Informatics in Education, 2007, Vol. 6, No. 1, 67–80

An Iterative Methodology for Teaching Object Oriented Concepts

Irit HADAR
Department of Management Information Systems, University of Haifa
Carmel Mountain, Haifa, Israel
e-mail: hadari@mis.haifa.ac.il

Ethan HADAR
Architecture and Shared Technology, HP Software
19 Shabazi Street, POB 170, Yehud 56100 Israel
e-mail: ehadar@hp.com

Received: December 2006

Abstract. Abstract thinking is a vital skill when learning computer science. Object technology and the concepts it is based upon make this skill even more crucial. However, previous research works show that students in top universities as well as experienced practitioners in industry encounter difficulties in thinking in abstract terms while practicing object oriented development. In this paper we suggest an iterative teaching methodology for supporting students in learning object oriented concepts. The suggested methodology is based on familiarizing students with modeling languages and tools at the early stages of their learning and iterating between model and code. We theoretically examine the contribution of modeling languages, in particular UML, to abstract thinking and consequently to the understanding of object oriented concepts and present some observations acquired during a trial execution of this methodology in a university course.

Key words: teaching object oriented, abstraction, visual models.

1. Introduction

Abstract thinking is a vital skill when learning computer science (CS). Object technology makes this skill even more crucial: abstraction was identified as one of the eight main quarks of object oriented (OO) development (Armstrong, 2006). Moreover, abstraction, when referred to as "a mechanism that allows us to represent a complex reality in terms of a simplified model so that irrelevant details can be suppressed in order to enhance understanding" (Armstrong, 2006, p. 124), underlies other basic concepts of OO such as *class*, *object*, *encapsulation*, *inheritance* and *polymorphism*.

However, research works in CS education have shown that students, in general, have difficulty in practicing abstract thinking. They tend to remain at a low level of abstraction when solving problems, caught in the details while missing the bigger picture (Aharoni and Leron, 1997; Holmboe, 1999; Fleury, 2001; Bucci *et al.*, 2001; Berge *et al.*, 2003; Hazzan and Hadar, 2005). Some of the common phenomena derived from this difficulty

are students' tendency to focus immediately on the code rather than the design when writing or reading a computer program; their tendency to focus on a single function or module/class while neglecting the wider view of the system; and their tendency to avoid using abstract classes. Moreover, a recent study (Hadar, 2004; Hadar and Leron, preprint) examined the cognitive processes taking place in OO design (OOD) practiced by experienced software developers and identified similar phenomena of difficulties in abstract thinking. Although these phenomena, as observed in the research on practitioners, were somewhat less severe than in the case of students, they were still surprising by their firm existence and extensiveness. This may imply that developing abstract thinking skills is very challenging and difficult to achieve even after gaining education as well as practical experience.

Observing the difficulties in thinking abstractly exhibited by both students and practitioners, one may consider searching for a solution in changing the initial educational path. Doing so may raise the following question: How can we provide abstract thinking skills to novice students learning OO concepts?

In this paper we address this question and present an iterative methodology for teaching OO utilizing both abstract representation and detailed programming experience. We review the principles on which our teaching methodology is based (Sections 2 and 3); present the iterative teaching methodology including practical detailed examples (Section 4); explore some phenomena related to its execution (Section 5); and finally, conclude with some thoughts and questions for the future (Section 6).

2. Teaching Object Oriented Development

Many discussions have been conducted regarding the question whether to teach procedural programming first and only then OO programming (OOP), or to teach OOP from the very start. The latter is known as the "object first" approach. However, Berge *et al.* (2003) claim that even when teaching "object first", the emphasis is put on the programming elements instead of learning OO thinking as a tool for software development. They suggest to start teaching modeling and design first, to provide students with wider perspective of OO concepts beyond the technicalities of programming languages. Bennedsen and Caspersen (2004) suggest a teaching approach adding conceptual modeling to the CS1 course, calling this a model-first approach; their focus is on elaborating OO concepts using modeling, and translating the models to generic coding patterns.

While the model and design first approaches seem to emerge naturally from the problems presented, many studies in education show that the development of abstract concept understanding is only possible after gaining vast experience and firm understanding of the more concrete concepts (e.g., Leron, 1987; Dubinsky, 1991; Sfard, 1991; Aharoni and Leron, 1997). Based on these reports it seems that no "shortcuts" to achieve abstract thinking skills can exist.

Since both teaching approaches – abstract first and concrete first – have their relative disadvantages, we seek to find a methodology utilizing the advantages of both approaches

while avoiding, as much as possible, their disadvantages. Taking this into consideration we suggest a teaching methodology that will be based on the following principles:

1. Supporting students in thinking abstractly in the earliest stages of learning OO, with the aid of modeling languages and tools.

2. Applying an iterative teaching methodology constantly cycling between model and code, namely high and low abstraction levels, to build a multi-aspect understanding of the basic concepts of OO.

In the following sections we explain the nature and expected contribution of these principles to the learning processes.

3. The Contribution of a Visual Modeling Language to Abstract Thinking

Modeling languages used for software design have several properties that potentially make them useful and practical tools for supporting abstract thinking. In particular, we examine the two following properties: abstraction barriers and visual representation.

Our demonstration utilizes the Unified Modeling Language (UML). UML has attained the status of a de facto modeling language standard (Booch, 1999; Kobryn, 1999) and is well known and widely used within the community of OO software development both in industry and academia (Kobryn, 1999; Leroux and Exton, 2001). More specifically to our context, UML was found to support the understanding of the "bigger picture", therefore, being an efficient means for understanding an existing software system (Hadar and Hazzan, 2004); verifying software architecture's fidelity and completeness (Siau *et al.*, 2001) and finding deficiencies in OOD (Laitenberger *et al.*, 1999). Particularly, we find the two aforementioned properties supporting abstract thinking to exist in UML.

3.1. *Abstraction Barriers*

Setting abstraction barriers is an important tool in design (Leron, 1987) and specifically for understanding the abstract concepts of OO paradigm (e.g., *class*, *abstract class*, *encapsulation*, *polymorphism*, etc.). Abstraction barriers define a specific level of abstraction to be addressed at a given stage. Some design approaches are aimed at emphasizing this particular aspect. For example, the Responsibility-Driven Approach (Wirfs-Brock and Wilkerson, 1989) supports the increase of encapsulation during design, by focusing first at the contractual responsibilities of a class, postponing implementation considerations until a later stage.

By setting abstraction barriers, UML leads the developer to refer only to the details appropriate to the abstraction level required at each stage or aspect of the development. For example, when defining the system's classes, the developer uses class diagram which supports definitions of classes and their names, the associations among them and the declarations of their properties and methods. However, there is no room in class diagram to refer, for example, to the inner algorithmic aspects of these methods. This leads the developer to stay at the required abstraction level at this stage, referring to the relevant static

aspect of the system, without being distracted by the many details related to the implementation. As mentioned, students tend to refer to low abstraction levels at early stages of the development, or when trying to comprehend an existing system, which leads them to low quality results of developed artifacts or program comprehension. Applying a tool that sets abstraction barriers (i.e., guides users to avoid relating to details not necessary for the current stage) may lead to better understanding and performance in OO development tasks.

3.2. *Visual Representation*

The syntax of UML is based on diagrams, thus functioning as a visual representation of the system. A visual model is a simplified representation of a phenomenon, used to support scientific exploration or explanation of the phenomenon (Gilbert *et al.*, 2000). In mathematics and natural sciences, and lately in CS as well, the importance of visual models such as graphs and diagrams is well acknowledged for the understanding of abstract terms and phenomena. While dealing with abstract concepts in CS, the use of visual models, either mental or on paper, lowers the effort and abstract thinking skills needed to understand it (Aharoni, 2000). Hendrix *et al.* (2000) empirically demonstrated that visual models have a measurable contribution to understanding concepts and principles in CS.

We believe that using UML supports ad-hoc work at a high abstraction level as well as the development of abstract thinking skills within this domain. This may release the students' thinking from "getting stuck" at the lower abstraction levels. However, we acknowledge that abstract thinking needs to evolve on the basis of the understanding of the concrete concepts. For these two aims we suggest to support students in acquiring both concrete and abstract thinking skills using an iterative teaching methodology.

4. The Iterative Teaching Methodology

The iterative concept is based on successive sessions of abstract modeling and detailed implementations by coding. By using a CASE tool, the lecturer initially draws the fundamental entities' shapes and their associations, discusses their theoretical meaning, generates the code automatically to demonstrate the code shell (class name, methods etc.) and concludes with inner detailed code insertion. After generating the code and viewing abstract as well as detailed levels (diagrams and code) on the same screen, the students are expected to absorb the equivalence of the two views and become familiar with the simplified abstract representation.

In what follows we demonstrate the suggested strategy for teaching the OO concepts: *Responsibility*, *Message Passing*, *Encapsulation*, *Generalization* and *Functionality Expansion.* For the sake of clarity, in this paper we chose to use *Magic Draw 11* as our CASE tool, and *Java* as the programming language.

Fig. 1. Abstract representation of a responsibility division.

4.1. *Teaching the Principle of Responsibility Division*

The goal of teaching the principle of responsibility division between objects and their methods of activation is targeted towards encapsulation and visibility. In this example the responsibility is divided between the object *objOutput* from class *Output*, presenting the graphical format of a string, and the object *objMessageFactory* from the class *Message-Factory*, generating the string to be displayed.

The evolving maturity is iteratively presented to the students. At each cycle a new concept is presented; the corresponding element is inserted at the UML diagram level; the code is automatically updated with the class properties by using the CASE tool and the internal details are manually added to the code. The two abstraction levels are presented as interweaved environments, which later assist in conducting further detailed analysis and design. The following example displays the abstract and concrete elements of this iteration.

Abstract step. Drawing classes and determining their members. The abstraction barrier focuses the students on the "separation of responsibility" (Fig. 1) and the concepts of visibility (private, public) by means of structural class representation.

Concrete step. Automatically generating the equivalent representation of the static class diagram to code:

```
Class Output {
   public void DisplayMessage( ) {
      }
}
Class MessageFactory {
   private String StringToDisplay="Hello World";
   public String GetString( ) {
      }
}
```

The *separation of responsibility* is implemented in a closed UML box as well as the parenthesis { } definition. However, the difference between the method and class scope, while may be confusing in the programming notation, is very clear within the UML notation due to separate visual representations. It is located in different boxes with relative names, which leads to the understating of the next notion of *encapsulation*.

4.2. *Teaching the Concepts of Message Passing, Encapsulation and Request for Service*

Abstract Step 1. Adding the "who uses who" relation to the class diagram demonstrates the abstraction barrier of dependency relations on class diagram (Fig. 2), and illustrates the *dependency (uses) notion*.

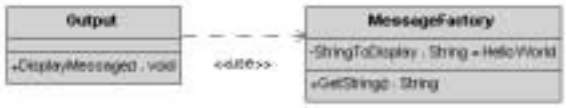

Fig. 2. Abstract representation of a dependency between classes example.

Abstract Step 2. The sequence diagram's abstraction barrier relates here to behavioral message passing between objects. It illustrates the concept of *service* and the notion of *insanitation* in order to provide a service (Fig. 3). This enables discussing message passing without considering the structural decomposition of each object.

Concrete step. After the generated signature of the classes is created by the CASE tool, the inner implementation of the DisplayMessage method is programmed by the lecturer within the IDE (integrated development environment) tool.

```
public void DisplayMessage( ) {
  MessageFactory objMessageFactory =
      new MessageFactory();
  String stringForMe=
      objMessageFactory.GetString();
  System.out.println(stringForMe);
}
```

From this point on, the lecturer can iterate between changing the default values of the display string and adding a different implementation for the display such as creating a graphical message box using the *alert* command. This example demonstrates two notions: the code level *encapsulation* concept and *information hiding* of an abstract model. It is important not to change the actual meaning of the diagrams in order to emphasize the differences between the abstraction levels of model and code.

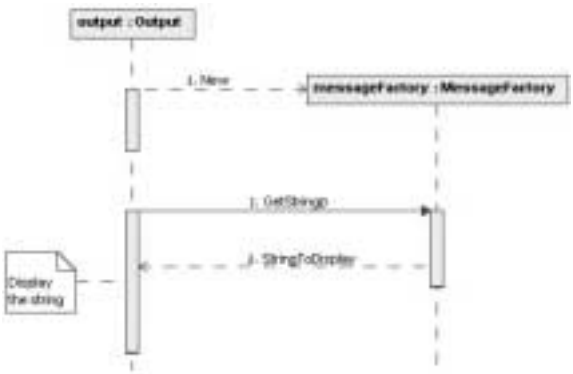

Fig. 3. Abstract representation of a service.

4.3. *Teaching the Concepts of Generalization and Functionality Expansion*

Abstract Step. This step illustrates the direction of *generalization* which is the direction of the UML arrow (Fig. 4). The basic example of the relation between classes `Sub` and `Super`, with relevant functionalities: `DoSubThing` and `DoSuperThing`. This means that when an object is created from the sub class, it "searches" for the requested behavior or property within the class code it was instantiated from. If the requested service is not defined at this class level, it is being directed according to the *generalization* UML arrow to its super class. This way, the concept of dynamically finding the behavior is quickly comprehended: just "follow the yellow brick (generalization arrows) road", until you find the requested behavior, and execute it.

The abstract steps are divided into two phases: the "finding the behavior" road and the viewing the accumulated available behavior of a class using a CASE tool. We first present the Super class methods (Fig. 5) followed by the Sub class methods (Fig. 6).

The students are presented with the two different sections of the CASE tool; the upper section with the title "General" was originated by the `Sub` class and the one with the title "Inherited" was generated by the `Super` class.

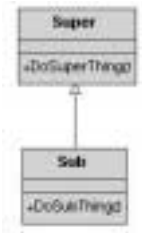

Fig. 4. Abstract representation of generalization.

Fig. 5. Magic Draw's representation of internal methods of the `Super` class.

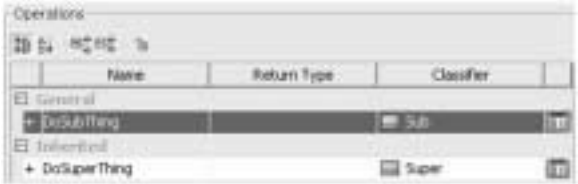

Fig. 6. Magic Draw's representation of internal and inherited methods of the `Sub` class.

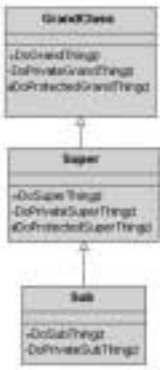

Fig. 7. Visibility (Private, Public, and Protected) and multilevel generalization.

Concrete Step. Automatically generate the code that illustrates these methods:

```
public class Super {
      public void DoSuperThing( ){
      }
}
public class Sub extends Super {
      public void DoSubThing( ){
      }
}
```

The students are referred to the keyword *extends* (Java) and are explained its resemblance to the UML arrow. Notice that the term “extends” may cause problems in cases where the sub class does not add extension, but rather overrides behavior, or is used for polymorphic purposes. Next, we iterate and insert the visibility properties of elements, namely the public, private and protected (Fig. 7). Different visibility attributes are added to the functions, as well as a higher super class named GrandClass.

Concrete Step. Using the CASE tool automatic code generation.

```
public class GrandClass{
      public void DoGrandThing( ){
      }
      private void DoPrivateGrandThing( ){
      }
      protected void DoProtectedGrandThing( ){
      }
}
public class Super extends GrandClass{
      public void DoSuperThing( ){
      }
      private void DoPrivateSuperThing( ){
      }
      protected void DoProtectedSuperThing( ){
      }
}
public class Sub extends Super{
      public void DoSubThing( ){
```

```
        }
        private void DoPrivateSubThing( ){
        }
}
```

The goal of this example is to show the automatic additions of behavior when extending a base class. By using a walkthrough conducted by the lecturer, it demonstrates direction of locating a behavior. Naturally, assuming there are no overriding or non-polymorphic duplicated methods, as this example exhibits.

5. Applying the Iterative Teaching Method: a Case Study

The iterative teaching method was applied during an "Introduction to Software Engineering" course, taught by the second author, at the section of the course related to OO. The students participating in the course had taken a C language course and a four-lessons introduction to C++ concepts during the previous year. The number of students participating in the course was 20, all from the Department of CS at the University of Haifa, during the third semester of their studies.

5.1. *Applying the Iterative Teaching Method*

The iterative teaching method was initially applied due to a time constraint. A serious condition was identified where the students were supposed to be familiar with the basic concepts of OO implemented with C++. However, when trying to perform a basic exercise in the beginning of the course, the students exhibited confusion and difficulties in using fundamental OO concepts. Among the difficulties observed were:

- Misusing *aggregation* and *generalization*. For example, constructing functional expansion only by using generalization. This led to generation of many *overrides* and *overloads* instead of combining basic service consumptions via aggregation.
- Abusing the usage of *polymorphism*. For example, creating huge sized classes (without using the Liskov substitution rule). This caused under-usage of classes' interfaces or misusage of abstract classes and their virtual substitution.
- Misunderstanding *coupling* and *dependencies*. Leading, for example, to incorrect message passing due to duplicated methods naming, erroneous separation of responsibilities of the designed classes and separation of data from behavior, generating high coupling.
- Low *cohesion*. Many cases were observed in which the classes included a mere collection of functions without any commonality or inner dependencies. When asked on the nature of encapsulation, the students replied with the visibility argument of preventing access to service: public, private, protected. However they rarely employed common usage of private functions among several public interface ones. The objects were loaded into the memory and then destroyed, merely for the usage of a single public function which does not require any object state, private or inherited behavior.

This situation called for finding a quick and effective way to teach the essential OO concepts in no more than 3 sessions, each of 4 hours, in order to reach a point where the students would be able to produce adequate artifacts for OO exercises. Achieving this goal was crucial before proceeding with the remainder of the course.

The lecturer decided to use the CASE tool provided for this course (Rational Rose) and started using it from the very start: teaching a concept, modeling, generating the respective code, and constantly synchronizing between the two views. This process was applied for each basic OO concept according to the principles presented in the previous section.

5.2. *Evaluation Tools*

As this was a single case study conducted mainly to qualitatively explore phenomena related to this experience of applying the iterative teaching methodology rather than statistically corroborating a hypothesis, our evaluation tools were chosen accordingly. Nevertheless, some comparison with other courses has been made, under reservations, as will be later elaborated. The evaluation tools for this study included:

1. The final test of the course.
   The test examined the students' understanding of the studied OO concepts and their ability to correctly use and implement these concepts.
2. A position questionnaire regarding the quality of the course.
   This questionnaire evaluated, among other things, the students' position regarding their general satisfaction from the course, whether they found the course useful and whether they thought it was interestingly taught.
3. Interviews with students, focusing on the iterative teaching methodology and its perceived contribution.

The quantitative outcomes obtained via the two first evaluation tools were compared to the respective outcomes of the same tools gathered in three other similar courses taught in the CS departments of three different universities (one of which was the university where the current course took place). In these three courses the iterative teaching method was not applied. All four courses (including the one presented in this case study) were taught by the same lecturer. The outcomes of the interviews and the textual data from the tests were qualitatively analyzed in order to understand how this approach affected students' perception and understanding of the material taught.

5.3. *Results*

The test given to the students in all four courses was similar in content-mixture and difficulty level. This was assured by the lecturer's test generation method. The lecturer holds a large bank of questions related to the course's content, where the questions are categorized according to issue and difficulty level. When constructing a test, the lecturer selects the questions according to fixed parameters of mix relations among issues and difficulty levels.

The iterative teaching method was aimed at closing the gaps in the students' existing knowledge. One analysis conducted to examine whether this was achieved was by comparing the test grades in this course to the test grades in the other three courses. However, this comparison needs to take into consideration that the students in the iterative course were less advanced than the students in the other (control) courses.

The students in the iterative course (noted at Fig. 8 as the *I course*) were university computer science students in their third semester, after two programming courses (one in C and one in C++).

All students in the three control courses were students during their fifth semester, after having had at least four OO related courses. The three control groups differed from each other in the university type and their study track. The first control group (*U1*) was in a prestige technology and science institute, in the computer science faculty, software engineering track. The second control group (*U2*) was in an academic college in the school of computer science, software engineering track. The third control group (*U3*) included computer science students in the same university and the same department (computer science) where the iterative course took place.

The relevant grades and their standard deviations are presented in Fig. 8. The iterative course average grade was 75.5, while the other courses' average grades were 72.4, 77, and 83.1. The highest average grade was achieved at the university where the general students' achievement profile is considerably higher with comparison to the others.

As can be seen, the students' average achievements in the iterative course were within the grade scales of the other courses. Therefore, one may conclude that the boot-camp succeeded in closing the gap in a short time, bringing these second year students to similar level of OO understanding as their peers achieved in their third year.

It is important to note that this comparison between the courses should be considered with much caution: while some variations were carefully controlled (e.g., the same lecturer taught all four courses and all courses were conducted in CS departments), some were not (e.g., different universities). This may give an initial indication that the suggested iterative teaching methodology is effective, however, a more comprehensive quantitative research needs to be conducted in order to further generalize this conclusion.

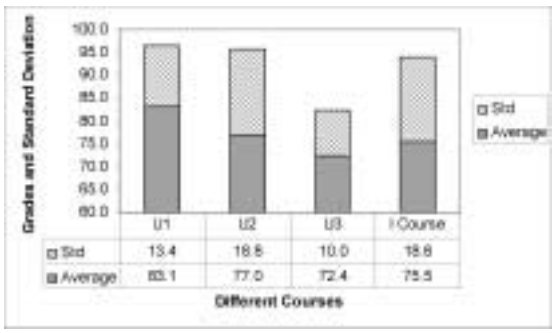

Fig. 8. Courses test grades comparison. The *I course* stands for the iterative teaching course, the *U1, U2, U3* markers stand for the other courses; *the I course* and the *U2* took place at the same university.

The results from the questionnaires showed a very high level of student satisfaction from the course in which the iterative teaching method was applied. Moreover, this course was selected as the best course taught in the department that semester, receiving the highest total course grade according to the university courses quality questionnaire, i.e., 4.89 out of 5. Specifically, in the "interesting teaching" section it received 4.95 and in the "coherent and clear teaching" section it received 4.68 out of 5. Both grades were at the top level of the university. The university's average of courses grades that semester was 4.43 with standard deviation of 0.41 out of 377 courses. In the other three courses, where the iterative teaching method was not applied, the range of the courses' satisfaction grades was 3.7–3.9.

The analysis of the interviews indicated that the students generally believed that this teaching method helped them significantly in their understanding of OO concepts and principles. Following are a few examples of the students' statements related to this issue. "This method makes it very easy to browse and understand the 'tons' of information required for OO courses"; "I don't understand why we were not using these tools from the start [previous courses]. The usage of the CASE tool assists considerably in 'seeing' and understanding the complex issues that we learned"; "The dual view of diagrams and code helps us focus and understand how the program should be constructed. It is easy to see a mistake such as high coupling or wrong decisions".

The textual analysis of the tests indicated that the basic difficulties identified in the beginning of the course were overcome: the students used the principles of aggregation, generalization and polymorphism properly; the levels of cohesion in their solutions were improved and the objects' responsibilities were well defined. We do not imply of course that the solutions were perfect, merely that a basic OO concept understanding has been achieved.

5.4. *Case Study Discussion*

The case study presented in this section relates to a single course of 20 students. During this course the motivation for a highly effective and efficient teaching method arose for resolving an ad-hoc crisis (namely, the absence of the knowledge that the students were expected to have); the new iterative teaching method was applied and its effect on students' understanding was explored. While further research needs to be conducted in order to obtain conclusions that may further be generalized, we find the results of the case study presented here as encouraging to follow this direction.

6. Conclusion

In this paper we presented a methodology for teaching OO concepts using an iterative approach cycling between visual representation and detailed code, utilizing a CASE tool. We found this iteration between abstract visual models and concrete implementation effective in focusing students on the different aspects of OO basic concepts. Instead of deciding whether to teach programming or model first, we propose to teach them both

in parallel, alternating between these two views. Consequently the development of both skills – programming and design, namely concrete and abstract thinking respectively – will be simultaneously encouraged. Moreover, we propose to use visual modeling as the universal language of design, whereas the selected implementation language serves merely as an example. Thus, it becomes easier to transfer the implementation knowledge among the various programming languages, while maintaining the core concepts of OO as mathematical axioms.

We believe that the iterative methodology for teaching OO concepts may be a step in the right direction towards supporting the development of students' abstract thinking skills, thus improving their OO concept understanding. Future research may quantitatively study the effect of the suggested methodology on the learning processes. An additional direction of research may explore the application of teaching methodology iterating between different abstraction levels to additional areas in CS and software engineering.

References

Aharoni, D. (2000). What you see is what you get: the influence of visualization on the perception of data structures. In T. Nakahara and M. Koyama (Eds.), *Proceedings of the 24th Conference of the International Group for the Psychology of Mathematics Education* (*PME24*), vol. 2. Hiroshima University, Hiroshima, pp. 2-1–2-8.

Aharoni, D., and U. Leron (1997). Abstraction is hard in computer-science too. In E. Pehkonen (Ed.), *Proceedings of the 21st Conference of the International Group for the Psychology of Mathematics Education*. University of Helsinki, Lahti, Finland.

Armstrong, D.J. (2006). The quarks of object-oriented development. *Communications of the ACM*, **49**(2), 123–128.

Bennedsen, J., and M.E. Caspersen (2004). Programming in context – a model-first approach to CS1. In *SIGCSE Technical Symposium on Computer Science Education*. pp. 477–481.

Berge, O., R.E. Borge, A. Fjuk, J. Kaasbųll and T. Samuelsen (2003). Learning object-oriented programming. Paper presented at the *Norsk Informatikkkonferanse* (Norwegian Informatics Conference).

Booch, G. (1999). UML in action. *Communications of the ACM*, **42**(10), 26–28.

Bucci, P., T.J. Long and B.W. Weide (2001). Do we really teach abstraction? In *SIGCSE Technical Symposium on Computer Science Education*, pp. 26–30.

Dubinsky, E. (1991). Reflective abstraction in advanced mathematical thinking. In D. Tall (Ed.), *Advanced Mathematical Thinking*. Kluwer, Netherlands, pp. 95–123.

Fleury, A.E. (2001). Encapsulation and reuse as viewed by Java students. *SIGCSE 2001*, 2/01, 189–194.

Gilbert, J.K., C.J. Boulter and R. Elmer (2000). Positioning models in science education and in design and technology education. In J.K. Gilbert and C.J. Boulter (Eds.), *Developing Models in Science Education*. Kluwer Academic Publisher, pp. 3–17.

Hadar, I. (2004). *The Study of Concept Understanding via Abstract Representation: The Case of Object Oriented Design*. Research thesis for Ph.D. Israel Institute of Technology, Technion.

Hadar, I., and O. Hazzan (2004). On the contribution of UML diagrams to software system comprehension. *Journal of Object Technology*, **3**(1), 143–156.

Hadar, I., and U. Leron (to be published). *How Intuitive is Object Oriented Design*? (preprint). To be published in *Communications of the ACM*.

Hazzan, O., and I. Hadar (2005). Reducing abstraction in graph theory. *Computers in Mathematics and Science Teaching*, **24**(3), 255–272.

Hendrix, D.T., J.H. Cross II, S. Maghsoodloo and M.L. McKinney (2000). Do visualizations improve program comprehensibility? *Experiments with Control Structure Diagrams for Java*, *SIGCSE*, March, 382–386.

Holmboe, C. (1999). A cognitive framework for knowledge in informatics: The case of Object-Orientation. In *ITiCSE'99 Conference Proceedings*, June, 1999. pp. 17–20.

Kobryn, C. (1999). A standardization Odyssey. *Communications of the ACM*, October, **42**, 29–37.
Laitenberger, O., A. Colin, M. Schlich and K. El-Eman (1999). *An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents*. NRC-CNRC, National Research Council Canada.
Leron, U. (1987). Abstraction barriers in mathematics and computer science. In J. Hilel (Ed.), *Proceedings of the 3rd Int. Conference for Logo and Mathematics Education*.
Leroux, H., and C. Exton (2001). Visualising the execution of concurrent object-oriented programs dynamically using UML. In *WSCG'2001 – The 9th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2001*, February 2001.
Siau, K., L. Lee and J. Korhonen (2001). Use case diagram in requirement analysis: an empirical investigation. In *Proceedings EMMSAD'01*. pp. VIII-1–VIII-7.
Sfard, A. (1991). On the dual nature of mathematical conceptions: reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, **22**, 1–36.
Wirfs-Brock, R., and B. Wilkerson (1989). Object-oriented design: a responsibility-driven approach. In *Proceedings OOPSLA'89*, pp. 71–75.

I. Hadar is a lecturer at the Department of Management Information Systems at the University of Haifa. Her research focuses on human aspects of software engineering: cognitive processes of software development; the role and influence of visual models in requirement engineering and software design; and human factors – organizational, management, social and cognitive factors – and their influence on software quality. Irit has her PhD from the Department of Education in Technology and Science of the Technion – Israel Institute of Technology. Both her master and bachelor degrees are from the Faculty of Industrial Engineering and Management of the Technion.

E. Hadar is a chief methodologist at HP Software. His responsibilities include the development of new methodologies in software engineering, service oriented architecture and object oriented technology, as well as conducting education and coaching sessions. Prior to his current job, Ethan was a faculty member at Netanya Academic College, at the Software Engineering Department. In parallel Ethan has more than 17 years of consulting experience in mentoring R& D teams in design and software engineering related issues. He holds a PhD degree from the Department of System Analysis and Operation Research of the Technion – Israel Institute of Technology. Both his master and bachelor degrees are from the Faculty of Mechatronics of the Technion.

Iteracinė metodologija mokant objektinių koncepcijų

Irit HADAR, Ethan HADAR

Abstraktus mąstymas yra svarbus mokantis kompiuterių mokslo. Kadangi kompiuterija iš esmės pagrįsta objektine technologija ir koncepcijomis, tai šis įgūdis tampa netgi esminiu. Ankstesni tiriamieji darbai teigia, kad studentai, besimokantys netgi geriausiuose universitetuose, taip pat kaip ir patyrę pramonės praktikai turi sunkumų vartodami abstrakčius terminus, plėtodami objektines technologijas. Šiame straipsnyje siūloma iteracinė mokymo metodologija, kuri būtų naudojama mokant objektinių koncepcijų. Pateikta metodologija remiasi tuo, kad besimokantiesiems siūloma susipažinti su modeliavimo kalbomis ir priemonėmis vos pradėjus studijas ir modeliai siejami su jų programomis. Teoriškai nagrinėjamas modeliavimo kalbų įnašas, iš dalies – UML, paskui pereinama prie abstraktaus mąstymo ir tada – prie objektinių koncepcijų supratimo. Straipsnyje pateikta keletas stebėjimų, atliktų metodologijos bandymo metu ir mokant konkretų kursą universitete.