I.J.Modern Education and Computer Science, **2013, 7, 1-15**
Published Online September 2013 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2013.07.01

Students' Understanding of Selected Aspects of Interface Class in Java

Ilana Lavy
Department of Management Information Systems, the Max Stern Yezreel Valley College (YVC), Israel
Email: ilanal@yvc.ac.il

Rami Rashkovits
Department of Management Information Systems, the Max Stern Yezreel Valley College (YVC), Israel
Email: ramir@yvc.ac.il

***Abstract*—**This study examines the understanding of various aspects relating to the concept of interface class by Management Information Systems students. The examined aspects were: definition, implementation, class hierarchy and polymorphism. The main contributions of this paper are as follows: we developed a questionnaire addressing the above aspects; we classified and analysed the students' responses to determine the students' understanding of the above aspects and to highlight common faulty solutions. The results obtained reveal that majority of the students demonstrated understanding of definition and implementation of interface class, however, only two-thirds of the students demonstrated understanding of interface class in the context of class hierarchy and only one third of them demonstrated understanding of polymorphism in the context of interface class. The students' utterances from the interviews shed light on their difficulties.

***Index Terms*—**Computer science education, software engineering, learning ability, advanced programming courses.

I. INTRODUCTION

Interface classes are considered to be one of the main constituents of modern object oriented programming languages, and are commonly used in various fundamental and advanced software packages. Using these packages necessitates the understanding of the interface class concept (for simplicity we will use 'interface' when referring to interface class in the rest of this paper). However, interface is among the most difficult issues to understand with respect to object oriented programming [1]. Its vague nature makes it non intuitive for learners who do not grasp its essence and purpose. Understanding the interface concept is based on the prior understanding of the following: definition and use of concrete classes, construction of class hierarchies, abstract classes, methods overriding and polymorphism. These concepts are complicated and difficult to understand. Moreover, any misunderstanding concerning one of them may make the understanding of interface harder.

The interface in its most common form is a collection (potentially empty) of related abstract methods, and may not contain data members, except constants. The purpose of interface is to form a public Application Programming Interface (API) related to a certain concept, and enforce the API on classes which apply the concept. Furthermore, interface defines a new data type in that each object applying it is defined as an item of that type. The latter enables objects belonging to unrelated class hierarchies to share a common type.

Many references in the research literature point towards students' difficulties concerning object oriented concepts, such as objects and classes [2,3], abstraction and object orientation [4], and inheritance and polymorphism [1,5,6,7]. Despite the fact that interface is a fundamental and essential constituent of the object oriented paradigm, it has only received minor attention.

Many researchers concerned with the issue of the levels of abstraction and understanding required for the learning of scientific concepts have enriched the research literature with useful models [8,9]. Other researchers concerned with issues regarding the causes underlying the difficulties in understanding scientific concepts have augmented the research literature with useful insights. In a previous study [10] we examined the scope of the implementation of the Java interface among third year Management Information Systems (MIS) students, after they had studied and applied this issue. The results obtained revealed that the majority of the students have difficulties in identifying situations where interfaces can be used to solve design problems, as well as in applying interfaces in the program.

In light of the above, the present study focuses on mapping the study participants' understanding concerning various aspects of the concept of interface class. The main contributions of this paper are as follows: (a) we construct a written questionnaire for mapping the students' understanding of various aspects of interface class; (b) we discuss the students' solutions provided at each aspect; (c) we discuss the students' reflections concerning the questionnaire.

The remainder of this paper is organized as follows. The theoretical background section provides background information on interface classes in Java, educational aspects of teaching interfaces, and known difficulties. The study section presents the study environment, the

various aspects of interface class examined, and the questionnaire used and its expected solutions. The results section presents the results obtained. It discusses the provided informal definitions of interface class, the provided faulty solutions, and the students' reflections on the questionnaire. The concluding section includes both concluding remarks and possible implications on the educational process concerning the teaching of interfaces.

II. THEORETICAL BACKGROUND

In what follows we present the interface concept and its relation with other object oriented constituents. Then we describe the context in which the interface concept is taught. In addition, a brief literature survey regarding difficulties in understanding and implementing object oriented constructs in general and interface in particular is presented.

A. Interface as an object oriented construct

One the main constructs that the object oriented paradigm provides is the inheritance mechanism, which allows the reuse of an existing class by other classes. Using this mechanism improves the quality of the design by allowing common attributes and methods defined as belonging to the super-class to be used later in sub-classes, and to refer to various classes in the same manner using a common reference (polymorphism). The object oriented paradigm also allows the definition of abstract classes which are not aimed to be instantiated, but to serve as a base class to other classes. This construct may include regular as well as abstract methods. Abstract methods serve as an obligatory contract for the classes that inherit them, namely an implementation for these methods must be provided. Interfaces are pure abstract classes that may only include abstract methods and constant variables. The general use of interface is to define common capabilities among classes that are not necessarily derived from the same class hierarchy. These common capabilities have to be implemented by each derived sub-class. Interfaces are also used for class typing and hence can be used in a polymorphic way, taking advantage of this benefit.

We scanned various definitions of interface in text books and other sources, and herein we present excerpts taken from [11] describing the various aspects of the interface construct:

"A Java interface is a collection of constants and abstract methods. … An interface cannot be instantiated … A class implements an interface by providing method implementations for each of the abstract methods defined in the interface … The interface guarantees that the class implements certain methods, but it does not restrict it from having others … Multiple classes can implement the same interface, providing alternative definitions for the methods … A class can implement more than one interface … The interface construct formally defines the ways in which we can interact with a class. It also serves as a basis for a powerful programming technique called polymorphism".

The above excerpts include both definition and possible uses of interface. Interface and abstract class concepts share the following attributes: both are not concrete, are designated for use by other classes, and both allow the definition of abstract methods (with no body) and constants. However, they differ in various aspects, among them being: (a) abstract class can include concrete methods and variables, while interface cannot; (b) a class can implement more than one interface but can only inherit from one abstract class; (c) abstract classes have a constructor, but an interface does not have one.

Interface class as an object oriented construct has received only minor attention in the research literature. Hu [12] provided a summary of common uses of type inheritance allowing the definition of new types based upon existing ones. Among these uses he indicates sub-typing by implementing interfaces. Hu [12] claims that using interface is often better than ordinary class for inheritance for the purpose of maintaining behavioural compatibility. Schmolitzky [13] refers to the difference between hierarchies of types (class-based) and hierarchies of implementations (interface-based), and states that most textbooks on object oriented programming in Java do not make a clear distinction between them.

B. Interface in the curriculum

In many university and college Information Systems (IS) programs, first year students learn OOP language in two successive courses. The first, 'Introduction to Computer Science,' teaches them basic programming concepts using Java programming language, and variables, arrays, classes and methods. The second course is 'Object Oriented Programming' (OOP) and includes advanced programming concepts including class inheritance, polymorphism, abstract classes, interfaces, exceptions handling, input/output and graphical user interface [14]. Examination of the timetables of OOP courses at various universities and colleges reveals that the OOP courses are quite intensive and in many cases the time dedicated to teach interfaces compared to class inheritance is rather short. Furthermore, interfaces are often taught in the second half of the semester leaving little time to practice application.

Teaching interface immediately following class inheritance, abstract classes, and polymorphism, rather than as a stand-alone issue, might also lead to a bias in the students' perception of its role and importance, including a diminution of its capabilities and uses [15, 16].

C. Difficulties in understanding object oriented constructs

The Object oriented paradigm has become the dominant programming paradigm for software development in the last two decades. Java is one of the

popular object oriented programming languages that is taught in higher education. Students who study the Java programming language often find some of its constituents, especially those related to inheritance and polymorphism, difficult to understand due to the high level of abstraction required [7,17]. Hadjerrouit [18] compared object oriented with procedural programming languages, and claimed that the former requires more sophisticated abstraction abilities as well as greater attention to analysis and design, especially when large programs are involved [19]. Many students, either novice or experienced with procedural programming language, demonstrate difficulties in understanding certain subjects in inheritance and polymorphism, including chain of constructor calls in object creation, dynamic binding, and casting issues [5, 6].

Interface, as an object oriented programming construct, has been one of the most difficult concepts for students to understand and apply properly [1]. In our previous work [10], we engaged IS students with solving problems using interfaces and mapped their performances to five levels of abstraction. Only a few students demonstrated a high level of abstraction concerning the design and implementation of interface, while most of them failed to identify the need to model common behaviours when applying interfaces.

III. THE STUDY

We conducted a study which aimed to examine students' understanding of various constituents of the interface class concept. We focused on students in order to explore the impact of the educational process on the students' understanding of this concept. For this matter we found the qualitative research methods to be the most appropriate for the aims of the present study, as it enables us in-depth exploration of the subject. We chose the following aspects related to the interface class concept: defining interface classes, implementing interface classes within concrete classes, implementing interface classes in the context of class hierarchy, conceptualizing interfaces in the context of polymorphism. We built an exam-like questionnaire that includes four clusters of questions addressing the four aspects stated above. In addition we conducted informal interviews with a selection of the participants, in which the students' reflections on the questionnaire were collected, analysed and categorised. In this section, data concerning the environment and the study population are presented followed by the data analysis tool used. We conclude with the questionnaire and the anticipated solutions.

A. *Environment and population*

The data were collected during the academic years 2011-2012. The study subjects were third (and final) year students on a BA degree in Management Information Systems (MIS) at an academic college. Sixty-three students participated in the research, 31 graduated in 2011, 32 graduated in 2012. All the participants were graduates from the following programming courses: "Object oriented programming", "Data structures and algorithms", and "System analysis workshop". All of these courses include references to interfaces, and the students were provided with problems which necessitated the use of interfaces. The questionnaire was given to the students at the end of the course of system analysis workshop as one of the course duties and hence we consider the data to reflect the students' actual understanding.

The participants were provided with a questionnaire which included various questions requiring the understanding of using interfaces. Whilst engaging with the questionnaire, the students were not allowed to use any supplementary material, and had to rely on their knowledge in order to examine their understanding without the mediation of IDEs which provides automatic completions and suggestions. Therefore, in the process of assessing the students' solutions we ignored syntax errors.

B. *Data analysis tool*

Among the abstraction mechanisms available to programmers are interface classes. Proper use of interface classes necessitates profound understanding of it. Hence, in order to map the students' understanding of this concept we constructed several questions in which the student had to apply their knowledge concerning this concept. For that matter we made a list of properties related to interface class as follows: (1) definition of interface class; (2) implementation of interface class; (3) implementation of interface class in the context of class-hierarchy; (4) conceptualizing interfaces in the context of polymorphism. It should be notified that the above list is not necessarily complete, however we chose the above properties since they are most in use.

In Table 1 we present detailed explanations concerning the various aspects of interfaces as appears in the questions. All the questions are based on a given class hierarchy (Figure 1).

TABLE I : DISTRIBUTION OF QUESTIONS ACCORDING TO INTERFACE CLASS ASPECTS

Question no.	Aspect	Explanation
1	Definition	The student is provided with a description from which he has to construct an interface class without any method
2		The student is provided with a description from which he has to construct an interface class with appropriate abstract methods
3	Implementation	The student is provided with a given interface class and a list of requirements according to which he has to implement it within a single class
4		The student is provided with **several** interface classes and a list of requirements according to which he has to implement them in a **single** class
5		The student is provided with **single** interface and a list of requirements according to which he has to implement it within **several** classes
6	Hierarchy	The student is provided with a given interface class and a list of requirements according to which he has to implement it within an abstract class for all its decedent classes
7		The student is provided with a given interface class and a list of requirements according to which he has to implement it within an abstract class for all its decedent classes and in additional concrete class that belongs to another class hierarchy
8	Polymorphism	The student is asked to construct a method with an object parameter referred as interface, and then to perform casting to the object's class type in order to access methods that are included in the class but not in the interface
9		The student is asked to construct a method with an object parameter referred as a base class, and then to perform a casting to the object's interface in order to access methods that are included in the interface but not in the base class

```
abstract class Animal{

      protected double weight = 0; // each animal has its own weight
      public double getWeight() { return weight;}
      public void setWeight(double w) { weight = w;}
}

abstract class Mammal extends Animal{}
abstract class Fish extends Animal{}
abstract class Bird extends Animal{}

Concrete classes:
class Mouse extends Mammal{}
class Dog extends Mammal{}
class Shark extends Fish{}
class Salmon extends Fish{}
class Hawk extends Bird{}
class Penguin extends Bird{}
```

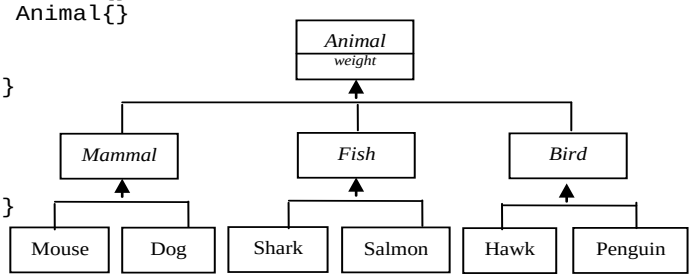

Figure 1: Animal class hierarchy

C. Questionnaire

In order to assess the students' understanding concerning various aspects of interface classes we designed a questionnaire consisting of several clusters of questions each addressing a certain aspect of interface class (Table 1). Some questions included code fragments to which the students were asked to refer to them. To avoid difficulties stemming from the understanding of complex task we decided to use a simplified example in which the student can concentrate mainly on the interface class concept and its aspects. In addition, most of the questions were designed independently of each other to avoid situations in which failure in one do not influence other.

The questionnaire consists of two successive parts: (A) one question in which the students had to provide an informal definition of the interface class concept and its possible uses; (B) nine questions in which the students' understanding of various aspects of the interface class concept is addressed. The question in part A of the questionnaire is:

Part A:

> Please describe in your own words what an interface is and specify its possible uses.

Part B:

First the students were presented with the following hierarchy of classes both as code fragment and as a schematic diagram (Fig. 1).

In what follows we present the questions of part B and the expected solutions organized in clusters, each refers to a certain aspect of interface class as stated in Table 1.

1) Interface class definition

The two questions in this category address the students' ability to define an interface class according to a list of requirements.

Question 1 addresses the ability of students to define a simple interface. The students were asked to define an interface that represents the concept of Endangered species. The students were notified that this property can be assigned to some animals. It is worth noting that the question does not include any requirements to specify attributes or methods concerning the endangered species concept, and hence no such constituents are needed in the solution.

> **Question 1:** It is well known that some species are declared endangered. Please provide an appropriate interface that represents endangered species.

Expected solution to question 1:

```
interface Endangered {}
```

It should be notified that the Endangered interface does not contain any declaration of a method, and hence each class that implements it, does not have to implement methods related to it.

Question 2 addresses the ability of students to define a simple interface with several methods. The students were asked to define an interface that represents the concept of Vegetarian, which is a property common to many types of animals though not to all of them. The question specifies that for each vegetarian animal the amount of plants eaten per day should be kept, and should be accessible for update and retrieve operations.

> **Question 2:** Some of the animals are vegetarians, but not all of them eat the same amount of plants. We want to keep the amount of plants (in kg) eaten by each vegetarian animal per day, and be able to update and retrieve this property. Please provide an appropriate interface addressing these requirements.

Expected solution to question 2:

```
interface Vegetarian {
    void setAmountOfPlants(double amount);
    double getAmountOfPlants ();
}
```

It should be mentioned that the Vegetarian interface contains two methods one for update and one for retrieve of the plants-per-day amount.

Addressing properly the questions in this cluster requires one to understand how interface classes are defined, and what methods' declarations should be included.

2) Interface class Implementation

The three questions in this category address the students' ability to implement given interface classes according to a list of requirements.

Question 3 addresses the ability of students to use a given interface and provide the proper implementation in the relevant classes according to the requirements. The students were given the Licensed interface that includes two abstract methods referring to update and retrieve a license number. They were asked to apply it merely on dogs.

> **Question 3**: The following interface defines the methods referring to licensing animals:
>
> ```
> interface Licensed  {
>  void  setLicenseNumber(int number);
>  int getLicenseNumber();
> }
> ```
>
> Given that dogs must be licensed, and each has its own license number, what changes should be made to address these requirements.

Expected solution to question 3:

```
class Dog extends Mammal
  implements Licensed {
  private int licenseNumber;
  public void setLicenseNumber(int n){
    licenseNumber = n;
  }
  public int getLicenseNumber(){
       return licenseNumber; }
}
```

It should be notified that the given Licensed interface contains two methods one for update and one for retrieve of the license number. Hence, the Dog class that implements this interface should define a corresponding attribute for the license number, and implement the two methods involved.

In question 4 definitions of two interface classes are provided. The first represents a capability of flying with two methods and the second represents a capability of hunting with two other methods. The students were asked to implement the above two interfaces in the *Hawk* class.

Question 4: The following interfaces refer to the capabilities of flying and hunting:

```
interface Flying {
    int  getMaxFlightHeight();
    void setMaxFlightHeight(int h);
}

interface Hunter {
   Animal getFavouritePrey();
   void       setFavouritePrey();
}
```

Given that hawks are capable of flying and hunting, what changes should be made to address these requirements. Note that not all birds can fly (e.g., penguins).

Expected solution to question 4:

```
class Hawk extends Bird
 implements Flying, Hunter {
       maxFlightHeight;
       Animal prey;

       int getMaxFlightHeight() {
         return maxFlightHeight;
       }

       void setMaxFlightHeight(int h) {
         this.maxFlightHeight = h;
       }

             Animal getFavoritePrey(){
         return prey;
       }

       void set FavoritePrey(Animal a) {
         this.prey = a;
       }
}
```

It should be notified that the Hawk class should declare on both interface classes and implement all of their methods. It should also define corresponding attributes in the class, to support the implementation of these methods.

Question 5 addresses the ability of students to use a given interface and provide implementations of it in two classes, each implemented differently. The students were asked to implement the given interface in the Mouse and the Salmon classes, each with a specified behaviour.

Question 5: It is well known that some animals are used for scientific research in the pharmacologic industry. The following interface class represents experimenters who take place in pharmacological experiments. For any animal that participates in a scientific research we want to retrieve and update the names of the medicines involved:

```
interface PharmaExperimenter {
    List<String> getMedicines();
    void setMedicines(String m);
}
```

Mice and salmons are used in such scientific research as experimenters. While mice are able to take simultaneously several medicines, salmons can take only one at the time.

Expected solution to question 5:

```
Class Mouse extends Mammal
 implements PharmaExperimenter {
  ArrayList<String> medicineI=List =
       new ArrayList<String>();

  List<String> getMedicines() {
   return medicineList;
  }

  void setMedicine(String medicine) {
    medicineList.add(medicine)} ;
  }

Class Salmon extends Fish
 implements PharmaExperimenter {
   String medicine;

    List<String> getMedicines() {
     return new ArrayList().add(medicine)
    }

    void setMedicine(String medicine) {
      this.medicine = medicine;
    }
}
```

It should be notified that the Salmon class should define only one String attribute to hold the medicine it takes, while Mouse has to define a list of String objects. Furthermore, the Salmon has to return a List of medicines as declared by the getMedicines() method's signature, and therefore has to create a temporary list with the containing the only medicine allowed, and return it.

Addressing properly the questions in this cluster requires one to understand how to implement a given interface within various classes. In addition he should be able to identify, define and use properly attributes required for the implementation of the interfaces' methods. He should also be able to implement several interfaces within a single class and be able to provide different implementation of the same interface class at different classes.

3) Interface class and class hierarchy

The two questions in this category address the students' ability to implement given interface classes within various classes along the hierarchy (abstract classes included) according to a list of requirements.

Question 6 addresses the ability of students to use a given interface and provide one implementation of it in an abstract class, as all descendants of this class inherits the same behaviour. In such case separate implementations in each concrete class is redundant and flew.

Question 6: The following interface refers to the capability of Swimming

```
interface Swimmer {
   void setMaxDepth(int meters);
   int  getMaxDepth();
}
```

In the given hierarchy it is known that sharks and salmons can swim. What changes should be made in the given classes in order to address these new requirements.

Expected solution to question 6:

```
abstract class Fish extends Animal
 implements Swimming {
   int maxDepth;

   void setMaxDepth (int meters) {
       this. maxDepth = meters;
   }

   int getMaxDepth () {
       return maxDepth;
   }
}
```

It should be notified that since the Salmon and the Shark classes are the only descendants of the abstract class Fish, the implementation of the Swimming interface should be located at the Fish class, as all descendants inherit the same behaviour.

Question 7 addresses the ability of students to use a given interface and provide two implementations of it: one for sub-tree of classes rooted at specified abstract class at the root, and second for another concrete class that does not belong to that tree.

Question 7: The following interface refers to pregnancy

```
interface Pregnant {
   void setGestationPeriod (int
d);
   int getGestationPeriod ();
}
```

It is known that the offspring of dogs, mice and sharks is born after a gestation period. What changes should be made in the given classes in order to address this new requirement.

Expected solution to question 7:

```
abstract class Mammal extends Animal
 implements Pregnant {
   int gestationPeriod;

   void setGestationPeriod (int days) {
       this.gestationPeriod = days;
   }

   int getGestationPeriod (){
       return this.gestationPeriod;
   }
 }

class Shark extends Fish
 implements Pregnant {
   int gestationPeriod;

   void setGestationPeriod (int days) {
       this.gestationPeriod = days;
   }

   int getGestationPeriod (){
        return this.gestationPeriod;
   }
 }
```

It should be stressed that since dogs and mice are the only descendants of the abstract class Mammal, the implementation of the Pregnant interface should be located at the Mammal class. However, since sharks do not inherit from mammals, a separate implementation of the Pregnant interface has to be added to the Shark class, although the implementation is identical.

Addressing properly the questions in this cluster requires one to understand the principles of class hierarchy in the context of interface classes. One has to be able to decide where to implement the interface class along the class hierarchy in a way that avoid redundant code duplications.

4) Interface class and polymorphism

The two questions in this category address the students' ability to exchange views of an object from its interface type to its object type and vice versa as needed.

Question 8 addresses the ability of students to use a proper reference type to access specified methods. More specifically, one has to be able to perform a casting operation from a base class type to an interface type in order to invoke the interface's methods.

Question 8: Please provide an implementation of a method that takes a *Flying* object and displays its max flight height. The method will also display the animal's weight, in case object is an animal.

Expected solution to question 8:

```
void someMethod(Flying obj){
   System.out.println
      (obj.getMaxFlightHeight());

    if (obj instanceof Animal)
       System.out.println
        (((Animal)obj.getWeight());
}
```

It should be notified that obj is accepted by its interface type and not by its class type. Therefore only the interface methods can be accessed via obj. Any other method of the object that has to be invoked must be preceded with a proper casting operation. However, it is not safe to perform a casting operation without first examine the type of obj. For example, one could implement a class representing planes, and choose to implement Flying interface within it. Obviously, Plane do not inherit from Animal, and casting obj to Animal without first examine its type may result in severe error.

Similarly to the previous question, question 9 addresses the ability of students to use a proper reference type to access specified methods. In this case, one has to be able to perform a casting operation from an interface type to a class type to in order to invoke the class methods.

Question 9: Please provide an implementation of a method that takes an *Animal* object as a parameter, and displays its weight. If the animal provided is a hunter, it also displays its favourite prey.

Expected solution to question 9:

```
void someMethod(Animal obj){
    System.out.println
      (obj.getWeight());

    if (obj instanceof Hunter)
     System.out.println
      (((Hunter)obj.getFavouritePrey());
}
```

It should be notified that obj is accepted by its base class type that already includes the weight attribute and related methods. Therefore, the weight-related methods can be accessed directly via obj. However, not all animals are hunters; therefore a proper casting to Hunter interface is required in order to invoke the methods related to it. As in the previous question, it is not safe to perform a casting operation without first examines the type of obj, and an examination if obj is a hunter precedes the casting operation.

Proper solution to question 9 requires one to be able to demonstrate mastery in the principle of polymorphism in the context of interface classes. It requires one to be able to properly handle an object based on its reference type. Namely, given that the reference used is of an interface type, only methods declared in the interface can be accessed. One has to perform a proper casting to another type of the object if he wishes to access other methods of it. Also, when the reference used is of a base class type that did not implement the interface, the methods defined in the interface cannot be accessed through it, and unsafe casting operation will fail.

The complete solution of the questionnaire, including all classes and interfaces involved are presented in figure 2. As shown, Endangered and Vegetarian are the solutions for questions 1 and 2 accordingly. Licensed is implemented by Dog class that adds one attribute to support the implementation of the methods involved. Flying and Hunter are implemented by Hawk class that adds two attributes to support the implementation of the methods of these two interfaces. PharmaExperimenter is implemented by Mouse and Salmon classes. Each of these classes defines different type of attribute to support the implementation of the interface's methods since each provides different implementation as required. Swimmer is implemented by Fish class, since swimming is ability common to all kinds of fish. For that matter Fish class adds one attribute to support the implementation of the methods involved. Pregnant is implemented by Mammal class and Shark classes. Since pregnancy is ability relevant to all mammals, it is implemented in the Mammal class. Since Shark is not a mammal, it must also implement the interface. Both classes add an attribute to support the implementation of the methods involved, and the implementation itself is identical.

IV. RESULTS AND DISCUSSION

In the following discussion we present: (a) informal definitions of interface class as provided by the study participants (the students' responses to part A question); (b) analysis of the students' solutions to the nine questions that were classified to various aspects of interface class (Table 1); (c) analysis of the students' reflections from the informal interviews regarding their perceptions of the interface class.

A. Informal definition of interface

In the first question of part A the students were asked to provide an informal definition of an interface, and specify its possible uses. The results obtained concerning the informal definition of an interface and its possible uses were analysed and categorised into the following categories: correct, semi-correct and incorrect. In this section, we elaborate on these categories:

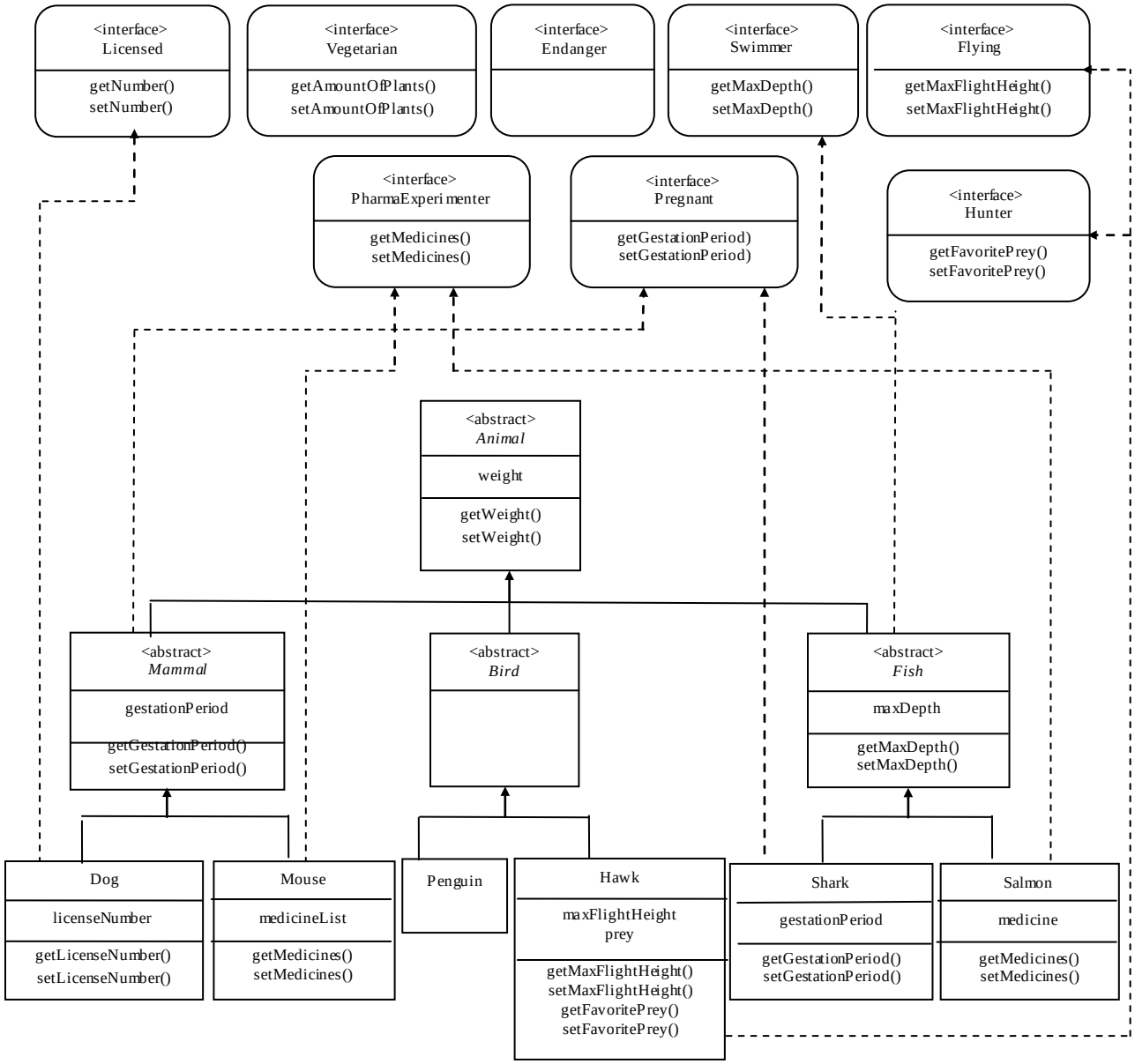

Figure 2: Class hierarchy and interfaces

1) Correct definitions

Four students provided a definition that refers to the most significant aspects of an interface and it uses. The following definition is a variation of the answers provided: (1) *"a special class that has no attributes (beside constants), and may contain abstract methods that other classes which implement this special class must provide an implementation for those methods. A class may apply as many interfaces as needed. Its main purpose is to enforce unity on the classes that apply it, and allow for them to be handled them in the same manner".*

2) Semi-correct definitions

Forty-one students provided either a partial definition of the interface concept or a partial list of possible uses of it. Some definitions did not include the explicit specification that a class may implement multiple interfaces. Some others did not include the explicit specification that multiple classes may implement the same interface, each providing a different implementation. Others did not refer to the possibility of having constant variables as part of an interface.

As for the possible uses of interfaces, most students did not refer to the unity enforced by the interface method's definitions on the classes that apply it. Many others ignored the possibility of handling different classes that implement an interface in the same manner (i.e. using a reference of the interface type). While some indicated that interface is a substitute to the multiple inheritance that was prohibited in Java, others ignored this aspect and did not provide any explanations regarding the need for this kind of construct.

3) Incorrect definitions

Five students provided incorrect definitions. Some students omitted the requirement for the method declared within the interface to be abstract. Others enabled the definitions of variables within the interface. Another incorrect definition referred to an extra restriction concerning the ability of a class to implement one interface at most. These students did not provide any possible uses of interface.

To conclude, the large number of semi-correct and incorrect definitions demonstrates only partial understanding of the interface concept. We may state that the above partial and incorrect definitions stem from several reasons: (a) confusing prior knowledge regarding related concepts such as abstract classes with interfaces; (b) forgetting some of the definition's constituents due to the time that has passed since they were engaged with interfaces; (c) partial understanding and assimilation of the interface concept.

B. Students' solutions and typical mistakes

In what follows we present analysis of the students' solutions according to the aspects specified in Table 1. In Table 2 we present the percentage of correct solutions and in the following subsections we discuss common mistakes provided by the students according to the examined aspects of interface class.

TABLE II: DISTRIBUTION OF CORRECT ANSWERS

Examined aspect	Question no.	No. of correct solutions
Definition	1	50 (79%)
	2	58 (92%)
Implementation	3	57 (90%)
	4	52 (83%)
	5	48 (76%)
Hierarchy	6	45 (71%)
	7	40 (63%)
Polymorphism	8	25 (40%)
	9	26 (41%)

1) Definition

50 out of 63 (79%) provided fully correct answer to the first question. As for the other 13 students, 2 students did not provide any solution, 7 provided a definition of the required interface class but added unnecessary methods. The following is a variation of this solution:

```
interface Endangered {
    boolean isEndangered();
}
```

The students who provided the latter solution were instructed to provide an interface and did so, but they felt obligated to add a method. They did not want to provide an interface with nothing inside it; hence they provided a declaration of a method that returns whether or not the animal is endangered. Obviously, such a method is redundant, as any class that implements Endangered will return true. This may point towards the students' difficulties in understanding that interface may not include methods and yet may be useful for tagging objects.

Four students ignored the explicit instruction to use interface to represent endangered animals, and suggested adding an abstract method to the Animal class that returns true or false regarding whether or not the class represents an endangered species. This method then must be overridden in Animal concrete successors.

58 out of 63 (92%) provided fully correct answer to the second question. As for the other 5 students, 3 of them provided a faulty definition of the required interface class including both methods and the amount attribute which is obviously not allowed in an interface class. Surprisingly, the methods declared remained unimplemented in these faulty solutions. One student provided a partial solution omitting the setAmountOfPlants() method. One student provided a definition of the interface and included a complete implementation of the methods inside the interface.

Obviously, implementing methods and declaring member variables are not feasible within an interface. We may infer that the above students did not understand the concept of interface at all.

According to the above results (questions 1 and 2) it can be concluded that most of the students were able to provide correct answers referring to the aspect of defining an interface class. However, minority of them demonstrated difficulties such as adding attributes to an interface, implementing the methods in the interface class, and partial declaration of the required methods.

2) Implementation

57 out of 63 (90%) provided fully correct answer to the third question. As for the other 6 students, they provided a faulty implementation of the Dog class omitting the licenseNumber attribute. This omission did not prevent them from using such a variable in the implementation of the methods. Maybe if they were using a code editor they would be notified on the problem and correct it.

52 out of 63 (83%) provided fully correct answer to the fourth question. As for the other 11 students, 3 students did not provide any answer, 5 students provided a faulty solution in which the Hawk class declared on the implementation of only one interface of the two required as follows:

```
class Hawk extends Bird implements Flying
or
class Hawk extends Bird implements Hunter
```

These students also omitted the implementation of the interface which was not declared, and implemented only the methods of the declared one. Another student also provided similar solution in which only Flying interface was declared, however, he implemented also the methods required by the Hunter interface, although not declared in the class definition. The other 3 students provided another faulty solution in which they set

Hawk's base class to be of type Flying, instead of type Bird, and added the Hunter in the implements-clause as follows:

```
class Hawk extends Flying implements
Hunter { ... }
```

These errors may point on the fact that these students do not realise that a class may implement multiple interfaces, and that an interface cannot serve as a base class for another class. These mistakes are not only syntactic since they require a high level of understanding of object oriented concepts, which presumably not all students possess. To be precise the students do not consider the importance of the class hierarchy. Omission of the Bird class from being Hawk's base class results in hawks that are not part of the birds' family (or of animals' family, implied).

48 out of 63 (76%) provided fully correct answer to the fifth question. As for the other 15 students: 3 students did not provide any answer; 2 students implemented the methods inside the interface class and added "implemets PharmaExperimenter" clause in the Mouse and Salmon classes; 10 students provided solutions which were erroneously implemented. Namely, they did not distinguish between the different implementations required in the two classes. They provided identical implementations to both.

According to the above results (questions 3-5) it can be concluded that many of the students were able to provide correct answer referring to the aspect of implementing an interface class. However, part of them demonstrated difficulties such as omitting required attributes in the classes that are necessary to support the implementation of the methods, implementing only one interface instead of two required, implementing identical behaviour of the same interface in two different classes although dissimilar implementation was expected. Some of these difficulties can be attributed to the students' habit of relying on the automatic correction of the development environment and hence not paying enough attention to these faults.

3) Hierarchy

45 out of 63 (71%) provided fully correct answer to the sixth question. As for the other 18 students: 4 students did not provide any answer; 14 students implemented the interface in both classes Shark and Salmon.

40 out of 63 (63%) provided fully correct answer to the seventh question. As for the other 23 students: 4 students did not provide any answer; 19 students implemented the Pregnant interface only in the Mammal class. Among them 5 students changed the hierarchy in a way that Shark extends Mammal instead of Fish, while the other 14 students simply ignored the part of the question related to sharks.

According to the above results (questions 6 and 7) it can be concluded that approximately two-thirds of the students were able to provide correct answers referring to the aspect of implementing an interface class in the context of class hierarchy. However, approximately third of them demonstrated difficulties such as implementing the interface in an identical manner at the lowest level of the hierarchy (i.e., the concrete classes) instead of implementing it at a more abstract level (i.e., the common ancestor), changing the hierarchy in a way that classes that wishes to have abilities already implemented in a certain class must extends this class instead of extending their current base class. It should be stressed that in this case the class hierarchy was given to the students and they had to use their knowledge to properly implement interface class in the given context. However it is well known that in cases they had to construct the class hierarchy as well they encounter more difficulties [7].

4) Polymorphism

25 out of 63 (40%) provided fully correct answer to the eighth question. As for the other 38 students: 10 students did not provide any answer, 15 students invoked the getWeight() method without proper casting, as follows:

```
void someMethod(Flying obj){
    System.out.println
      (obj.getMaxFlightHeight());
    System.out.println(obj.getWeight());
}
```

The obj parameter is of type Flying and therefore cannot be used to invoke the getWeight() method. However, the object pointed at by obj may include a getWeight() method, and a casting of obj to type Animal is required in order to access it. 13 students indeed performed a proper casting to type Animal as follows:

```
    System.out.println
      ((Animal)obj.getWeight());
```

However they did not precede a testing operation using the instanceof operator to ensure that the object is of type Animal. Without the testing a runtime error of type ClassCastException can occur. For example, if obj is of type Airplane which implements the Flying interface but is not a successor of Animal, the casting will fail. Avoiding such an error requires a deep understanding of the polymorphism principle in the context of class hierarchies and interfaces. The above results are in line with [5] who found that students have difficulties in understanding polymorphism in general and casting between types in particular.

26 out of 63 (41%) provided fully correct answer to the ninth question. As for the other 39 students: 10 students did not provide any answer, and the other 29 students provided solutions with errors similar to those of question 8 concerning proper casting. 18 students failed to perform a proper casting to the parameter before invoking the getFavouritePrey() method, as follows:


```
void someMethod(Animal obj){
       System.out.println
         (obj.getWeight());
       System.out.println
         (obj.getFavouritePrey());
}
```

Unlike the previous question, the method's parameter is of type Animal, which is the base abstract class of the given hierarchy. Since not all animals are of type Hunter, the object must be checked as being of that type (using the instanceof operator) before one can invoke the getFavouritePrey() method. Accessing the later method also requires a proper casting to Hunter. 11 students performed proper casting to Hunter, however, they did not precede a testing operation using the instanceof operator to ensure that the object is of type Hunter. This is in line with [6] who found that students have difficulties in dynamic binding in the context of proper casting.

According to the above results (questions 8 and 9) it can be concluded that approximately one-third of the students were able to provide correct answers referring to the aspect of polymorphism in the context of interface class. Approximately two-thirds of them demonstrated difficulties such as accessing methods without performing a proper casting to the type that enables desired access, performing unsafe casting from interface to class and vice versa without examining first the type of the object at hand. This is in line with [10], and [6] who found that students have difficulties related to class inheritance and polymorphism.

To summarise, most students demonstrate good understanding concerning the definition and implementation of interface classes. This understanding is demonstrated by their ability to define interface according to the requirements, and implement given interfaces in one or more concrete classes. As a profound understanding of the infrastructure of interface and its relations to principles of polymorphism and class hierarchy is needed, the number of students who demonstrate such an understanding decreases. Specifically, in case that: (a) several interfaces are involved, or some classes need to implement an interface; (b) casting is needed to access methods belong to a specific type; (c) abstract classes are involved or an interface with no methods appears, the number of the correct solutions decreased significantly. These results strengthen the findings resulting from in part A of the questionnaire, in which many students failed to provide good definitions that encompass the various aspects of interfaces. The upshot of such misunderstandings is the students' unsatisfactory results when using, designing and implementing interfaces [1].

C. Students' reflections on the questionnaire

After the students had finished answering the questionnaire we conducted informal semi open interviews with twenty six of them, in which they were asked to provide reflections concerning their performances on the questionnaire. Using analytic induction [20] and content analysis [21] in reviewing the entire corpus of data to identify themes and patterns of the focal points of the study, the students' reflections were classified into the following categories: the essence of interfaces; the complexity embedded in the interfaces versatility; and lack of experience with interface programming. In this section, we elaborate on the students' reflections regarding each of these categories.

1) The essence of interfaces

Some students provided reflections similar to the following, concerning the essence of interfaces:

Dafna: *"I do not use interfaces in my programs, unless I'm specifically required to do so. I do not find it useful, and to tell the truth I never understood what it is good for. If one wants to implement some methods in a class you may do it without employing interfaces, so why bother?"*

Gideon: *"I totally misunderstand the concept of interface. If I want something to be abstract I use abstract classes. In what sense are interfaces better? They do not even allow defining data members!"*

Gadi: *"The combination of interfaces and abstract classes totally confused me. In one of the question I think I was supposed to implement an interface within a given abstract class, and I was not sure if it was legal. Therefore I decided to avoid doing so, and instead I implemented the interface in each class separately. Maybe I redundantly duplicated code, but I'm sure it works."*

The students consider interfaces as not being useful. They confuse them with abstract classes and cannot understand the difference. They cannot think of a problem in which interfaces would be their best solution, and they avoid using it. The above excerpts point towards the students' misunderstanding of the essence of interfaces. As a result of this misunderstanding, the students (a) do not find it useful; (b) do not use it unless forced; (c) bypass employing interfaces by using abstract classes instead. The students' misunderstanding may stem from several reasons: (a) insufficient time dedicated to the learning of interfaces; (b) insufficient exposure to examples that demonstrate the unique advantages of interfaces over other object oriented mechanisms such as abstract classes. (c) lack of continual exposure to interfaces in various courses. The students' misunderstanding concerning interfaces was reported by [1], who found that interfaces are among the most difficult concepts for students to understand when they study object oriented programming.

2) The complexity embedded in the interfaces versatility

Some students provided reflections similar to the following, concerning the complexity embedded in the versatility of interfaces:

Alex: *"The use of the 'instanceof' operator is not difficult to me in the context of a class hierarchy, but I did not remember that it works with interfaces."*

Ruth: *"The casting operations are very difficult to me. The rules are not intuitive. I do not understand why one would use a reference to 'Animal' when you actually want to treat the parameter as 'Flying', or vice versa. The questions concerning the casting were artificial."*

David: *"I couldn't solve the problem concerning the Hawk that is both 'Flying' and 'Hunter' since I did not recall that a class can implement more than one interface."*

Gal: *"The first question regarding the 'empty' interface confused me. I added a method to the interface not because they asked for it, but because I couldn't leave the interface empty. It seems odd to do so, and I even thought it was illegal. Isn't it?"*

The above excerpts refer to the versatility of interfaces, which makes it difficult to apply it properly in various contexts. Alex refers to the polymorphism aspect of interfaces and his difficulties expressed by his inability to tie together the instanceof operator with the interface type. This difficulty may stem from his misunderstanding that implementing an interface is similar to extending a base class. Ruth refers to difficulties stemming from another aspect of working with interfaces. She finds it difficult to perform casting operations required to handle an object through different views. Namely, when an object is accessed by Animal reference she had difficulties in switching the view for that object to Flying (via casting operation) in order to access the getMaxHeightFlight() method. David raises an additional difficulty referring to a third aspect of interface which concerns the possibility of one class implementing multiple interfaces. This difficulty may stem from his faulty analogy to class inheritance in which only one base class is allowed. Gal refers to another aspect of the use of interfaces, which is concerned with tagging objects via interfaces. Tagging is used when one has to distinguish between objects based on some property. A known use for tagging in Java is the Serializable interface which does not contain any method declarations and is aimed at being implemented by classes which permit serialisation of their objects to/from input/output streams. These difficulties are in line with [6] who found that students have difficulties in understanding and applying various issues regarding inheritance and polymorphism.

3) Lack of experience with interface programming

Some students provided reflections similar to the following, concerning their lack of experience with interface programming:

Dorit: *"Interfaces are sophisticated. I think that professional programmers use them, but I'm just a beginner. I cannot think of a situation I would consider using it to solve a problem. Maybe after I gain more experience I'll find it useful".*

Boris: *"I do not remember that we paid much attention to interfaces when we learned object oriented programming. The lecturer explained its purpose, and we even practiced it, but we practiced the use of regular and abstract classes a lot more."*

Ron: *"I do not feel I understand interfaces. I remember we studied it, but I do not remember much. It is probably not so important, otherwise we would use it more often, and I wouldn't forget how it works. I do not remember I ever used it again in successive courses."*

From the above excerpts we can learn about possible explanations why students face difficulties when using interfaces properly. Dorit attributes her difficulties to both the complexity of interfaces and her minor experiences as a programmer. Boris, on the other hand, attributes his difficulties to the small amount of time dedicated to the learning and practicing of interfaces compared to the time dedicated to the learning and practicing of class inheritance. Ron refers to the discontinuity of practicing interfaces in successive courses. As a consequence he forgot how to use it properly, and considers the issue to be less important than other object oriented constructs. No doubt intensive and continual practicing may raise the students' comprehension of the concept under study. It is desirable to practice interfaces as well as other important programming constructs in successive courses, and in stressing the possible interrelations among them. Loftus et al. [22] reached the conclusion that most graduating students cannot design systems properly. Therefore Hu [12] suggests rethinking the pedagogy used to teach object orientation. He raises the following questions: (a) Where, when, and how can inheritance and polymorphism be learned in a truly problem-solving environment? (b) Would it be advantageous for students to learn problem solving with object aggregation before they learn inheritance and polymorphism? (c) Should the interface construct be introduced before class inheritance (thus, students would be "forced" to think in terms of polymorphic implementations of an interface)?

V. CONCLUDING REMARKS AND IMPLICATIONS TO EDUCATION

In this paper, we have presented and analysed the understanding of college students' concerning various aspects of the concept of interface class. The results obtained reveal that most of the study participants understand how to properly define and implement interface classes in concrete classes. Nevertheless, few of the students demonstrated difficulties in defining an interface class with no methods; in implementing multiple interface classes in one concrete method; and implementing a single interface class in different concrete classes. As to understanding of the use of interface classes in a given class hierarchy only two-thirds of them demonstrated a proper understanding. The other third had difficulties in implementing an interface within an abstract class together with separate implementation in additional concrete class. As for the polymorphism aspect, only one third of the student was able to change views of the object from one type to another. The other two-third were accessing methods without performing a proper casting from its 'class' type to its 'interface' type and vice versa. These results are

consistent with previous research regarding the object-oriented design capabilities of novice programmers [6, 7, 23], and regarding the use of interfaces during object oriented design and implementation [10].

There is no doubt that the interface concept enables the programmer to design more flexible and modular computer programs. Nevertheless, the time devoted to it in the MIS curriculum is minor, and as a consequence the students have difficulties in understanding it profoundly, and in using it properly. Usually, this topic is studied towards the end of the second programming course (Object Oriented programming), after learning the concepts of class inheritance, abstract classes and polymorphism. Difficulties in understanding these issues result in difficulties in understanding the interface. Moreover, the students are not exposed enough to interfaces in other contexts other than the objected oriented programming course, and hence tend to underestimate its value. Therefore, to facilitate the students' understanding of the issues involved we suggest the following: (1) incorporate the issue of interface class together with abstract class, emphasizing the similarities and the difference between them; (2) dedicate more time to teach interface classes. The extra time will be devoted to the practice of advanced properties of interface classes in order to raise the students' awareness to their advantages; (3) emphasize the contribution of interface classes to the quality of the code in general and to modularity and flexibility in particular, by providing an example in which one interface (e.g., interface List) can be implemented by two concrete implementations (e.g., class ArrayList and class LinkedList) and can be used interchangeably without further modifications in other parts of the code; and (4) add tasks involving the use of interfaces in advanced programming courses, in order to demonstrate its importance and relevance in other contexts (e.g., data-structures, algorithms, distributed systems, etc).

Furthermore, we recommend on spiral learning [24] of the interface concept. At first, even before introducing the class concept, the interface concept cab be introduced as a general declaration of some capability, including only a collection of related methods without an implementation. Then, the course moves on to classes and their implementation. The next time interfaces can appear is when the polymorphism concept is presented. It can be used as a parameter type to some method, and the students are presented again with the interface concept and its uses. The course moves on to abstract methods and classes, and then the interface class should be represented with compare to the abstract class construct. It is now the time to specify the advantages of using interfaces, specifically their flexibility (i.e., any class can implement them) and their modularity (i.e., any concrete implementation of it will fit in). After that, combinations of interfaces and abstract classes should be presented, and a discussion of the contribution of each construct to the solution should take place. The use of an interface to label classes can be also presented and discussed in a separate lesson. In the following courses (e.g., data structures, distributed systems) the educators should use interfaces in their examples, and require the use of them, this way the students would have a chance to revisit the concept, and to internalize the advantages of using it.

Finally, we believe that further research with a large number of participants should be conducted in order to substantiate our results.

REFERENCES

[1] Hu, C. (2006). When to use an interface? *SIGCSE Bulletin, 38(2)*, 86–90.

[2] Eckerdal, A. & Thune, M. (2005) Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory. In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05). pp 89-93.

[3] Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström J.E., Thomas, L. and Zander, C. (2008) Student understanding of object-oriented programming as expressed in concept maps. In Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08), pp 332-336.

[4] Eckerdal, A. (2009) Novice Programming Students' Learning of Concepts and Practise. PhD thesis, Uppsala University, Sweden.

[5] Benaya, T. & Zur, E. (2008). Understanding object oriented programming concepts in an advanced programming course. In R.T. Mittermeir and M.M. Sysło (Eds.), ISSEP 2008, LNCS 5090 (pp. 161–170). Berlin/Heidelberg: Springer-Verlag.

[6] Liberman, N., Beeri, C. & Ben-David Kolikant, Y. (2011). Difficulties in learning inheritance and polymorphism. ACM Transactions on Computing Education (TOCE), 11(1), pp. 1–23.

[7] Or-Bach, R. & Lavy, I. (2004). Cognitive activities of abstraction in object-orientation: An empirical study. The SIGCSE bulletin, 36(2), 82–85.

[8] Bloom, B. S. (ed.) (1956). Taxonomy of Educational Objectives, the classification of educational goals – Handbook I: Cognitive Domain. New York: McKay.

[9] Biggs, J.B. and Collis, K.F. (1982). Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome). New York: Academic Press.

[10] Lavy, I., Rashkovits, R., & Kouris, R. (2009). Coping with abstraction in object orientation with special focus on interface. The Journal of Computer Science Education, 19(3), 155–177.

[11] Lewis, J., Loftus, W., Struble, C., & Cocking, C. (2003). Java software solutions. Boston: Addison-Wesley Longman.

[12] Hu, C. (2011). When to inherit a type: What we do know and what we might not. ACM Inroads, 2(2), 52–58.

[13] Schmolitzky, A. (2006). Teaching inheritance concepts in Java. In Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java (PPPJ'06) (pp. 203–207).
[14] Topi, H., Valacich, J.S., Wright, R.T., Kaiser, K., Nunamaker, J.F., Sipior, J.C., & De Vreeda, G.J. (2010). IS 2010: Curriculum guidelines for undergraduate degree programs in Information Systems. Communications of AIS, 26, 359–428.
[15] Schmolitzky, A. (2004). Objects first, interfaces next or interfaces before inheritance. Conference on Object Oriented Programming Systems Languages and Applications: 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (pp. 64–67).
[16] Cornelius, B. (2000). Teaching a course on understanding Java. Proceedings of the 4th Java in the Computing Curriculum Conference (JICC 4).
[17] Détienne, F. (2001). Software design – Cognitive aspects. F. Bott (Ed.), Berlin: Springer.
[18] Hadjerrouit, S. (1998). A constructivist framework for integrating the Java paradigm into the undergraduate curriculum. SIGCSE Bull, 30(2), 43–47.
[19] Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C.L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. Interacting with Computers, 11(3), 255–282.
[20] Goetz, J. P. & LeCompte, M. D. (1984). Ethnography and qualitative design in educational research. New York: Academic Press.
[21] Neuendorf, K. (2002). The Content Analysis Guidebook, Thousand Oaks, CA: Sage Publications.
[22] Loftus, C., Thomas, L., & Zander, C. (2011). Can graduating students design: Elaborated. In Proceedings of the 42th SIGCSE Technical Symposium on Computer Science Education, Dallas, TX (pp 105-110).
[23] Sim, E.R., and Wright, G. (2001). The difficulties of learning object-oriented analysis and design: An exploratory study. Journal of Computer Information Systems, 42(4), 95–100.
[24] Harden, R.M. & Stamper, N. (1999). What is a spiral curriculum? Medical Teacher, 21, 2. 141-143.

Prof. Ilana Lavy is an Associate Professor with tenure at the Academic College of Yezreel Valley and is the department head of Management Information Systems since October 2012. Her PhD dissertation (in the Technion) focused on the understanding of basic concepts in elementary number theory. After finishing doctorate, she was a post-Doctoral research fellow at the Education faculty of Haifa University. Her research interests are in the field of pre service and mathematics teachers' professional development as well as the acquisition and understanding of mathematical and computer science concepts. She has published over seventy papers and research reports (part of them is in Hebrew).

Dr. Rami Rashkovits is a Lecturer at the Academic College of Yezreel Valley since 2000 in the department of Management Information Systems. His PhD dissertation (in the Technion) focused on content management in wide-area networks using profiles concerning users' expectations for the time they are willing to wait, and the level of obsolescence they are willing to tolerate. His research interests are in the fields of distributed systems as well as computer sciences education.

