Computer Science Education
Vol. 15, No. 3, September 2005, pp. 203 – 221

Routledge
Taylor & Francis Group

A Long-Term Investigation of the Comprehension of OOP Concepts by Novices

Noa Ragonis* and Mordechai Ben-Ari
Department of Science Teaching, Weizmann Institute of Science, Rehovot 76100, Israel

This article describes research on the learning of object-oriented programming (OOP) by novices. During two academic years, we taught OOP to high school students, using Java and BlueJ. Our approach to teaching featured: objects-first, teaching composed classes relatively early, deferring the teaching of main methods, and focusing on class structure before algorithms. The research used a constructivist qualitative research methodology using observations and field notes, audio and video recordings, and an analysis of artifacts such as homework assignments. The findings were divided into four primary categories: class vs. object, instantiation and constructors, simple vs. composed classes, and program flow. In total, 58 conceptions and difficulties were identified. Nevertheless, at the end of the courses, the students understood the basic principles of OOP. The two main contributions of this research are: (i) the breadth and depth of its investigation into the concepts held by novices studying OOP, and (ii) the nature of the constructivist qualitative research methodology.

1. INTRODUCTION

1.1. The Paradigm Issue

There is a virtual consensus that when teaching fundamentals of computer science the focus should be on scientific principles, problem-solving and project development skills, rather than on specific artifacts such as languages and operating systems (Gal-Ezer, Beeri, Harel, & Yehudai, 1995; ACM/IEEE, 2001). However, even if we emphasize principles, there are a number of significantly different approaches to program design and implementation that are called *paradigms*. The difference between paradigms is in the way the program designer analyzes a given problem, and in the presentation of the design (Détienne, 2001). There is much disagreement among computer science (CS) educators concerning the roles of paradigms in teaching. Studies have dealt with the following topics: What skills are required from

*Corresponding author. Department of Science Teaching, Weizmann Institute of Science, Rehovot 76100, Israel. E-mail: noa.ragonis@weizman.ac.il

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/05/030203-19 © 2005 Taylor & Francis
DOI: 10.1080/08993400500224310

the learners in each paradigm? Should a learner study more than one paradigm? What paradigm should be taught first? What happens when transferring from one paradigm to another? (Osborne, 1992; Mazaitis, 1993; Reid, 1993; Brilliant & Wiseman, 1996; Maheshwari, 1997; Wegner, 1997; Hadjerrouit, 1998a; 1998b; Bergin, 1999; Lewis, 2000; Bucci, Heym, Long, & Weide, 2002).

In recent years, many professional software engineers have come to use the object-oriented paradigm, and as a result, object-oriented analysis, design and programming have found their way into computer science education. This presents CS educators and researchers with a set of dilemmas as to the place of object-oriented programming (OOP) in the curriculum and a set of challenges to develop suitable pedagogies for teaching it (Rosson & Alpert, 1990; Brilliant & Wiseman, 1996; Schoenefeld, 1997; Bishop & Bishop, 2000).

The objective of this research project was to take one particular approach called objects-first, to teach it to novices, and to study their learning in great depth. We hoped to answer the following questions: Is objects-first a viable approach for teaching novices? What are the conceptions that novice students build? What are their difficulties? What recommendations can be derived concerning ways of teaching OOP to novices?

1.2. The Research Background

During the past decade, an up-to-date curriculum was developed for teaching computer science in Israeli high schools (Gal-Ezer, Beeri, Harel, & Yehudai, 1995; Gal-Ezer & Harel, 1999). A student can study CS for one, three or five units, each unit consisting of 90 hours (three hours a week for a school year). The first two units, called ''Foundations of Computer Science,'' teach algorithms and programming with the procedural paradigm. For the third unit, the student can elect to study an application such as graphics or an alternate paradigm such as logic programming. The fourth unit, called ''Software Design,'' teaches modularization and basic data structures such as stacks, lists and trees. The fifth unit is theoretical and includes alternatives such as automata theory and concurrent programming.

Because of the popularity of the OOP paradigm, we investigated the feasibility of starting with OOP in the ''Foundations'' course. Teaching OOP for high school novices is being done elsewhere (Lattu, Tarhio, & Meisalo, 2000; Schulte & Niere, 2002), including in the U.S. with the decision to change the paradigm and language of the Advanced Placement examinations (The College Board, 2004). Thus the methodology and the results of this project should be widely applicable throughout the CSE community.

2. COURSE DEVELOPMENT

In this section, we describe the learning unit that we developed as part of the project: the programming language and development environment, the teaching approach, and the objectives and instructional techniques that we chose.

2.1. Choosing a Language and an Environment

Educators and researchers agree that the specific computer language is not the most important aspect of a course in OOP, as long as it can be used to teach the principles of OOP (Biddle & Tempro, 1998; Bishop, 1997; Hadjerrouit, 1998b). There are some comparative studies regarding the language and development environment most suitable for novices (King, 1997; Kölling, 1999a, 1999b). There are those who support less common languages such as Smalltalk or languages that were developed for teaching (Kölling & Rosenberg, 1996b; Fernández & Peña, 2002). However, the overwhelming majority of educators these days use Java (Biddle & Tempro, 1998; Bishop, 1997), although the language presents some pedagogical problems (Biddle & Tempro, 1998; Hadjerrouit, 1998a; Lewis, 2000). We did not stray from this consensus and chose Java as our programming language. Java is an objects-only language and thus discourages students from writing procedural programs. In addition, the language and a plethora of pedagogical tools are available free of charge. Finally, the CSE community uses Java extensively so that we could benefit from ongoing development and research of pedagogical tools and techniques.

As the development environment we chose BlueJ (Kölling & Rosenberg, 2000, 2001; Kölling, Quig, Patterson, & Rosenberg, 2003) because it was designed for teaching. BlueJ provides all the usual tools for editing, compiling and debugging programs written in Java, but its significant advantage is the visual presentation of classes and objects, and its support for interactive invocation of constructors and methods. This enabled us to demonstrate and exercise the fundamental concepts of OOP at the beginning of the course without getting bogged down in the details of the programming language. The subsequent transition to studying the language was easy. The simplicity of BlueJ was a significant issue. Given the young age of our students, we wanted to be careful not to inflict upon them the complexities of a professional environment. In the years since this study was launched, BlueJ has become popular in many academic institutions (Buck & Stucki, 2000; Kölling & Rosenberg, 2001; Nourie, 2002) and many studies have been published regarding using it in teaching (Kölling & Rosenberg, 1996a, 2001; Patterson, Kölling, & Rosenberg, 2003; Kölling, Quig, Patterson, & Rosenberg, 2003; Barnes & Kölling, 2003; Thramboulidis, 2003). Were we to begin the project today, we would evaluate other environments such as DrJava and jGrasp (Allen, Cartwright, & Stoler, 2001; Hendrix, Cross, & Barowski, 2004). Although Jeliot is primarily an animation tool for Java programs, the newest version could also be considered as a development environment for novices (Ben-Ari, Myller, Sutinen, & Tarhio, 2002).

2.2. Syllabus and Pedagogy

Our central meta-objectives were to have the students come to understand the following important concepts in OOP: modularity, encapsulation, and information hiding. Operationally, we wanted the students to be able to approach a problem by dividing it into classes and implementing the classes in Java. We knew that when teaching objects-first, there are a large number of elementary concepts that the

student must learn together. We can no longer start with a one-line ''Hello world'' program; instead, classes, objects, methods, fields, parameters, constructors, and so on must be presented in an integrated fashion. Therefore—and unlike the existing ''Foundations'' course—we decided to defer the study of algorithms until after the basic OOP concepts have been thoroughly covered.

Two specific dilemmas pursued us through the project. The first was when and how to present a *composed class*, which is the term we use for a class that has fields of a user-defined class and not just fields of primitive types or predefined types such as String. We decided—in the interests of better understanding of the paradigm—to teach composed classes relatively early. We were encouraged in this decision by the support given by BlueJ for working visually with composed classes. The second dilemma was when and how to present the main method, since BlueJ enables the programmer to do significant work without one. Initially, we deferred teaching the main method, believing that it is too procedural and thus would interfere with the understanding of the OOP paradigm. Eventually, we came to see that the main method should not be postponed indefinitely, because that interferes with understanding dynamic aspects of the program.

Here is a description of the teaching sequence that developed over time:

- We introduced the concepts of class and object using diagrams.
- Interactive method invocation in BlueJ was used to familiarize the students with operations on objects.
- Students were then taught to program classes in Java, including attributes (fields), constructors and methods. The methods included mutators and accessors, and simple assignments and expressions.
- The complexity of the student activities was gradually increased. First they used existing classes, then they changed lines in method bodies, then they wrote methods and only then did they build new classes.
- The next step was building composed classes out of simple classes.

Once these concepts were understood, the syllabus continued with the usual topics: control statements, arrays and so on.

3. RESEARCH DESIGN

3.1. Rationale

An extensive survey of the literature on teaching OOP showed that the following subjects have interested researchers: OOP vs. procedural programming, languages, environments and other tools such as visualizations, teaching approaches and relevant teaching theories such as constructivism and cognitive apprenticeship.

Researchers who chose the objects-first approach described concepts that need to be taught and their sequence. The OOP model must be exposed from the beginning

by using simple projects that include several classes (Lewis, 2000). In introducing objects, the following concepts have to be treated: the object state, changing the object state and the way objects relate to each other (Woodman & Holland, 1996). It is better to postpone the study of control flow, complex statements and advanced OOP topics such as inheritance (Stein, 1997). Educators agree that there is need for a different pedagogical approach for teaching OOP (Bishop, 1997; Bergin, 1999, 2000; Mitchell, 2001; Neubauer & Strong, 2002), and that there is lack of proven pedagogical approaches, appropriate books and suitable environments for novices (Börstler, Johansson, & Nordstrom, 2002; Bucci, Heyn, Long, & Weide, 2002). Some authors focus on integrating constructivist learning theories into OOP teaching and learning (Hadjerrouit, 1999; Thramboulidis, 2003). Holland, Griffiths, & Woodman (1997) gave a detailed list of students' misconceptions regarding OOP concepts and suggested a source for each of them. Examples are: equality between a class and an object; operations used only for computations and not for changing attributes or for their side-effects such as printing; confusion between an object and a "name" attribute. Fleury (2000) found that students construct their own rules. Examples are: since two methods in a class cannot have the same signature, students assumed that this rule holds also for methods in different classes; they also thought that object creation relates only to executing the constructor method and not to allocating memory.

Researchers claim that high cognitive demands are required from learners of OOP: abstraction, analysis, design, analogy, and more (Hadjerrouit, 1998a, 1998b, 1999; Parlante, 1997; Woodman, Davies, & Holland, 1996). Détienne (2001) dedicated an entire book to the cognitive aspects of software design in the OOP environment. Her emphasis is on abstraction dimensions especially those that relate to beginners, such as mapping from the problem domain to the programming domain.

Most of the previous studies on OOP described a single lecturer/investigator who gave a course on the topic, and determined the contents of the course and the order of their presentation. The studies reported the problems that arose while teaching the course, and suggestions for improvements. Other studies were based on a single questionnaire, which aimed to examine specific concepts, or to do a comparison between novices and experts, or between programmers in different paradigms. We did not find any publications that reported a long and formal research project on teaching OOP. There was a noticeable absence of research projects dealing with young novices such as high school students, though recently a report has been published describing a three-year research project in progress at Oslo University (Berge, Fjuk, Groven, Hwgna, & Kaasbøll, 2003).

3.2. Research Questions

The research questions were formally posed as follows:

1. What key concepts of OOP are important and can be included in an introductory course?

2. What conceptions do novice students build when learning fundamental concepts of OOP?
3. What is a suitable teaching sequence for teaching OOP to novices?

3.3. Population and Implementation

During the academic year 2000–2001, we taught a two-hour per week class (approximately 50 total hours of teaching) in OOP to 18 novices studying CS in the tenth grade (ages 15–16). We repeated the course during 2001–2002 with a class of 29 students.[1] In each meeting, one hour was dedicated to lectures and discussion in the classroom, and the other to demonstrations and exercises in a computer lab.

Lectures were given by the second author while the first author observed and took comprehensive field notes. Additional data gathered included all the homework, lab exercises, tests and projects. See Ragonis (2004) for a complete analysis of the data.

The experimental setup was slightly less than optimal because the students continued to study the procedural "Foundations" course as required by the national curriculum. Furthermore, the OOP course was defined as part of the students' "enrichment lessons," so the students did not feel the same commitment to performing the tasks we required as they did in their required courses. Despite these problems, we were more than satisfied with the amount and quality of the research material gathered. We grew to know the students, the students got to know us, the topic and our demands, and the length of the experiment meant that any local crises did not affect the long-term quality of learning and research.

4. RESEARCH METHODOLOGY

This research used a constructivist qualitative research methodology, based upon constructivist learning theories that relate to reality as a human structure, formed by the cultural and personal conditions of the researcher and her research population (Sabar Ben-Yehoshua, 2001). This study presents a series of meetings between a world of understandings of the researcher and the perception of reality by the population under investigation. The understandings and perceptions of the researcher compared with the understandings and perceptions of the students affected and changed each other. The process is that of a *participating observer* who is an investigator involved in the field she researches, and where she constitutes part of this field (Hazan, 2001). According to this approach, the researcher is conscious of the field, conscious of herself, affects reality, but also creates it (Phillips, 1990). The voice of the researcher is present throughout the research work. The researcher does not come into the research field *tabula rasa*, but possesses previous attitudes regarding the field of study (Hammersley, 1995). The experience of the researcher affects her development, views and attitudes (Ely, Vinz, Downing, & Anzul, 1997). This was true in our case because of our extensive experience in CS education, as teachers, as developers of learning materials and as researchers.

The characteristics of this study were similar to those of *action research*, in which the researcher is versed in her subject matter, locates or feels a problem and tries to

identify and solve the problem (Shkedi, 2003). The entire process is documented and described, and in the end an evaluation is made as to whether the objective was attained. When the investigated phenomenon is complex, there is a significant advantage to a researcher who experiences it from the inside and can describe it from a point of view close to that of the students.

It is important to note that the processes and understandings that took place in the students are the heart of the study rather than their achievements. Quantitative measures of achievement were used only sparingly to support the qualitative findings.

Extensive data collection was carried out through the entire period: observations and field notes, audio and video recordings, and collection of artifacts. The latter—homework assignments, class work, examinations, final project—proved to be especially fruitful in that they showed precisely what concepts were understood and what concepts were problematical.

On-the-fly analysis of the results of the first year of teaching enabled us to draw important conclusions and led to changes in the syllabus and pedagogy used in the second year. Some of those conclusions were:

- Classes and objects should be introduced first using diagrams.
- There is a need to use algorithms that are well-suited to the paradigm.
- Examples using graphics should be avoided because novice students conflated the ''object'' with its rendering on the screen (and when using BlueJ with its icon in the environment).
- Problems relating to computer systems should be used, not just ''real-life'' problems involving employees and animals.
- Problems that use loops do not naturally appear within the context of elementary OOP, so the topic should be deferred and taught together with array objects.

The importance of the first year lay in confirming our basic assumption that the OOP paradigm can be taught to young novices, although issues of syllabus and pedagogy are critical to successful learning.

5. RESULTS AND DISCUSSION

5.1. Data Analysis and Presentation

The amount of data collected was massive. To make sense of so much data, the analysis was performed as follows: first, the questionnaires, tests and homework were reviewed to extract interesting episodes. These were then formally analyzed, using the field notes to provide the context and clarifications needed to understand the episodes. The analysis was done in several phases using various modes of presentation:

1. The analysis from the first year was summarized in a ''research narrative,'' describing in detail the course of teaching and the research results. This analysis was used as the basis for preparing the instruction during the second year.

2. A concept map for OOP was built to serve as a reference against which all subsequent analysis was conducted.
3. Significant episodes were identified and characterized according to the OOP concept involved. Episodes were significant, not only when they showed difficulties and errors shown by the students, but also when they showed evidence of comprehension of a concept. We were not just cataloguing misconceptions, but trying to understand what made for effective learning!
4. Episodes were assigned to one or more categories of OOP concepts. As needed, sub-categories of concepts were created in each category.
5. The categories and sub-categories were structured into a tree. We chose four categories that were found to be significant and that seemed to be sufficient to provide answers to the research questions. We chose categories that: (i) related to main OOP concepts, (ii) affected comprehension and perception of the discipline, and (iii) contained many important findings. We dropped categories concerning the programming language since language per se was not significant in this learning unit.
6. The four categories that we chose were: *class vs. object, instantiation and constructors, simple vs. composed classes, program flow*. The categorization of episodes was validated by two independent teachers.

The presentation of the findings was structured as follows. For each category there were four paragraphs:

- Introduction: A description of the concepts included in the category, an overview of previous studies, and the details of the teaching principles.
- Findings: A qualitative analysis of the difficulties and erroneous perceptions, divided into sub-categories, a quantitative analysis (where available), examples for improving the pedagogy, and an analysis of attitudes at the end of the course.
- Discussion: A summary table of all the categories, sub-categories, and conceptions, a comparison of the findings with findings of other studies, and a discussion of the effect of using BlueJ on learning these concepts.
- Recommendations and teaching dilemmas.

For a subcategory, each difficulty or erroneous conception is presented by a definition and a short description of the problem, followed by a table of episodes. An episode might be taken from a discussion in the class, an answer to a questionnaire, or a solution of a homework exercise. An episode is described by: (i) the situation in which it occurred, (ii) a description of what had taken place, (iii) when it occurred, and (iv) a code for the name of the student involved. Since the episodes come from a large variety of examples and exercises, we used a running example to demonstrate each problem; this enables the reader of the research presentation to understand the description of the difficulties and misconceptions without detailed knowledge of the wealth of material used in the course.

5.2. Categories of Student Understanding of OOP Concepts

This section presents the full list of the categories and sub-categories that we found (Ragonis, 2004). For each sub-category we also give the list of difficulties and misconceptions comprising it. Although space prevents a full discussion of each one, we believe that instructors with experience teaching OOP will be able to relate to them.

Category 1 — Object vs. Class

It is essential that students understand the relationship between a class and objects of the class. A class is a template from which objects can be created, but to find the fields and methods of an object you must look at the text of the class.

Subcategory 1.1: The nature of a class as a template.

1.1.1 Difficulties in understanding the static aspect of the class definition.
1.1.2 Difficulties in understanding that a method can be invoked on any object of the class.
1.1.3 Misconception: You can define a method that doesn't access any attribute.
1.1.4 Misconception: You can define a method that adds an attribute to the class.
1.1.5 Difficulties in understanding the classification of methods that we used (constructors, mutators, accessors, "others").
1.1.6 Misconception: You can invoke a method on an object only once.

Subcategory 1.2: Connections between objects and classes.

1.2.1 Misconception: A class is a collection of objects, rather than a template for creating objects.
1.2.2 Misconception: You can define a non-constructor method to create a new object.
1.2.3 Misconception: You can define a method that replaces the object itself.
1.2.4 Misconception: You can define a method that destroys the object itself.
1.2.5 Misconception: You can define a method that divides the object into two different objects.

Subcategory 1.3: Object creation.

1.3.1 Difficulties in understanding the process of creating an object.[2]

Subcategory 1.4: Identification of objects.

1.4.1 Misconception: Two objects of the same class cannot have equal values for their attributes.
1.4.2 Misconception: Two objects can have the same identifier if there is any difference in the values of their attributes.

1.4.3 Misconception: An attribute value can be used as the object identifier.
1.4.4 Misconception: The object identifier is one of the object's attributes.
1.4.5 Difficulties recognizing an object due to multiple representations (the set of values of attributes, the BlueJ icon and the graphical rendering).

Category 2—Instantiation and Constructors
Instantiation is the process by which an object is created from a class: memory allocation and execution of the constructor. Students must understand these both conceptually and in terms of the constructs in the programming language.

Subcategory 2.1: General understanding of instantiation.

2.1.1 Misconception: There is no need to invoke the constructor method, because its definition is sufficient for object creation.
2.1.2 Misconception: Constructors can include only assignment statements to initialize attributes.
2.1.3 Misconception: Instantiation involves only the execution of the constructor method body, not the allocation of memory.
2.1.4 Misconception: Invocation of the constructor method can replace its definition.

Subcategory 2.2: Understanding instantiation in a composed class.

2.2.1 Misconception: If objects of the simple class already exist, there is no need to create the object of the composed class.
2.2.2 Misconception: Creation of an object of a composed class automatically creates objects of the simple class that appear as attributes of the composed class.
2.2.3 Difficulties in understanding where objects of the simple class are created, before the creation of the object of the composed class.

Subcategory 2.3: Understanding instantiation is affected by its implementation in the programming language.

2.3.1 Difficulties understanding the empty constructor.
2.3.2 Difficulties understanding objects if their attributes are not explicitly initialized.
2.3.3 Initializing an attribute with a constant as part of its declaration causes confusion in distinguishing between a class and an object.
2.3.4 Initializing an attribute with a constant within the constructor declaration causes confusion in distinguishing between a class and an object.
2.3.5 Misconception: If the attributes are initialized in the class declaration there is no need to create objects.

Category 3—Simple vs. Composed Classes

Students showed difficulties in understanding a core concept of OOP, defining classes that contain attributes of other classes.

Subcategory 3.1: Understanding encapsulation.

3.1.1 Misconception: An object cannot be the value of an attribute.

3.1.2 Misconception: The attributes of the composed class include all the attributes of the objects of a simple class, *instead of* the objects themselves.

3.1.3 Misconception: The attributes of the composed class include all the attributes of the objects of a simple class, *in addition to* the objects themselves.

3.1.4 Misconception: Methods that are declared in the simple class have to be declared again in the composed class for each of the simple objects.

3.1.5 Misconception: There is no need for mutators and accessors for attributes that are of the simple class within the composed class.

3.1.6 Misconception: To change the value of an attribute of an object of a simple class that is the value of an attribute in an object of a composed class, you need to construct a new object.

3.1.7 Misconception: Methods can only be invoked on objects of the composed class, not on objects of the simple class defined as values in its attributes.

Subcategory 3.2: Understanding modularity.

3.2.1 Misconception: After a composed class is defined, new methods cannot be defined in the simple class.

3.2.2 Methods from the simple class are not used; instead, new equivalent methods are defined and duplicated in the composed class.

3.2.3 Methods in different classes are not distinguished if they have the same signature.

Subcategory 3.3: The class as a collection of objects.

3.3.1 Misconception: Objects of a simple class, used as values of the attributes of a composed class, have to be identical.

3.3.2 Misconception: In a composed class you can develop a method that adds an attribute of a simple class to the composed class.

3.3.3 Misconception: In a composed class you can develop a method that removes an attribute of a simple class from the composed class.

Subcategory 3.4: Understanding information hiding.

3.4.1 Misconception: Attributes of the simple class must be directly accessed from the composed class instead of through an interface.

Subcategory 3.5: Personification.

3.5.1 Misconception: Attributes in a simple class are automatically replicated in the composed class by transferring its meaning.

Subcategory 3.6: Understanding invocation on the implicit object.

3.6.1 Misconception: A method must always be invoked on an explicit object. (We did not teach the use of an explicit "this," so the students invented an imaginary object upon which to invoke the method.)

Category 4—Program Flow

Students found it hard to create a general picture of the execution of a program that solves a problem. We found students asking questions of the form: What actions are carried out? When are they carried out? What triggers the action? What is the order of execution of actions?

Subcategory 4.1: Understanding executions of methods.

4.1.1 Misconception: Methods are executed according to their order in the class definition.
4.1.2 Misconception: Every method can be invoked only once.[3]
4.1.3 Difficulties distinguishing when there is a need to explicitly write the identifier of the object.
4.1.4 Difficulties understanding the influence of method execution on the object state.
4.1.5 Difficulties understanding the invocation of a method from another method.

Subcategory 4.2: Understanding data flow.

4.2.1 Where do the values of the parameters come from?
4.2.2 To where does the return value of a method go?

Subcategory 4.3: Things happen with no cause.

4.3.1 Misconception: Objects are created by themselves.
4.3.2 Misconception: Attributes values are updated automatically according to a logical context.
4.3.3 Misconception: The system does not allow unreasonable operations.

Subcategory 4.4: "How does the computer know?"

4.4.1 How does the computer know what the class attributes and methods are?
4.4.2 How does one class recognize another?

Subcategory 4.5: Difficulties in understanding the overall flow of execution: What happens and when?

5.3. An Example Problem and its Table of Episodes

Since the entire set of results is too long for an article, we give one example taken from category 3 to show the form and content of a typical result and the type of episodes from which it was derived.

Category 3—Simple vs. Composed Classes
Students showed difficulties in understanding the definition of classes that contain attributes of other classes.

Subcategory 3.3: The class as a collection of objects.
A composed class may be perceived as a collection of objects of the type of a simple class, rather than as a *class* that just happens to have some of its fields of the type.[4]

3.3.2 Misconception: Students conclude that a method can be defined to add an attribute of the simple class to the composed class.

This subcategory and misconception are presented first using our running example and then giving authentic episodes from which the result was deduced (Figure 1). The running example was a class *Disc* which had three attributes *song1*, *song2* and *song3* of type *Song*. Each episode consists of an example or exercise given in class or as homework, the answer given by the student (together with an identification of its source) and followed by our interpretation of the answer as demonstrating the problem or misconception.

5.4. Recommendations for Teaching

The research conclusions suggested guidelines for teaching and a recommended syllabus for teaching OOP to novices. This section presents some of these guidelines:

- Although constructors are difficult to understand, it is better to "bite the bullet" and teach them relatively early. Constructors are at the heart of OOP and are executed in any case, so ignoring them will cause more difficulties than it will solve. We found it preferable to use a full constructor—assigning initial values to each attribute—rather than a simpler constructor based upon default or constant values (Ragonis & Ben-Ari, 2002). Teachers must explicitly explain that there are two steps in the instantiation of an object: allocating memory and then invoking the body of the constructor.
- Objects are a complex concept: they are created, they have an identity and methods can be invoked on them. It is important to relate objects to their

Running example
In the class *Disc*, define a method *addSong(_song)* which adds a song to the collection of songs on the disc.
Episodes
Question from a homework exercise: Describe a simple class and a composed class that uses it. The definition must include attributes and actions for each class. [This exercise was given in the initial stages of learning OOP, before the programming language was used.] **An answer:** The student defined simple classes *Cupboard*, *Mirror*, *Bed*, and a composed class *Bedroom*, whose attributes were *cupboard* of class *Cupboard*, *mirror* of class *Mirror*, and *bed* of class *Bed*, and whose actions included *addMirror(_mirror2)*. Source: Student AB2, Exercise from Appendix 2.2 p. 17, question 2. **Our interpretation:** The student is confused because adding a piece of furniture to a room is possible in real life, but cannot be done in OOP dynamically by adding an attribute to a class.
Question posed to the students in class: Define a significant action in the class *Tower*. **An answer:** *void addAnOtherDie( )* *{ Die d4; }* Source: Student Q2, Exercise with classes *Cube* and *Tower*, Exercise from appendix 2.16. **Our interpretation:** The student's intention was to dynamically add an attribute to a class, but the semantics in Java is to declare a local variable.

Figure 1. A misconception and its episodes.

identities and states all the time, paying particular attention to changes in state during execution.

- A composed class must include attributes of different types to avoid the misconception that a composed class is just a collection of identical objects, and also to emphasize the need to create objects of the composed class, even though the simple objects have already been created.
- The instructor must integrate the teaching of the concept of program flow into the classical OOP concepts of encapsulation, modularity and data hiding.

- Problem solving must also be emphasized in order to create an appropriate context for OOP problems, rather than focusing just on entities in the target system.
- An appropriate place (relatively early in the syllabus) must be found for teaching the main method. We believe that this would help in understanding concepts such as object creation and identification, the need for mutators and accessors, and understanding a self invocation. Furthermore, it will enable students to first understand encapsulation in a simple class before encountering composed classes. Study of the main method will also help students internalize issues of control and data flow.

5.5. Contributions of the Research

This research has made two primary methodological contributions:

First, it is unprecedented in the length and breadth of its investigation into the concepts of novices studying OOP. The four main categories were divided into eighteen sub-categories, and include 58 conceptions and difficulties. This detail of analysis, together with an integrated view, gives educators a source of data which can inform the development of syllabi for OOP and engender a much better understanding of students' difficulties by their teachers. Once a teacher is aware of the potential difficulties, she can plan her teaching approach accordingly.

The second contribution is the use of the constructivist qualitative research methodology. The research employed a wide variety of tools that gave a rich picture of students' understanding and the difficulties they encountered over the long period of the teaching-learning process. Each issue could be checked from different points of view and in a range of situations. It also ensured the validity of the research results.

The categories and sub-categories were not just a background to a narrative description; rather, the detailed classification was supported with authentic episodes which can be verified. Similar episodes are likely to appear in the teaching of other instructors.

The use of a running example is a novel way of documenting research results. This enables the reader to study the details of the various conceptions and difficulties without needing to absorb the entire set of exercises the students carried out.

6. CONCLUSION

This article presents an overview of a research project on teaching OOP to novices. The emphasis is on the theoretical background, the research methodology, the analysis technique and the mode of presentation of the findings.

From an examination of students' understanding at the end of the teaching process, we found that important principles of OOP such as encapsulation, modularity and data hiding were understood. In the summary questionnaire, all the students correctly explained the objective of instantiation, explained the process involved in its activation and also implemented the creation of a new composed

object in Java. They also demonstrated an almost perfect capability to classify methods to the appropriate (simple, composed or main) class. Their explanations used the appropriate OOP terms. Students also demonstrated an understanding of program flow through description, analysis and expansion of the main method defined in the project, including a detailed description of the process scenario that results from executing the main method. In a stand points questionnaire filled out before the final project, we saw that they no longer considered most of the concepts to be difficult, although three concepts remained so: composed classes, the main program, and mutator and accessor methods. The success of the final projects as measured by criteria of OOP principles was very high.

This study showed that it is possible to teach OOP to high school novices. The success of the students in planning and implementing a final project (as well as other findings) confirms this claim. We learned that the order in which concepts are presented is extremely important. The difficulties described do not mean that teaching OOP is inappropriate for novices. Most of them appeared with low frequency and characterized a particular period of learning, disappearing as the course progressed. Teachers, as well as developers of learning materials, should be aware of the large number of conceptions and difficulties that were found, so that they can improve the process of teaching and learning OOP.

ACKNOWLEDGEMENT

We wish to express our gratitude to the Gan-Nahum Gymnasia high school in Rishon leZion that enabled us to carry out this extensive research project, and in particular to the students who cooperated with our ever-changing ideas on how to teach OOP. We also thank the reviewers and editors for their helpful comments and suggestions.

NOTES

1. The first author also taught similar material to students at a teachers' college, but the analyses of the results from these classes were not directly used in this research.
2. This subcategory is further expanded in Category 2.
3. This appeared in item 1.1.6 in the context of a class declaration, while here it appears in the context of program flow.
4. We were careful to define a class as a type (template) from which objects can be created and not as a collection of (existing) objects.

NOTES ON CONTRIBUTORS

Noa Ragonis recently received her Ph.D. degree in science teaching from the Weizmann Institute of Science. She has taught computer science for twenty years, first in high schools and now at the Beit Berl College, where she is currently head of the computer science track at the department of education. She has written three high-school computer science textbooks (in Hebrew), including a pioneering course on expert systems.

Mordechai Ben-Ari is an associate professor in the Department of Science Teaching of the Weizmann Institute of Science, where he heads a group that develops courses in computer science for high school students. He holds a Ph.D. in mathematics and computer science from the Tel Aviv University. In 2004, he received the ACM/SIGCSE Award for Outstanding Contributions to Computer Science Education. He is the author of six textbooks on concurrent computation, programming languages and mathematical logic, most recently ''Mathematical Logic for Computer Science (Second Edition)'' published by Springer-Verlag London. His research interests include the use of visualization in teaching computer science, the pedagogy of concurrent and distributed computation, and the application of theories of education to computer science education.

REFERENCES

ACM/IEEE (2001). Computing curricula 2001. *Journal on Educational Resources in Computing, 1*(3), article No.1.

Allen, E., Cartwright, R., & Stoler, B. (2001). DrJava: A lightweight pedagogic environment for Java. *ACM SIGCSE Bulletin, 34*(1), 137–141.

Barnes, D.J., & Kölling, M. (2003). *Objects First with Java - A Practical Introduction using BlueJ.* New Jersey: Pearson Education.

Ben-Ari, M., Myller, N., Sutinen, E., & Tarhio, J. (2002). Perspectives on program animation with Jeliot. In S. Diehl (Ed.), *Software Visualization.* Lecture Notes in Computer Science *2269*, 31–45.

Berge, O., Fjuk, A., Groven, A., Hwgna, H., & Kaasbøll, J. (2003). Comprehensive object-oriented learning–An introduction. *Computer Science Education, 13*(4), 331–335.

Bergin, J. (1999). Why procedural is the wrong first paradigm if OOP is the goal. *Presented at OOPSLA99 Educator's Symposium.* Retrieved October 30 2004 from http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html

Bergin, J. (2000). Teaching objects with elementary patterns. Retrieved October 30 2004 from http://csis.pace.edu/~bergin/patterns/Whynotproceduralfirst.html

Biddle, R., & Tempro, E. (1998). Java pitfalls for beginners. *ACM SIGCSE Bulletin, 30*(2), 48–52.

Bishop, J.M. (1997). A philosophy of teaching Java as a first teaching language. *ACM SIGCSE Bulletin, 29*(1), 140–142.

Bishop, J.M., & Bishop, N. (2000). Object-orientation in Java for scientific programmers. *ACM SIGCSE Bulletin, 32*(1), 357–361.

Börstler, J., Johansson, T., & Nordstrom, M. (2002). Teaching OO concepts–A case study using CRC-Cards and BlueJ. *Proceedings of ASEE/IEEE Frontiers in Education Conference, FIE2002, T2G1-6,* Boston, MA.

Brilliant, S.S., & Wiseman, T.R. (1996). The first programming paradigm and language dilemma. *ACM SIGCSE Bulletin, 28*(1), 338–342.

Bucci, P., Heym, W., Long, T.J., & Weide, B.W. (2002). Algorithms and object-oriented programming: Bridging the gap. *ACM SIGCSE Bulletin, 34*(1), 302–306.

Buck, D., & Stucki, D.J. (2000). Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. *ACM SIGCSE Bulletin, 32*(1), 75–79.

Détienne, F. (2001). *Software Design–Cognitive Aspects.* London: Springer.

Ely, M., Vinz, R., Downing, M., & Anzul, M. (1997). *On Writing Qualitative Research: Living by Words.* London: Falmer.

Fernández, L., & Peña, M.R. (2002). PIIPOO: An adaptive language to learn OO Programming. *Presented at Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts ECOOP2002,* Málaga, Spain. Retrieved October 30 2004 http://prog.vub.ac.be/ecoop2002/ws03/acc_papers/L_Fernandez_Munoz.pdf

Fleury, A.E. (2000). Programming in Java: Student-constructed rules. *SIGCSE Bulletin, 32*(1), 197 – 201.

Gal-Ezer, J., Beeri, C., Harel, D., & Yehudai, A. (1995). A high-school program in computer science. *IEEE Computer, 28*(10), 73 – 80.

Gal-Ezer, J., & Harel, D. (1999). Curriculum and course syllabi for a high-school CS program. *Computer Science Education, 9*(2), 114 – 147.

Hadjerrouit, S. (1998a). Java as first programming language: A critical evaluation. *ACM SIGCSE Bulletin, 30*(2), 43 – 47.

Hadjerrouit, S. (1998b). A constructivist framework for integrating the Java paradigm into the undergraduate curriculum. *ACM SIGCSE Bulletin, 30*(3), 105 – 107.

Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. *ACM SIGCSE Bulletin, 31*(3), 171 – 174.

Hammersley, M. (1995). *The politics of social research.* Thousand Oaks, CA: Sage.

Hazan, H. (2001). The other voice: On the qualitative research sound. In N. Sabar (Ed.), *Qualitative Research: Genres and Traditions in Qualitative Research* (pp. 9 – 12), Tel-Aviv, Israel: Zmora Bitan (in Hebrew).

Hendrix, T.D., Cross, J.H., & Barowski, L.A. (2004). An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. *ACM SIGCSE Bulletin, 36*(1), 387 – 391.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *ACM SIGCSE Bulletin, 29*(1), 131 – 134.

King, K.N. (1997). The case for Java as a first language. *Proceedings of the Annual ACM Southeast Conference,* Murfreesboro, Tenn., 124 – 131.

Kölling, M. (1999a). The problem of teaching object-oriented programming, Part I: Languages. *Journal of Object-oriented Programming, 11*(8), 8 – 15.

Kölling, M. (1999b). The problem of teaching object-oriented programming, Part II: Environments. *Journal of Object-oriented Programming, 11*(9), 6 – 12.

Kölling, M., & Rosenberg, J. (1996a). An object-oriented program development environment for the first programming course. *ACM SIGCSE Bulletin, 28*(1), 83 – 87.

Kölling, M., & Rosenberg, J. (1996b). Blue - a language for teaching object-oriented programming. *ACM SIGCSE Bulletin, 28*(1), 190 – 194.

Kölling, M., & Rosenberg, J. (2000). Objects first with Java and BlueJ (seminar session). *ACM SIGCSE Bulletin, 32*(1), 429.

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. *ACM SIGCSE Bulletin, 33*(3), 33 – 36.

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education, 13*(4), 249 – 268.

Lattu, M., Tarhio, J., & Meisalo, V. (2000). How a visualization tool can be used: Evaluating a tool in research and development project. *12th Workshop of Psychology Programming Interest Group,* Corenza, Italy, 19 – 32. Retrieved October 30, 2004, from http://www.ppig.org/papers/12th-lattu.pdf

Lewis, J. (2000). Myths about object-orientation and its pedagogy. *ACM SIGCSE Bulletin, 32*(1), 245 – 249.

Maheshwari, P. (1997). Teaching programming paradigms and languages for qualitative learning. *Proceedings of the Second Australasian Conference on Computer Science Education (ACSE),* Melbourne, Australia, 32 – 39.

Mazaitis, D. (1993). The object-oriented paradigm in the undergraduate curriculum: A survey of implementations and issues. *ACM SIGCSE Bulletin, 25*(3), 58 – 64.

Mitchell, W. (2001). A paradigm shift to OOP has occurred. . . implementation to follow. *Journal of Computing in Small Colleges (JCSE)*, *16*(2), 95 – 105.

Neubauer, B.J., & Strong, D.D. (2002). The object-oriented paradigm: More natural or less familiar? *Journal of Computing in Small Colleges (JCSE)*, *18*(1), 280 – 289.

Nourie, D. (2002). Teaching Java technology with BlueJ. *Technical Articles*, Online article at java.sun.com. Retrieved October 30, 2004, from http://java.sun.com/features/2002/07/bluej.html

Osborne, M. (1992). The rule of object-oriented technology in the undergraduate computer science curriculum – educators' symposium. *Addendum to the Proceedings of OOPSLA'92, Vancouver, British Columbia, Canada, 4*(2), 303 – 308.

Parlante, N. (1997). Teaching with object-orientation libraries. *ACM SIGCSE Bulletin, 29*(1), 140 – 144.

Patterson, A., Kölling, M., & Rosenberg, J. (2003). Introducing unit testing with BlueJ. *ACM SIGCSE Bulletin, 35*(3), 11 – 15.

Phillips, D.C. (1990). Subjectivity and objectivity: An objective inquiry. In E.W. Eisner & A. Peshkin (Eds.), *Qualitative inquiry in education: The continuing debate* (pp. 19 – 37). New York: Teachers College Press.

Ragonis, N. (2004). *Teaching Object Oriented Programming to Novices.* Ph.D. thesis, Weizmann Institute of Science, Rehovot, Israel (in Hebrew).

Ragonis, N., & Ben-Ari, M. (2002). Teaching constructors: A difficult multiple choice. *Presented at Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts ECOOP2002,* Málaga, Spain. Retrieved October 30 2004 from http://prog.vub.ac.be/ecoop2002/ws03/acc_papers/Noa_Ragonis.pdf

Reid, R.J. (1993). The object-oriented paradigm in CS1. *ACM SIGCSE Bulletin, 25*(1), 265 – 269.

Rosson, M.B., & Alpert, S.R. (1990). The cognitive consequences of object-oriented design. *Journal of Human-Computer Interaction, 5*(4), 345 – 379.

Sabar Ben-Yehoshua, N. (2001). The history of qualitative research – influences and directions. In N. Sabar (Ed.), *Qualitative Research: Genres and Traditions in Qualitative Research* (pp. 13 – 16). Tel-Aviv, Israel: Zmora Bitan (in Hebrew).

Schoenefeld, D.A. (1997). Object-oriented design and programming: an Eiffel, C+ , and Java for C programmers. *ACM SIGCSE Bulletin, 29*(1), 135 – 139.

Schulte, C., & Niere, J. (2002). Thinking in object structures: Teaching modeling in secondary school. *Presented at Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts ECOOP2002,* Malaga, Spain. Retrieved October 30 2004 from http://prog.vub.ac.be/ecoop2002/ws03/acc_papers/Joerg_Niere.pdf

Shkedi, A. (2003). *Words That are Trying to Touch: Qualitative Research Theory and Practice.* Tel-Aviv: Ramot (in Hebrew).

Stein, L.A. (1997). Beyond objects. *Educators Symposium, Conference on OOP Systems, Languages, and Applications*, Atlanta, Georgia. Retrieved October 30 2004 from http://www.ai.mit.edu/projects/cs101/beyond-objects.ps

The College Board (2004). *The Advanced Placement Program*. Retrieved October 30 2004 from http://www.collegeboard.com/ap/students/compsci/java_subsetA.html or http://www.collegeboard.com/student/testing/ap/sub_compscia.html

Thramboulidis, K.C. (2003). Sequence of assignments to teach object-oriented programming: A constructivism design-first approach. *Informatics in Education, 2*((1), 103 – 122.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM, 40*(5), 80 – 91.

Woodman, M., & Holland, S. (1996). From software user to software author: An initial pedagogy for introductory object-oriented computing. *ACM SIGCSE Bulletin, 28*(SI), 60 – 62.

Woodman, M., Davies, G., & Holland, S. (1996). The joy of software - starting with objects. *ACM SIGCSE Bulletin, 28*(1), 88 – 92.