_b"

—C24 (dynamic call graph of methods): arrow from method "hunger" to method "feed"

—C25 (static existence of a parameter): for example, "int" in "hunger(int)"

—C27 (dynamic existence of a parameter, id not visible): for example, "5" on the arrow pointing to the method "hunger"

—C28 (parameter's belonging to the called method or constructor): for example, "hunger(5)"

—C35 (existence of "main" method's local variable, id visible): for example, "owner_a" on the upper left corner

—C45 (expression or condition with specific values): "food-=5"

—C46 (consecutive values of an attribute, variable, or parameter): for example, crossed "50" followed by "45"

—C47 (other): chain of arrows from the method "feed" to the attribute value "45" describing which object's attribute the method changes

The drawing contains several errors: the name of the first attribute in "Owner" objects should be "firstName" (instead of "name"); the updated value of the attribute "owner_b.food" should be 120 (instead of 100); the arrow labeled "100" should not start from "owner_a" (but from the "main" method not depicted in the drawing); and the arrow labeled "hunger(5)" should not start from "dog1" (but from the "main" method, also). All content element types were, however, scored as correct because the drawing contains mainly correct attribute names, calculation errors are not interesting when we are looking at students' understanding of OO concepts, etc.

Table II. Overall Drawing Approaches and Their Frequencies (%)

Drawing Task					
1st	2nd	3rd	Code	Central Concepts	Subordinate Concepts
15	22	7	A1	Existing objects	
20	44	55	A2	Existing objects	Active methods
44	19	10	A3	Existing objects, all methods	
5	0	0	A4	All methods	Effects on objects
0	0	3	A5	Classes	Active program code, active objects
2	0	3	A6	Active object	Important program code fragments
7	7	3	A7	Path of execution	Objects, methods
0	4	10	A8	Path of active execution	Objects, methods

In the scoring scheme, three of the content element types are inherently erroneous. Element C29 means that a parameter is depicted as belonging to the object whose method has been called. However, parameters do not belong to objects—they belong to methods. Element C34 means that the owner of a local variable is understood in the same wrong way. Finally, C38 is used when an object reference that belongs to the "main" method is depicted as a property of the object that it refers to. With C29, C34, and C38, correct use was coded if the drawing agreed with the program even though ownership was represented incorrectly.

When the element analysis was done, it was obvious that the overall approach of drawings was not captured. For example, two drawings containing objects and methods could be very different if one had objects containing methods and the other consisted of methods with code fragments referring to objects. In the element analysis, they could get similar scores even though the central concepts and the overall approaches were very different. Therefore, we made also an *approach analysis*. This was made by two researchers together. They looked at all drawings and grouped them according to their overall approach. The recognized approaches (Table II) are described in the next section.

4. RESULTS AND DISCUSSION

There were 41 students participating in the first drawing task, 29 in the second, and 30 in the third task. They returned 0, 2, and 1 empty drawings, respectively. Moreover, one drawing in the first task was not included in the analysis because it contained few and unrelated words only. When a drawing was accompanied with a textual explanation, only the drawing part was analyzed.

4.1 Approach Analysis

Overall, students' drawings consisted of components (objects, methods, etc.) connected with arrows and lines that described relationships between the components. However, there was a significant diversity in what components were central and what were subordinate to the central components. Table II lists eight approaches identified in the drawings, and their frequencies in the three drawing tasks. The frequencies do not add up to 100% because a few drawings in each task could not be categorized due to students' serious

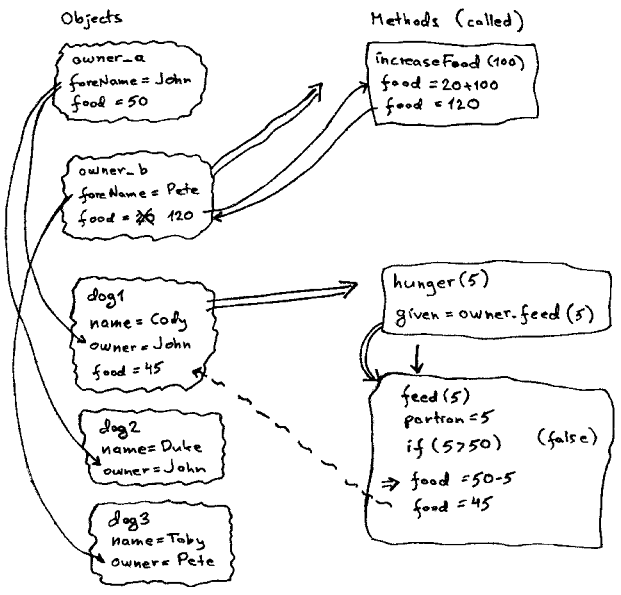

Fig. 3. A drawing describing existing objects and called methods (A2). The drawing contains content element types C1, C6, C9 (wrong direction; object identified by its attribute), C12, C15, C17, C18, C22 (wrong direction), C24, C27, C28, C31, C33, C35, C38, C45, C46, and C47 (attribute duplicated in method; final value transferred to wrong object). (Translated from Finnish and redrawn.)

misunderstanding of basic OO concepts. (Note that for the sake of avoiding undue accuracy, frequencies are given in all tables as integers even though they are percentage figures.)

The approach used in both animation tools is A2, existing objects as the central concept and active methods as subordinate to them. Pictures based on this approach are also frequent in OO programming textbooks. Figure 3 presents a typical example of this approach; Figure 4 gives another way of describing basically the same information. Even though both animation tools used this approach, only 20% of students used it in the first drawing task, and even though its frequency grew during the course, only 55% used it in the last task. In this approach, objects were typically depicted as rectangles, and active methods were located as separate boxes or inside objects, or attached to objects in the style of the metaphor animation tool. In no drawing were methods stacked in the Jeliot style.

In the second most common approach, A3, objects and methods are considered to be equally central concepts. Objects are typically depicted at other side of the drawing and all methods (whether they are called or not) on the other side (see Figure 2). Arrows represent method call nesting and several arrows may enter a method if it has been called several times. This approach was most common in the first task and its popularity decreased during the course. Taken together, approaches A2 and A3 covered 63–65% of drawings in each task.

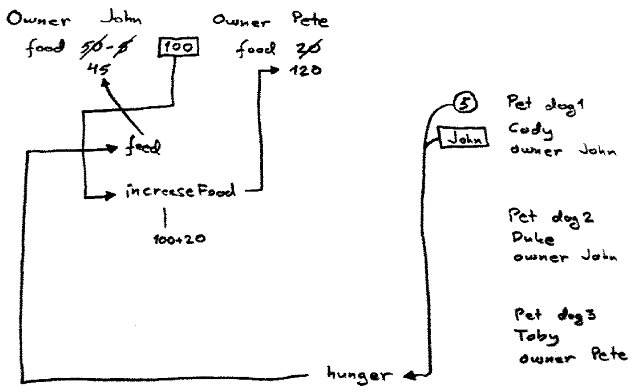

Fig. 4. A drawing describing existing objects and called methods (A2). The drawing contains content element types C6, C7, C9, C12, C13, C15, C17, C18, C22 (wrong direction), C24, C27, C28, C29, C35, C38, C45, C46, and C47 (transfer of attribute through method). (Translated from Finnish and redrawn.)

In the first two tasks, approach A1 was quite popular. In this approach, only objects and possibly their attributes are depicted, but methods are omitted. Note that the task instructions explicitly asked students to "draw a picture that includes existing objects and methods and their relationships." One might be tempted to assume that students just did not have enough time or ability to draw methods, which would make approach A1 to be a preliminary version of approaches A2 and A3. However, the decrease of A1 does not increase the popularity of A2 and A3 whose joint frequency remains stable. Thus, the reason for using A1 may not be simply a desire to use A2 or A3, but to fall on A1 because of an inability to visualize methods. It may, for example, be that students who used A1 in the first tasks were unable to continue the course and quit, or that their mental representations of program state did not handle methods in the usual way and they used uncommon approaches (A4–A8) later during the course.

In approach A4, all methods form the starting point of the drawing. Method calls are represented by arrows from one method to another and several arrows may enter a method just as in A3. In contrast to A3, objects are not depicted at all. Instead, methods contain their effects on attributes of various objects, that is, path of execution during calls of the method. This approach was used in the first task only; it may be affected by previous experience with procedural programming in which program execution is structured by function calls. Our data does not allow checking this hypothesis.

Classes make up the structure of drawings in approach A5 (see Figure 5). Currently active program code within the nested method calls is depicted with no reference to the associated objects. Objects are mentioned as a part of the active program code only. In this approach, objects are not considered to be important in describing program state—classes and program code are. As the task instructions asked to depict objects and methods, it seems that students using this approach may not see clearly the distinction between classes and objects.

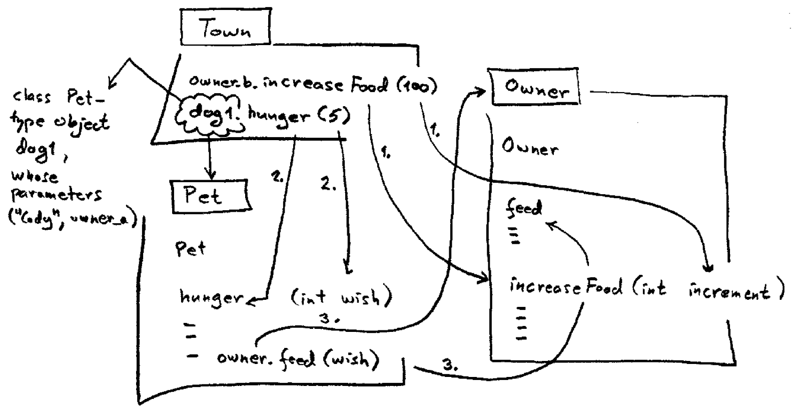
Town
owner.b.increase Food (100)
dog1. hunger (5)
class Pet-type object dog1, whose parameters ("Cody", owner_a)
Pet
Pet
hunger
(int wish)
owner.feed (wish)
Owner
Owner
feed
increase Food (int increment)
1.
2.
3.

Fig. 5. A drawing with classes as central concepts (A5). The drawing contains content element types C2, C4, C5, C6, C7, C16, C17, C18, C21, C22, C24, C25, C27, C28, C35, and C47 (execution of the "main" method within the containing class). (Translated from Finnish and redrawn.)

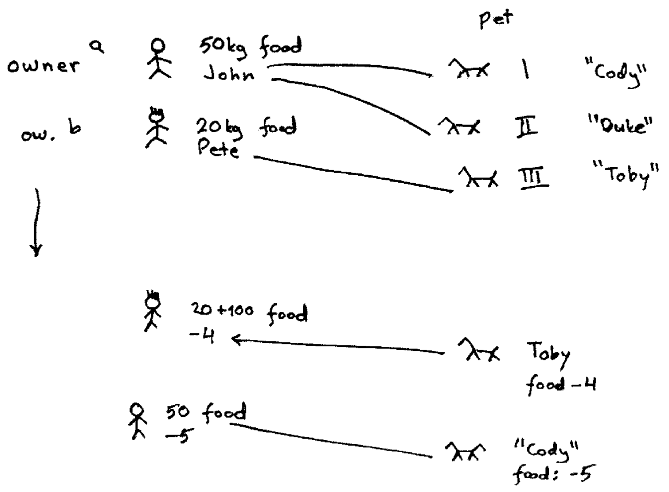

Fig. 6. A drawing describing the path of execution (A7). The drawing has many metaphoric elements and contains content element types C1, C6, C7, C8, C10, C12, C13, C15, C35, C45, and C47 (effect of methods in application domain terminology). (Translated from Finnish and redrawn.)

In approaches A6–A8, the execution path is depicted in some way. In A6, the starting point is the currently active object; in A7, it is the beginning of the whole program execution; in A8, it is the start of the current method call in the "main" method. Elements in the path include objects, method calls, and code fragments. Figure 6 gives an example of approach A7: time flows from top to bottom and objects are duplicated whenever they are needed. These approaches illustrate an explanation of what happens during program execution. Indeed, some students drew a minimal picture accompanied with a long textual answer; the approach of their explanation was most often similar to A7.

Table III. Impact Classes and Their Frequencies (%)

Drawing Task				
1st	2nd	3rd	Code	Impact Class
2	7	0	I1	Jeliot
7	11	7	I2	Metaphor animation
0	0	0	I3	UML
0	0	0	I4	Other

Overall, the variability of different approaches was surprisingly large. A larger data set would probably include some new approaches, but we do not expect that the dominance of A1–A3 would disappear. The increase in the popularity of A8 raises an interesting question: Do students realize that the number of objects in large programs is so huge that it becomes impractical to draw all objects? Another open question is the possible (in)validity of mental representation of OO concepts by students who use various approaches, e.g., A5 where objects are replaced by classes. These questions need more research in order to be resolved.

4.2 Element Analysis

Let us now turn to the individual elements. Table III gives frequencies of various drawing style impact types for each drawing task. An impact was coded if a drawing contained even a small but clear effect that could be interpreted to be caused by some known visualization. However, an object represented by a rectangle containing its attributes was not coded as an impact of UML (Unified Modeling Language) because that is a typical representation used in textbooks. If the attribute values were inside the object and in smaller rectangles, both Jeliot and metaphor animation impact were coded. The single remaining Jeliot impact was the use of Jeliot-type object reference arrows with a special arrow end; metaphor animation impacts were more varying and included the location of methods (i.e., attached to objects), object references represented with pennants, etc.

Table IV gives two figures for each drawing task and each visual element form type: (1) mean, and (2) frequency of drawings with a nonzero score. Note that the figures do not tell the number of elements but the number of different element types, e.g., the number of different rectangle types instead of the number of individual rectangles. For example, in the first task the average number of different rectangle types was 1.2, and rectangles were used in 83% of the drawings. As the average number also includes drawings with no rectangles, mean value of different rectangle types in drawings containing rectangles was higher (about 1.4). The number of individual rectangles cannot be inferred from Table IV.

Text, numbers, and special characters were commonly used in the drawings. On average, 1.3 different text styles (“fonts”) were used to highlight distinction, e.g., between titles and program components, or between object names and attribute identifiers. The mean values of different special characters were small (2.3–2.9), so text consisted mostly of letters and numbers.

Table IV. Form Types and Their Frequencies (%)

Mean Drawing Task			Use Frequency Drawing Task				
1st	2nd	3rd	1st	2nd	3rd	Code	Form Type
1.3	1.2	1.1	100	100	100	F1	Text
0.9	0.9	0.9	78	89	83	F2	Number
N.A.	0.2	0.2	N.A.	19	21	F3	String
2.5	2.9	2.3	85	85	72	F4	Special character
1.2	1.0	0.7	83	67	59	F5	Rectangle
0.2	0.2	0.2	22	19	21	F6	Ellipse
0.1	0.1	0.2	5	7	17	F7	Cloud
0.9	1.1	1.2	71	67	83	F8	Connecting arrow
0.0	0.0	0.0	0	4	3	F9	Pointing arrow
0.4	0.3	0.4	37	30	34	F10	Connecting line
0.2	0.0	0.1	10	4	7	F11	Metaphoric element
0.3	0.3	0.1	24	26	10	F12	Other

Most common graphical elements were rectangles, and connecting arrows and lines. The use of rectangles, however, decreased during the course, compensated for by the increasing use of clouds. Rectangles, ellipses, and clouds were used for the same purposes: to represent objects, classes, methods, attributes, local variables, and parameters.

Table V gives, for each content element and drawing task, three frequencies: (1) incorrect use, (2) correct use, and (3) total use. Note that use was scored if even a single element was present in a drawing. For example, headings (C1) were used correctly in 59% of the drawings in the first task, but the number of individual headings was not counted and cannot be inferred from Table V.

We will first look at the frequencies in the third drawing task, i.e., at the end of the course. Changes during the course will be treated in the next subsection.

The most common element, object (C6, 100%), occurred in all drawings in the third task whereas classes were infrequent (C5, 14%). Attributes, either with their identifier (C12, 41%) or without it (C13, 52%), were frequently used, and their relationship to the owning object was commonly depicted (C15, 72%). Object references were represented more often without the name of the variable (C10, 62%) than with the name (C9, 24%).

Dynamic existence of active method execution was also common (C17, 72%), and so was the past existence of finished methods (C18, 48%). This is in contrast with the rarely depicted existence of finished constructors (C20, 7%), which suggests that students do not consider constructors as important as methods. Furthermore, the existence or execution of the "main" method was not depicted in any drawing (C39, 0%), which suggests that students' mental representation of the "main" method is radically different from other methods. Finally, methods' relationship to the called object was more common (C22, 69%) than their relationship to the calling method (C24, 55%).

Dynamic existence of parameters was usually represented with the value of the parameter only (C27, 55%); the identifier of the parameter was rare (C26, 7%). In the case of local variables, identifiers were rather used (C31, 24%; C35, 79%) than not used (C32, 0%; C36, 0%). The lifetime of parameters and local variables is almost equal, i.e., the duration of a method call; yet, their mental

Table V. Content Elements and Their Frequencies (%) in the Form Incorrect Use + Correct Use = Total Use

Drawing Task				
1st	2nd	3rd	Code	Content Element
0 + 59 = 59	0 + 44 = 44	0 + 34 = 34	C1	Heading
0 + 17 = 17	0 + 22 = 22	0 + 21 = 21	C2	Data type (e.g., int)
0 + 17 = 17	0 + 7 = 7	0 + 7 = 7	C3	Publicity (e.g., public)
0 + 10 = 10	0 + 7 = 7	0 + 3 = 3	C4	Program fragment (at least two lines)
0 + 12 = 12	4 + 22 = 26	3 + 10 = 14	C5	Class
7 + 85 = 93	4 + 93 = 96	0 + 100 = 100	C6	Object
2 + 7 = 10	0 + 19 = 19	7 + 28 = 34	C7	Object’s belonging to its class
2 + 5 = 7	0 + 15 = 15	0 + 21 = 21	C8	Object’s belonging to other objects in the same class
2 + 32 = 34	7 + 37 = 44	3 + 21 = 24	C9	Object reference, id of the referring element visible
12 + 12 = 24	7 + 48 = 56	17 + 45 = 62	C10	Object reference, id of the referring element not visible
0 + 10 = 10	0 + 4 = 4	0 + 0 = 0	C11	Static existence of an attribute
2 + 51 = 54	4 + 48 = 52	0 + 41 = 41	C12	Dynamic existence of an attribute, id visible
0 + 17 = 17	4 + 48 = 52	0 + 52 = 52	C13	Dynamic existence of an attribute, id not visible
0 + 7 = 7	4 + 4 = 7	0 + 3 = 3	C14	Attribute’s belonging to its class
2 + 56 = 59	0 + 81 = 81	0 + 72 = 72	C15	Attribute’s belonging to its object
2 + 56 = 59	0 + 19 = 19	0 + 17 = 17	C16	Static existence of a method
2 + 85 = 88	0 + 78 = 78	0 + 72 = 72	C17	Dynamic existence of an existing method
2 + 66 = 68	0 + 37 = 37	0 + 48 = 48	C18	Dynamic existence of an exited method
0 + 20 = 20	0 + 0 = 0	0 + 3 = 3	C19	Static existence of a constructor
0 + 15 = 15	0 + 4 = 4	0 + 7 = 7	C20	Dynamic existence of an exited constructor
0 + 5 = 5	0 + 7 = 7	0 + 10 = 10	C21	Method’s or constructor’s belonging to its class
27 + 54 = 80	30 + 44 = 74	41 + 28 = 69	C22	Method’s or constructor’s belonging to its object
2 + 0 = 2	0 + 0 = 0	0 + 0 = 0	C23	Static call graph of methods
0 + 5 = 5	0 + 52 = 52	10 + 45 = 55	C24	Dynamic call graph of methods

Table V. Content Elements and Their Frequencies (%) (cont'd)

Drawing Task 1st	2nd	3rd	Code	Content Element
0 + 12 = 12	0 + 7 = 7	0 + 14 = 14	C25	Static existence of a parameter
0 + 12 = 12	0 + 15 = 15	0 + 7 = 7	C26	Dynamic existence of a parameter, id visible
0 + 46 = 46	4 + 56 = 59	0 + 55 = 55	C27	Dynamic existence of a parameter, id not visible
2 + 59 = 61	0 + 63 = 63	0 + 59 = 59	C28	Parameter's belonging to the called method or constructor
0 + 5 = 5	4 + 11 = 15	0 + 3 = 3	C29	Parameter's belonging to the called object
N.A.	0 + 0 = 0	0 + 3 = 3	C30	Static existence of a local variable (not in "main")
N.A.	0 + 30 = 30	0 + 24 = 24	C31	Dynamic existence of a local variable (not in "main"), id visible
N.A.	0 + 0 = 0	0 + 0 = 0	C32	Dynamic existence of a local variable (not in "main"), id not visible
N.A.	0 + 26 = 26	0 + 21 = 21	C33	Local variable's belonging to the called method or constructor
N.A.	0 + 4 = 4	0 + 0 = 0	C34	Local variable's belonging to the called object
0 + 90 = 90	0 + 74 = 74	0 + 79 = 79	C35	Existence of "main" method's local variable, id visible
0 + 2 = 2	0 + 0 = 0	0 + 0 = 0	C36	Existence of "main" method's local variable, id not visible
0 + 10 = 10	0 + 0 = 0	0 + 0 = 0	C37	Main method's local variable's belonging to the "main" method
0 + 68 = 68	0 + 48 = 48	0 + 59 = 59	C38	Main method's local variable's belonging to the referred object
5 + 17 = 22	0 + 7 = 7	0 + 0 = 0	C39	Existence or execution of the "main" method
0 + 5 = 5	4 + 4 = 7	0 + 0 = 0	C40	Main method's belonging to something
0 + 0 = 0	0 + 0 = 0	0 + 0 = 0	C41	Existence of the "this" variable
N.A.	0 + 4 = 4	0 + 0 = 0	C42	String as an object
2 + 7 = 10	0 + 0 = 0	0 + 0 = 0	C43	Point of execution
0 + 5 = 5	0 + 7 = 7	0 + 7 = 7	C44	Existence of an expression or condition
0 + 17 = 17	4 + 37 = 41	0 + 31 = 31	C45	Expression or condition with specific values
0 + 24 = 24	0 + 30 = 30	0 + 21 = 21	C46	Consecutive values of an attribute, variable or, parameter
32 + 37 = 68	33 + 37 = 70	24 + 45 = 69	C47	Other

representations seem to differ. In practice, parameters are typically not assigned a new value within methods, so their task is to transport some value from the calling method to the called method. They can be identified by that value with no need to cite the identifier of the parameter. A local variable, on the other hand, can have many values through its lifetime and cannot be identified by a single value. Therefore, it is natural that students' mental representations of parameters build on the value of a parameter while their mental representations of local variables build on variable identifiers.

This explanation predicts that in the case of attributes, identifiers would be used frequently in drawings. However, this was not the case: Attributes were represented more often without identifiers (C13, 52%) than with identifiers (C12, 41%). The high frequency of missing identifiers, however, is partially due to object reference attributes that were represented with an arrow connecting two objects. The object references did not change during the program and therefore such attributes could be identified by the object they pointed to.

The relationship between attributes, local variables, and parameters and their owning objects or methods was depicted almost always. For example, parameters' relationship to the associated method (C28) was depicted in 59% of the drawings, that is, practically in all cases when parameters were depicted (C26, 7%; C27, 55%). However, local variables of the "main" method (C35, 79%) were never attached to the "main" method (C37, 0%) but to the object they referred to (erroneous element type C38, 59%). Combined with the total neglect of the existence of the "main" method (C39, 0%), this suggests that students' mental representation of the "main" method is so unclear that they cannot depict it in a drawing at all.

4.3 Content Element Development

In Subsection 4.1 several changes in the frequencies of the overall approaches were found during the course. The largest change was a shift from existing-objects-and-all-methods approach to existing-objects-with-subordinated-active-methods approach. This shift is reflected in the element analysis in the decreasing frequencies of static existence of methods (C16: 59%, 19%, and 17% in the first, second, and third task) and static existence of constructors (C19: 20%, 0%, 3%). Frequencies of dynamic existence also decrease but to a smaller extent (C17: 88%, 78%, 72%; C18: 68%, 37%, 48%; C20: 15%, 4%, 7%) Thus, the first mental representation that treats methods as entities with a static existence is replaced during the course by another representation that treats methods by their nested calls (C24: 5%, 52%, 55%).

Frequency of the existence or execution of the "main" method is small and decreases during the course (C39: 22%, 7%, 0%). In the first drawing task methods did not call other methods—the only exception was the "main" method—and the frequency of dynamic call graphs increases dramatically from the first task to the others (C24: 5%, 52%, 55%). Taken together, these changes suggest that the "main" method is not considered by the students to be a method at all. Rather, it seems to be a part of the program code that need not be included in a drawing that depicts program state. Furthermore, this view seems to become stronger during the course.

Some changes may be due to an increase in the economy of expression supported by an increase in learning. For example, the use of headings decreases (C1: 59%, 44%, 34%) perhaps because students do not need them anymore to see, for example, which elements are objects and which are methods; or because these concepts are so familiar to them that they consider headings to be useless for a reader of the drawing. Similarly, items are depicted without their identifiers to a larger extent as learning increases. This is true for object references (id visible, C9: 34%, 44%, 24%; id not visible, C10: 24%, 56%, 62%), attributes (C12: 54%, 52%, 41%; C13: 17%, 52%, 52%), and parameters (C26: 12%, 15%, 7%; C27: 46%, 59%, 55%). In an exam answer, a missing identifier would undoubtedly be considered as an error but the previous finding suggests that it may in fact represent better learning.

Frequency of an object's relationship to the associated class increases during the course (C7: 10%, 19%, 34%) even though classes are depicted only rarely in the first and third task (C5: 12%, 26%, 14%). This suggests that students consider the class of an object more important as their learning increases and that students see classes as a part of the program code that is not a part of the program state. The fluctuation in the frequency of classes suggests that it is hard for the students to understand how classes and program state are actually related.

There are two more content element types with a large (at least 20% units) difference among the three drawing tasks: (1) erroneous positioning of an object reference variable of the "main" method within the object that it refers to (C38: 68%, 48%, 59%) and (2) an expression or condition with specific values (C45: 17%, 41%, 31%). In both, the frequencies fluctuate and we cannot suggest any interpretation.

4.4 Common Errors

Errors made by the students can be divided into two main groups: (1) misunderstanding of OO concepts; and (2) misunderstanding of the studied Java programs. From our point of view, the former group is more interesting. On the other hand, if a large number of errors concerning the studied programs occur with some content element type, then the element type may be hard for students to understand.

Three content element types (C29, C34, C38) were erroneous by nature. Out of these, only C38 was frequent: in more than half of the drawings, local object reference variables of the "main" method were not attached to the "main" method nor were they located outside the box representing the object they referred to, but inside the box. This error is related to the problems with the "main" method described earlier: students seem to have a vague understanding of the role of the "main" method with respect to program state. They don't know where local variables of the "main" method should be depicted and how the "main" method relates to other methods.

The relationship between a method and the associated object (C22) was frequently depicted erroneously with an arrow from the object to the method. When a method is called, space is reserved for its parameters and local variables, including the special variable "this" that refers to the associated object. Thus, the method knows which object it operates on, not vice versa. The direction of arrows may seem to be unimportant but, in fact, it is a sign of a fundamental misunderstanding of the relationship between objects and methods. In drawing after drawing, students used various techniques to show that assignments made within methods affect attributes within objects. (In fact, a large proportion of "other" elements [C47] were of this type; see Figures 2, 3, and 4). These techniques include:

—duplicating an attribute as a local variable with the name of the attribute within the method's rectangle; both the attribute and the duplicate are updated with assignments
—duplicating an attribute as above; initialized with an arrow from the attribute located within the object rectangle; updated with assignments of the method; "returned" with an arrow back to the attribute at the end of the method
—duplicating an attribute as above; initialized with zero; updated with assignments of the method; no effect on the attribute within the object
—representing an attribute as a local variable of the method only; no reference to the object and no indication of that attribute within the object

These examples demonstrate that these students did not have an exact understanding of how methods operate on attributes. Many of the strategies used by the students manage to produce a correct outcome in simple cases, but they fail in more complicated cases where nested method calls—let alone independent threads—operate on the same attributes. The programs used in the drawing tasks did not contain such complex cases and more research is needed to find out how students manage in such situations.

Object references (C10) were often depicted erroneously with an arrow pointing to the wrong direction: from the referenced object to the referring object. Moreover, objects were often identified with the value of one of their attributes, for example, objects of the class "Pet" with the value of the attribute "Name". This occurred both in the object itself; that is, an object could be depicted with the value of the attribute only, and in object references, that is, an object reference was depicted as the value of the attribute in the referenced object. Neither of these errors—using wrong direction of reference and identifying objects with attributes—cause no harm as long as object references and attribute values do not change which is the case in many textbook examples. It is possible that students thus develop a wrong mental representation of object references and identification, but our data do not allow us to study this question further.

Errors with object references and identification can be partially explained also by students' use of application domain knowledge. For example, in real life dogs are identified by their names and the direction of object references

Table VI. Form Types F4 and F11 and Their Frequencies (%) Depending on the Visualization Tool Last Used

Mean Drawing Task			Use Frequency Drawing Task				
1st	2nd	3rd	1st	2nd	3rd	Code	Form Type
2.2	3.3	3.8	78	93	100	F4/J	Special character (after Jeliot animation tool)
2.7	2.4	1.5	88	75	60	F4/M	Special character (after metaphor animation tool)
0.0	0.0	0.0	0	0	0	F11/J	Metaphoric element (after Jeliot animation tool)
0.3	0.1	0.2	17	8	13	F11/M	Metaphoric element (after metaphor animation tool)

between owners and dogs may represent the relation own rather than the relation belong to which is used in the program. Two frequent error types are clearly affected by application domain knowledge: First, in the program of Figure 1, pets ask their owners to feed them but in several drawings this relationship was reversed and the owner was depicted as the caller of the feed method. Second, in the other program, trains could be attached to each other but driving one train did not affect the odometer attribute of the attached train; yet in many drawings odometers of both trains were updated. The use of application domain knowledge may help students in obtaining correct answers in exams but it does not help in debugging programs. Thus, its use should be considered harmful in programming education.

Finally, the static and dynamic existence of methods were mixed in various ways. These errors may be related to the changes in the frequencies of static and dynamic method existences (C16, C17, C18) and provide more evidence to the suggestion that students' mental representation of methods changed its form during the course.

4.5 Effects of Animation Tools

In the previous subsections, students have been treated as a single group, i.e., no attention has been given to which visualization tool they have last used. In some element types the visualization tools seem to have effects, but in others they don't. For example, consider the frequencies of special characters (F4) and metaphoric elements (F11) in Table VI.

At first sight, the use of special characters (i.e., characters that are not letters or digits) seems to be larger if the last visualization tool has been Jeliot rather than metaphor animation (78%, 93%, 100% in F4/J versus 88%, 75%, 60% in F4/M). However, because all students used one tool before the first drawing task and the other tool after it, the same students that appear in the Jeliot row in the first drawing task appear in the metaphor tool row in the other tasks. Thus, the frequencies of using special characters in the three tasks are for these students 78%, 75%, and 60% respectively. Similarly, for the students who saw the metaphor animation tool first, these frequencies are 88%, 93%, and 100%. Thus, the difference between these two groups may also be due to the individual differences between students in these two groups.

Table VII. Impact Classes I1 and I2 and Their Frequencies (%) Depending on the Visualization Tool Last Used

Drawing Task				
1st	2nd	3rd	Code	Impact Class
6	0	0	I1/J	Jeliot (after Jeliot)
0	17	0	I1/M	Jeliot (after metaphor)
6	0	0	I2/M	Metaphor animation (after Jeliot)
8	25	13	I2/M	Metaphor animation (after metaphor)

On the other hand, the effect of the last used tool is clear in metaphoric content (F11/J versus F11/M): after Jeliot use, drawings have no metaphoric elements; after the metaphor animation tool, metaphoric content is found in 8–17% of the drawings. It should be noted that the metaphoric elements found in the drawings are not necessarily the same metaphors as in the animation tool. For example, the state of the Java program that dealt with trains, was depicted by one student by pictures of trains, and the service method by a picture of a wrench. Thus, a visualization based on rectangles and arrows seems to limit students' representations to neutral, nonmetaphoric graphics.

Table VII shows the frequencies of impact classes I1 (evident Jeliot impact in a drawing) and I2 (evident metaphor animation impact) depending on the visualization tool last used. A small impact of Jeliot is seen in the first drawing task after Jeliot and in the second drawing task if made after metaphor animation. However, only one of these impacts is genuine to Jeliot: Others can be impacts of metaphor animation as well. Impact of metaphor animation is larger and more genuine (pennants, garbage vehicle, ...) but it disappears after the use of Jeliot. Thus, it seems that a visually stronger animation tool causes more effects on the drawings and hence on mental representations but this effect can be erased by the use of a more neutral tool.

In content elements frequencies, the effect of the last used visualization tool is sometimes clear. Overall, object references were used in the third task more often without the identifier of the referring variable (C10, 62%) than with the identifier (C9, 24%). This difference is due to students that have used the metaphor tool last (C10/M, 80%; C9/M, 13%); the difference among the other group is opposite and small (C10/J, 33%; C9/J, 42%). A similar effect can be seen in the use of identifiers with attributes: students who have used the metaphor tool last are less willing to use attribute identifiers (C13/M, 67%; C12/M 33%) than students who have used Jeliot last (C13/J, 33%; C12/J 50%). Both tools use identifiers in the same way in their visualizations, so the difference must have some other roots. In Subsection 4.3, we hypothesized that a low frequency in identifier use reflects better learning but this requires further study.

Another clear difference between the tools can be found in the frequencies of methods. In the third task, students who had used Jeliot last included existing methods and exited methods equally in their drawings (C17/J, 83%; C18/J, 83%). On the other hand, students who had used the metaphor tool last included exited methods in their drawings only rarely (C17/M, 60%; C18/M, 20%). The visualization of method calls is very different in the tools: Jeliot uses a method call stack that shows only the uppermost method call clearly. Metaphor animation depicts methods as workshops that are not stacked on each other. Thus, the set of currently existing method calls can be seen clearly in the metaphor tool but not in Jeliot.

In Subsection 4.4, it was found that the relationship between a method and the associated object (C22) was frequently depicted erroneously. This error is much more common among students who have used Jeliot last in the second task (C22-wrong/J, 40%; C22-wrong/M, 17%) and in the third task (C22-wrong/J, 67%; C22-wrong/M, 20%); in the first task no difference can be found (C22-wrong/J, 28%; C22-wrong/M, 25%). In Jeliot, this relationship is depicted with an arrow within a set of tightly aligned arrows; the direction of the arrow is correct but the arrow is hard to see because of the other arrows. In the metaphor tool, method workshops are attached to the object watch panels; there is no indication of any direction of this attachment. The reason for the better correctness after the metaphor tool remains an open question.

Finally, students who had seen Jeliot last were more willing to use expressions or conditions with specific values (C45/J, 50%; C45/M, 20%) and consecutive values of attributes and variables (C46/J, 33%; C46/M, 13%) in the third task. Jeliot visualizes the evaluation of expressions while the animation tool does not. Therefore, Jeliot stresses the importance of expression evaluation which may result in a stronger position of expressions in students' mental representations of the execution of a program.

4.6 General Discussion

We have studied students' visualizations of OO program state and the notional OO machine by asking them to draw pictures of given programs at a specific point of execution. Such tasks were given three times along an elementary Java programming course. The drawings were analyzed for their overall approach, frequencies of specific element types, and systematically occurring errors.

The first striking finding was the large variability in the overall approaches used by the students. We expected that students would almost always mimic drawings they had seen in lectures, textbooks, and program visualization tools, but this turned out not to be the case. When we started our analysis and looked at the first drawings of the first task, and each of the first six seemed to have its own approach, we realized that students' understanding of central OO concepts, their relationships, and their dynamic manifestation do vary a lot. We have also started to collect similar drawings from other institutes and, in each case, the teacher of that class has been surprised to see in how many different ways his or her students understand and depict program state.

Students' drawings do not differ only in their overall approach, but in their detailed contents, also. Some drawings are complicated and represent many aspects of the program state while others consist of a few elements only. The results indicate that students' mental representations of OO concepts and program execution not only grow as new material is covered in teaching, but they also change. The first drawings treat methods as having primarily a static existence whereas later in the course, methods are seen as dynamic invocations that call each other. The role of classes in program execution fluctuates during learning, which means that students have problems in locating the notion of class in its proper place with respect to, e.g., objects. Eckerdal and Thuné [2005] have found several categories that describe how students perceive the concepts of class and object, and Teif and Hazzan [2006] have found several misconceptions concerning these concepts. Our study confirms that students' mental representations of OO concepts do change during learning and, thus, supports their findings. We have moreover documented specific changes in the contents of the drawings.

We found two major sources of problems in the drawings: (1) the relationship between object and method, and (2) the role of the "main" method with respect to program state. These problems manifested themselves in many different forms: misconceptions about assignments to attributes, object references located inside the referred objects etc. Other problems were caused by a too-simplistic understanding of object identification and (improper) use of application domain knowledge.

Finally, program animation tools seem to have effects on students' visualizations. After the use of a metaphor-based animation tool, students' visualizations contained metaphoric elements—both those used in the tool and those invented by the student. After the use of a technical animation tool, metaphoric content disappeared. Moreover, the impact of metaphor animation seemed to be larger than that of the tool with more technical visualization. Ragonis and Ben-Ari [2005] have suggested the use of dynamic program visualization to prevent certain misconceptions; our study suggests that the effects of different dynamic visualizations do differ, so the selection of a tool deserves careful consideration.

4.7 Validity Issues

In this study, students were asked three times during an elementary Java programming course to draw a picture of program state including objects and methods. Our goal was to find out what kind of a notional OO machine students perceive and what misconceptions they have of OO concepts.

The students had seen various visualizations of programs and program execution during lectures and they had also used two program animation tools. Nevertheless, students drawings do only occasionally follow these examples and they show a large variability both in overall approach and in details of expression. Thus, it seems that even though effects of prior exposure to example visualizations can be identified, students used their own understanding to produce their drawings. On the other hand, repeated use of the same task type may have resulted in (perhaps unconscious) mental search for the "correct" drawing and deeper mental processing of the structure and contents of the notional OO machine. Thus, the study design may have caused changes in students' drawings in the later tasks.

Individual students could not be traced between tasks, which weakens our results. In some cases the same participant could be tracked (e.g., because of special handwriting), and some students seemed to change their drawing styles between tasks more than others. As only few students could be traced, the data does not allow the study of individual differences.

The students were not interviewed and it is therefore not clear whether some misconceptions are misunderstandings regarding drawing conventions or instructions rather than misunderstandings of OO concepts. On the other hand, we have collected evidence from different viewpoints to support our conclusions. For example, the wrong direction of arrows with method calls could be a result of carelessness, but the numerous errors in manipulating attributes supports our suggestion that the relationships between methods and objects are poorly understood.

A possible weakness of the analysis is that it is based on the selected classification of drawings' elements. The classification was used by two researchers who could easily agree on the scoring in individual cases. Thus, the results presented above are based on solid data. On the other hand, there may be other classifications that could provide insight on issues that are not covered by the used classification.

The comparison of the two visualization tools suffers from the low usage of the tools during the course. Both tools are supposed to be used more extensively during lab exercises and, in case of the metaphor tool, the metaphors should be used during lectures. Thus, a proper use of the tools would require two student groups with extended use of a single tool in each group.

Finally, our analysis does not tell whether the identified misunderstandings can be easily overcome with suitable teaching techniques, whether the effects of exposure to a visualization tool are transient or whether the results can be generalized to other learning contexts, etc. However, we consider our study to be a step toward the analysis of students' understanding of OO program execution and the notional OO machine.

5. CONCLUSION

We have studied students' visualizations of OO program state in order to see how students perceive program execution: What are the central concepts and relationships that make up the program state? The results indicate that students have a vague understanding of dynamic aspects of OO programs; moreover, their mental representations not only grow as new things are learned, but they also change during learning. We have also found differences in students' representations after they used different program visualization tools.

We also identified several open issues that could not be resolved based on our data. These include the (in)validity of mental representation of OO concepts within students who use uncommon overall approaches; reasons for students' invalid mental models of object references; connection between learning and frequency of identifier use in visualizations; students' ways of coping with attributes manipulated by methods; and detailed effects of animation tools on students' understanding of the notional OO machine.

We are currently collecting similar data from other institutes in order to get a better understanding of this phenomenon. We will also revise the analysis scoring because in the current form the general approach is not taken into account and errors are identified in nonuniform ways.

We hope that this research will give a better understanding of novices' mental representations of OO concepts and program state, provide a method to systematically reveal students' misconceptions, and eventually lead to better programming education techniques.

ACKNOWLEDGMENT

We would like to express our gratitude to Pirkko Voutilainen and Matti Tedre, the teachers of the course.

REFERENCES

DOUGLAS, S., HUNDHAUSEN, C., AND MCKEOWN, D. 1995. Toward empirically-based software visualization languages. In *Proceedings of the 11th IEEE International Symposium on Visual Languages*. IEEE Computer Society Press, 342–349.

DU BOULAY, B. 1989. Some difficulties of learning to program. In *Studying the Novice Programmer*, E. Soloway and J. C. Spohrer Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 283–299.

EBRAHIMI, A. AND SCHWEIKERT, C. 2006. Empirical study of novice programming with plans and objects. *SIGCSE Bull. 38,* 4, 52–54.

ECKERDAL, A. AND THUNÉ, M. 2005. 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)*. ACM, New York, NY, 89–93.

FLEURY, A. E. 2000. Programming in Java: student-constructed rules. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE'00)*. ACM, New York, NY, 197–201.

FORD, L. 1993. How programmers visualize programs. In *Empirical Studies of Programmers: Fifth Workshop*, C. R. Cook, J. C. Scholtz, and J. C. Spohrer Eds. Norwood, NJ. Ablex Publishing Company.

GARNER, S., HADEN, P., AND ROBINS, A. 2005. My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian Conference on Computing Education (ACE'05)*. Australian Computer Society, Inc., Darlinghurst, Australia, 173–180.

GOOD, J. AND BRNA, P. 2004. Program comprehension and authentic measurement: A scheme for analysing descriptions of programs. *Int. J. Hum.-Comput. Stud. 61,* 2, 169–185.

GROSS, P. AND POWERS, K. 2005. Evaluating assessments of novice programming environments. In *Proceedings of the International Workshop on Computing Education Research (ICER'05)*. ACM, New York, NY, 99–110.

HOLLAND, S., GRIFFITHS, R., AND WOODMAN, M. 1997. Avoiding object misconceptions. In *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'97)*. ACM, New York, NY, 131–134.

HOLLIDAY, M. A. AND LUGINBUHL, D. 2004. CS1 assessment using memory diagrams. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)*. ACM, New York, NY, 200–204.

HÜBSCHER-YOUNGER, T. AND NARAYANAN, N. H. 2003. Dancing hamsters and marble statues: Characterizing student visualizations of algorithms. In *ACM 2003 Symposium on Software Visualization (SoftVis'03)*. ACM, New York, NY, 95–104.

HUNDHAUSEN, C. D., DOUGLAS, S. A., AND STASKO, J. T. 2002. A meta-study of algorithm visualization effectiveness. *J. Vis. Lang. Comput. 13*, 259–290.

KELLEHER, C. AND PAUSCH, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comp. Surv. 37,* 2, 83–137.

KÖLLING, M. 2007. BlueJ—the Interactive Java Environment, http://www.bluej.org/. (accessed July 10th, 2007).

LEWANDOWSKI, G., GUTSCHOW, A., MCCARTNEY, R., SANDERS, K., AND SHINNERS-KENNEDY, D. 2005. What novice programmers don't know. In *Proceedings of the 2005 International Workshop on Computing Education Research (ICER'05)*. ACM, New York, NY, 1–12.

LISTER, R., ADAMS, E. S., FITZGERALD, S., FONE, W., HAMER, J., LINDHOLM, M., MCCARTNEY, R., MOSTRÖM, J. E., SANDERS, K., SEPPÄLÄ, O., SIMON, B., AND THOMAS, L. 2004. A multinational study of reading and tracing skills in novice programmers. *ACM SIGCSE Bull. 36,* 4, 119–150.

MORENO, A., MYLLER, N., SUTINEN, E., AND BEN-ARI, M. 2004. Visualizing programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'04)*. Gallipoli, Italy. ACM, New York, NY, 373–376.

OR-BACH, R. AND LAVY, I. 2004. Cognitive activities of abstraction in object orientation: An empirical study, *SIGCSE Bull. 36,* 2, 82–86.

PENNINGTON, N. 1987. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*. G. M. Olson, S. Sheppard, and E. Soloway Eds. Ablex Norwood, NJ, 100–113.

RAGONIS, N. AND BEN-ARI, M. 2005. On understanding the statics and dynamics of object-oriented programs. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'05)*. ACM, New York, NY, 226–230.

ROBINS, A., ROUNTREE, J., AND ROUNTREE, N. 2003. Learning and teaching programming: A review and discussion. *Comput. Sci. Educ. 13*, 137–172.

SAJANIEMI, J., BYCKLING, P., AND GERDT, P. 2006. Metaphor-based animation of OO programs. In *Proceedings of the ACM Symposium on Software Visualization (SOFTVIS'06)*. ACM, New York, NY, 173–174.

SAJANIEMI, J. AND KUITTINEN, M. 2004. Visualizing roles of variables in program animation. *Informa. Visualiza. 3,* 3, 137–153.

SANDERS, K. AND THOMAS, L. 2007. Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM, New York, NY, 166–170.

SCHULTE, C. AND BENNEDSEN, J. 2006. What do teachers teach in introductory programming? In *Proceedings of the International Workshop on Computing Education Research (ICER'06)*. ACM, New York, NY, 17–28.

TEIF, M. AND HAZZAN, O. 2006. Partonomy and taxonomy in object-oriented thinking: Junior high school students' perceptions of object-oriented basic concepts. *SIGCSE Bull. 38,* 4, 55–60.

THOMAS, L., RATCLIFFE, M., AND THOMASSON, B. 2004. Scaffolding with object diagrams in first year programming classes: Some unexpected results. In *Proceedings of the 35th SIGCSE Technical Symposium on CS Education (SIGCSE'04)*. 250–254.

THOMASSON, B., RATCLIFFE, M., AND THOMAS, L. 2006. Identifying novice difficulties in object oriented design. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)*. ACM, New York, NY, 28–32.

TOLHURST, D., BAKER, B., HAMER, J., BOX, I., LISTER, R., CUTTS, Q., PETRE, M., DE RAADT, M., ROBINS, A., FINCHER, S., SIMON, S., HADEN, P., SUTTON, K., HAMILTON, M., AND TUTTY, J. 2006. Do map drawing styles of novice programmers predict success in programming?: A multinational, multiinstitutional study. In *Proceedings of the 8th Australian Conference on Computing Education (ACE'06)*. Australian Computer Society, Inc., Darlinghurst, Australia, 213–222.

VAINIO, V. AND SAJANIEMI, J. 2007. Factors in novice programmers' poor tracing skills. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM, New York, NY, 236–240.

WINSLOW, L. E. 1996. Programming pedagogy — a psychological overview. *SIGCSE Bull. 28*, 17–22.

Received October 2007; accepted October 2007