Cognitive Activities of Abstraction in Object Orientation: An Empirical Study

Rachel Or-Bach and **Ilana Lavy**
Emek Yezreel College
Emek Yezreel 19300, Israel
orbach@yvc.ac.il, ilanal@yvc.ac.il

Abstract

Alongside the widespread support for adopting object orientation there are reports on difficulties in learning object oriented programming and design. This indicates the need for refining the research on cognitive difficulties in a way that will offer guidelines for better designing respective education. The presented findings of our study relate to general issues of object-oriented design and in particular to the abstraction issue with its various manifestations. Based on students' solutions we extracted a cognitive task analysis taxonomy regarding abstraction and inheritance. We discuss possible implications of our results for the teaching of object orientation and for further needed research.

Keywords: Object-oriented design, object-oriented programming, abstraction, inheritance, task analysis

1. Introduction

Object orientation is the current widely advocated programming paradigm and in recent years the use of an object-oriented approach in early stages of computing curricula has risen dramatically. Programming is not an isolated technical skill; it is a complex cognitive activity associated with a programming paradigm. The programming paradigm guides the way of thinking about problem solving. McLaughlin [6], in a paper titled "Oh, by the way, Java is object-oriented…", criticizes the switch to Java as the first programming language without capitalizing on the object-oriented features of Java. Educators consider object orientation as a paradigm that provides a stronger foundation for software design. As Lewis [4] stated, "Object orientation has been embraced as a worthy approach in part because it hands us the tools that we've needed all along to create effective software designs" … "An object is the ultimate Abstract Data Type, encapsulating a set of operations and the data that they manage".

However, alongside the widespread support for adopting object orientation there are reports on difficulties in learning object oriented programming and design [13], reports on problems encountered when programmers move to object–orientation [10], and critiques on the way object-orientation is taught [5]. These findings show that there is a need for refinement of the relevant research on object-oriented programming and design. Such research should explore the various cognitive difficulties of object-oriented programming, in a way that will offer guidelines for designing respective educational materials. The research should reveal implications regarding the order of topics, supporting tools, the way to integrate design and programming, the choice of first programming language/paradigm to be taught and more.

In this paper we describe a study that we have conducted to investigate the understanding of fundamental concepts in object-oriented design. The next section describes the audience, the task given to the audience and some characteristics of the expected solution. The following section (3) presents some analysis of our findings and is composed of two parts. The first part describes general issues of the students' designs, mostly related to problem decomposition. The second part suggests a cognitive taxonomy based on cognitive task analysis with regard to abstraction and inheritance. The last section of the paper discusses possible implications of our results for the teaching of object orientation and for further needed research.

2. A Short Description of the Study

Thirty-three college students from the Computer Science and Information Systems department participated in this study. The group consisted of third year students who had two courses on object oriented programming and design: one year course on Object Oriented Programming (OOP) with implementation in JAVA, and a one semester course on Software Engineering with emphasis on Object Oriented Analysis and Design (OOAD). We prepared one question which involves a solution that can demonstrate understanding of fundamental OO concepts, such as classes, inheritance, and polymorphism. The question was as follows:

> *In a high-tech company there are two kinds of employees. Permanent employees get a monthly salary and temporary employees are paid by the hour. The*

permanent employees are rewarded for extra hours. Each employee who works at the company over three years gets a present for the holidays. Cumulative information on costs related to salaries and presents for eligible employees is managed and processed by the human resources department.

a. *Describe schematically the relevant classes and the connections between them.*
b. *Include in your schematic design details of the classes' components using object oriented programming principles. The classes' components should include: class name, class attributes and class methods (with explanation about their purpose).*
c. *Extract and outline from your schematic description the concepts/principles of object-oriented programming (such as inheritance, polymorphism etc.) that you have employed. Describe explicitly how each of these concepts/principles is exhibited in your design.*

It should be noted that the task does not include any requirements for a specific formal notation for the schematic design. The reason for this is that we wanted students to concentrate on the task without worrying how well they remember and know to implement the notation they had learned some time before. The third paragraph of the task directs the students to employ object-oriented principles, and provides an additional opportunity for them to express their object-oriented design characteristics in a way that is convenient for them.

An expected correct answer should have several characteristics as follows. Relevant classes are presented in a schematic hierarchy, while each class includes a list of relevant attributes and a list of relevant methods. The set of classes should include an abstract class for employee and two sub-classes: for temporary employee (hourly paid) and for monthly employee. An additional class of human resources should also be included. The abstract class for employee should include the following attributes: name, starting date of work and optionally additional relevant ones. The abstract class should include also: a method for calculating the length of employment; a method for deciding the eligibility for a holiday present; and an abstract method for calculating the salary. Inheritance should be demonstrated by attributes and methods appearing in the abstract class.

3. Findings

3.1 General Issues in the Object-Oriented Design

Most of the students (thirty one out of thirty three) constructed the hierarchy with the abstract class of an employee and the two respective sub-classes. Six students did not include a class of human resources to deal with the salaries. Besides these classes for the employees and human resources, that are necessary for the solution, several students included additional classes. The additional classes were of two types: classes that were irrelevant to the solution, such as a class for the whole company; and classes that actually could have been integrated as methods or attributes of existing classes. Three students added classes for special procedures such as calculating the eligibility for a present. One student added an abstract class for employee's benefits, with two subclasses: one for benefits of a temporary employee, and the other for benefits of the monthly employee. This decomposition task, the decision about the relevant classes to be included in a design is considered to be a difficult one. In a study of cognitive activities in object-oriented development [13], problem decomposition was perceived as the activity that caused the most difficulties related to object oriented techniques. As our problem was relatively simple, problem decomposition was not the major difficulty, but still a few students had difficulties to decide whether some real world element, such as a holiday present, deserves a class.

Within the students that included a human resources class, ten students included in this class methods for calculating an employee salary. In these cases the design looks like a procedural programming design, where the procedure is separated from the data it acts upon.

3.2 Abstraction, Inheritance and Polymorphism

Abstraction is a fundamental concept in programming in general and in object-oriented programming in particular. We have tried to categorize the various abstraction cognitive activities that students employed in their solution for the given problem.

Defining a class with its relevant attributes and methods is the basic required abstraction. Most of the students did it but few students still added some irrelevant attributes, such as phone number and address. Using a hierarchy of classes with an abstract class is a more complex abstraction task. We were interested especially in this kind of abstraction and the inheritance that it enables.

Within this context, analysis of students' responses reveals three categories of abstraction. These categories define a taxonomy for this task analysis (Figure 1). The first category refers to the inclusion of attributes in the abstract class that was defined (class employee). The second category refers to the inclusion of attributes and implemented methods in this abstract class. Relevant implemented methods were methods for calculating length of employment and for calculating eligibility for a holiday present. The third category refers to solutions in which the abstract class includes attributes, implemented methods and abstract methods. The relevant abstract method is for salary calculation. Figure 1 depicts the three levels of the taxonomy. The numbers in brackets indicate the number of students associated with each level.

Four students handed the design we expected, with the abstract class of employee including an abstract method for

salary calculation, and of course also fulfilling the lower levels of the taxonomy. As can be seen by the number of students that did it, it was the most difficult task. The design of seven students included attributes and implemented methods in the abstract class of employee, but did not include the required abstract method. The designs of twenty students included only attributes for the abstract class of employee with no methods at all. Several of them included length of employment as an attribute, rather than as a method. These proportions strengthen our hypothesis about the taxonomy we suggested for this task analysis.

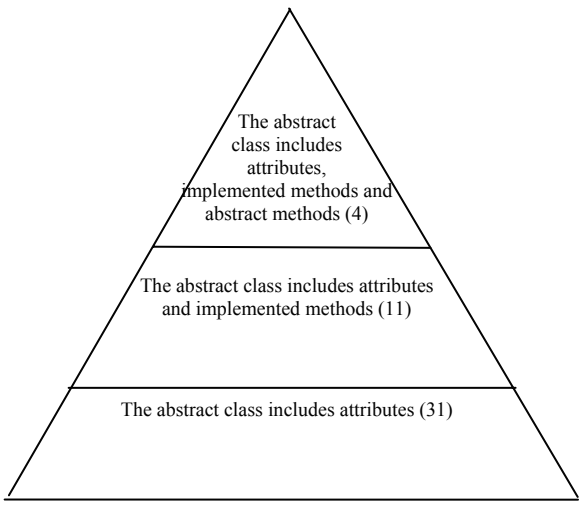

Figure1. The taxonomy of the task analysis

The results can be explained by the emphasis in learning object orientation on structure rather than on process. Several students even completely ignored the salary calculation issue. They were so concentrated on the structural elements of the design, that they forgot the expected solution that should involve some calculation.

Polymorphism is actually an abstraction manifestation. This property supports greater abstraction in an object's message interface. Students' use of an abstract method for salary calculation potentially enables polymorphism, even though not all of the students indicated it explicitly as was asked in the given question. Two students described explicitly polymorphism with regard to messages sent by the human resources class for managing the cumulative data regarding salaries.

4. Conclusions and Discussion

The first part of this section discusses the rational for the methodology of our study that enabled the finer granularity of our task analysis. The second part presents and discusses the main findings of our study - the taxonomy described previously regarding abstraction and inheritance. The third part deals with implications of our findings for instructional design. The last part describes our plans for further research.

In planning the study, we found it important to emphasize the design aspect and to have students actually solve a problem and not just report on their perception of the difficulties with object orientation as was done in other studies [7, 10, 13]. We phrased the task with explicit hints to the principles we expect to see in the solution and we let the students use both informal diagrams and free text to express the object-oriented principles they employ in their designs. The fact that we used actual designs instead of pre-prepared questionnaires enabled the identification of specific cognitive difficulties regarding abstract methods.

From the results of our study it seems that the major cited advantages of object orientation are exactly the same issues that make object orientation so difficult for students, and probably widens and makes clearer the difference between good programmers and those that are not. Object oriented programming involves object-oriented design, modeling capabilities, identification of relations between the world and the program objects, along with abstraction capabilities. These are always considered higher order cognitive skills. Unfolding the task of "abstraction" revealed different levels of it. The taxonomy we suggested defines three main levels incorporating the majority of students' solutions with regard to abstraction and inheritance. The analysis of students' designs shows that the majority of students did not reach the desirable level of abstraction. The taxonomy we extracted from the students solutions is not so much surprising in its categories but in the large number of solutions (20) that include only attributes in the abstract class with no methods of any type.

We expected difficulties with the abstract method for salary calculation, but not with implemented methods such as the method for length of employment and a method for checking the eligibility for a holiday present. Similar findings about students' use of methods and attributes with regard to abstraction were found also by other researchers. Detienne [2] claims that one of the main difficulties experienced by novices is the articulation between declarative and procedural aspects of the solution. Some novices do not succeed in decomposing the large procedure into smaller functional units and they associate the procedure as a whole to a single class. We found this in several solutions that resemble a procedural solution rather than an object-oriented one. In the same survey, Detienne [2] cites her previous studies showing that while novices have difficulties in using the inheritance property of static characteristics, they have even more difficulties in using the inheritance of functionality. A research by Pennington et al. [9] provides empirical descriptions of design activities and of the evolving designs for: expert procedural designers, expert OO designers and novice OO designers. The findings regarding the novice OO are of particular importance for our study. Novice OO designers spent large proportions of their time creating and abandoning entities

in the process of defining classes. Also novices do not begin considering procedure categories of design until much later than experts. Novices did not even mention functionality of the system until the second third of their design sessions. They found that expert OO designers tended to define methods with associated classes, whereas novices defined classes first and then methods. These results are compatible with what we found with our novice object oriented programmers. Giving students opportunities and guidance for revising their designs might have given different results. In another study, with emphasis on analysis and design instead of programming, students also reported on more difficulty in understanding and modeling the behavioral aspects of OO analysis and design than the structural aspects [11]. An object is an abstract data type, but for object-oriented design it seems more appropriate to consider the abstraction inherent in object orientation as behavior, rather than data, abstraction. This seems to be difficult for student to conceptualize.

As a result of our findings we have some recommendations for respective instructional design. Educators stress the benefits of integrating examples in instructional materials. A critical component of this integration is the choice of examples. The taxonomy we found can serve as a basis for designing examples that support the understanding of the OO concepts students had difficulties with. The examples should sharpen the concept meaning and use. Examples of correct solutions should be provided both as an example for the use of a concept (e.g. abstract method) and as a non-example, which is a solution without the use of this concept. The presentation of examples and non-examples of a concept use might deepen the understanding of the concept. This presentation should be accompanied by a respective discussion with the students. Discussions in which students communicate, present and evaluate different approaches to solving complex problems can develop their sense of criticism towards quality of solutions. Consequently the capacity to reflect on solutions and to engage in a self-assessment is increased [12]. This approach can also incorporate Pennington et al. [9] suggestion that training of novices could include materials that explicitly address some of the differences between novice and expert behavior in object-oriented design. To emphasize abstraction in learning object orientation, we agree with Machanick [5] about the importance of presenting students with existing abstractions to use as building blocks before designing their own abstractions. But still the taxonomy for the task analysis of abstraction that we extracted from students' solution is very important for planning the order and emphasis for these learning materials. We also agree with computer science educators [1, 8] that OOAD should be introduced in the first programming course. When we presented our findings to one of the object-oriented-programming course lecturers, he said that if the students had to write also the code for the problem solution, it would have helped them to correct their preliminary design.

This issue reflects the tight relationship between programming and design that should be emphasized in object orientation. We assumed that the graphical design captures the understanding, but it might be that the actual code would have been different. In fact, the difference is an interesting research issue for its own sake. Even though students were asked explicitly to describe how the concepts are shown in their design, most of them did not give a verbal description in addition to the graphical design. Having additional evidence about students' intentions, by code or by intermediate designs, might have made students conceptions more clear for use in further analysis.

Our study can be extended in two main directions. One direction is by adding the task of implementing the design in Java and looking at the interplay between code and diagram as was mentioned in the previous paragraph. The other direction for extending the study is by following the design process and analyzing the students' evolving designs and not just their final product. The integration of respective interviews might be very helpful. These two directions can also be combined for following possible evolving relations between design and code.

References

[1] Bergen, J. Teaching Object-Oriented Analysis and Design in CS 1, http://csis.pace.edu/~bergin/papers/OOAD.html 1996.

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

[3] Holmboe, C. A cognitive framework for knowledge in informatics: The case of object-orientation. ITiCSE 1999.

[4] Lewis, J. Myth about Object Orientation and its pedagogy. *Journal of Computer Science Education*, 14 (3 & 4), (2001), 22-26.

[5] Machanick, P. The abstraction-first approach to data abstraction and algorithms. *Computers and Education*, 31, (1998).

[6] McLaughlin, P. "Oh, by the way, Java is object oriented…" http://www.ulst.ac.uk/cticomp/oh.html 1997.

[7] Milne, I. and Rowe, G. Difficulties in learning and teaching programming – Views of students and tutors. *Education and Information Technologies*, 7, 1 (2002), 55-66.

[8] Nguyen, M. and Wong, S. OOP in introductory CS: Better students through abstraction. Fifth workshop on pedagogies and tools for assimilating object-oriented concepts, in OOPSLA 2001.

[9] Pennington, N., Lee, A. Y. and Rehder, B. Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction*, 10 (1995) 171-226.

[10] Ross, J. M. and Zhang, H. Structured programmers learning Object-Oriented Programming. *SIGCHI Bulletin*, 29, 4 (1997).

[11] Sim, E.R. and Wright G. The difficulties of learning object-oriented analysis and design: An exploratory study. *The Journal of Computer Information Systems*, (Winter 2001/2002).
[12] Simon, M. Learning mathematics and learning to teach: learning cycles in mathematics teacher education, *Educational studies in mathematics*, 26, (1994), 71-94.
[13] Tegarden, D. P. and Sheetz, S. D. Cognitive activities in OO development. International Journal of Human-Computer Studies, 54, (2001), 779-798.