The methodology *Design Patterns First* and inductive learning in interactive mode

Rudolf Pecinovský

ICZ a.s., Prague
University of Economics Prague, Department of Information Technologies rudolf@pecinovsky.cz

ABSTRACT
Our experience shows that girls often prefer to think in inductive rather than deductive manner. This can be a problem especially in programming, where the deductive approach is generally preferred. The paper shows how we can modify the methodology of teaching to satisfy these needs and how to teach Object Oriented Programming in a less abstract way. The approach is based on the Design Patterns First *methodology that suggests starting the course in an interactive mode, where students play a role of an object in a project. The paper presents how we can use the interactive mode for teaching various topics including interface and its purpose in the program and how to follow it by explaining basic design patterns and demonstrating their influence to the overall quality of program.*

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*.

Keywords
Objects First, Design Patterns, Software Engineering Education, Design-Patterns-First, teaching theory.

1. INTRODUCTION

Our school concentrates on education in ICT and IT. From the point of view teaching programming girls comprise an interesting group of our students. Our experience shows that they often prefer to think in inductive rather than deductive manner. They do not like to learn general rules that should be applied to concrete cases. They prefer to see a set of solved examples and then derive a general rule for solution of such examples. They often cannot formulate this rule, but they can apply it. This can be a problem especially in programming, where the deductive approach is generally preferred.

We slightly modified the methodology of teaching to satisfy these needs and teach Object Oriented Programming (OOP) in a less abstract way. Our approach is based on the *Design Patterns First* methodology that suggests starting the course in an interactive mode. We use the *BlueJ* development environment that allows us to concentrate on the program architecture without disturbing students by syntax from the very beginning. One of the big advantages of this IDE is its capability to work in the interactive mode in which students work with an opened program as if they were a part of it, in other words as if they were one of the program's objects and so they were able to send messages to the other objects. The side effect of this interaction with other objects is that students learn to think in the object oriented paradigm. Thus they learn general rules and properties of OOP before writing a first line of code.

Most textbooks that employ *BlueJ* as the starting IDE use the interactive mode only for the demonstration of the basic object properties and behavior and further for quick tests of the developed classes. As you will see, the interactive mode has higher educational potential.

2. OBJECTS, CLASSES AND MESSAGES

In the beginning lesson we introduce a simple project allowing some operations with geometrical shapes (Figure 1). Using this project we explain the concept of objects and classes. Here, at the very beginning, it is quite important to show that objects in the program are not merely representations of real life "objects", but that they may also represent properties, events, activities or other concepts – shortly an object is anything what we can name by a noun.

In our first project students meet the objects representing colours and directions. Simultaneously we explain, that a class is also an object – an object with several special properties explained in [4], e.g. it is the only object that can create new objects – its instances. This interpretation of classes is supported by *BlueJ* that works with object and classes in a very similar way.

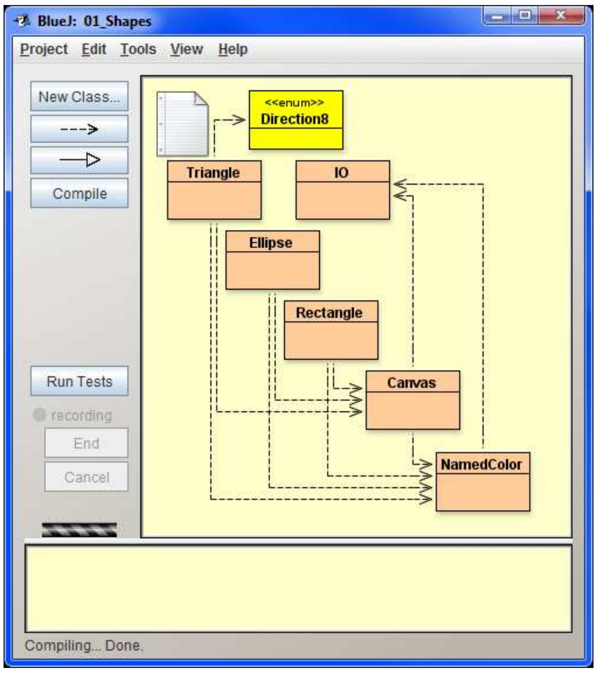

Figure 1: Introductory project

3. CREATING THE TEST CLASS

After the introduction to sending messages we show students how they can save the sequence of the sent messages for a future use. We teach them how to define a new class – the test class. In *BlueJ* the test class has one advantage: we can define its test fixture and test methods in the interactive mode. *BlueJ* tracks our activities. We can ask it for generating a code containing a record of these activities.

When the students know how to create a class and to define its methods, they are able to prepare an assignment. This assignment has the following parts:

1. Creating a new class with the given name derived from the student's name.
2. Defining a test fixture whose objects form a picture.
3. Defining two test methods animating the created picture.

Together with the assignment the students obtain also a program that can check, if their solution fulfills all requirements – e.g. that the picture contain a minimal number of components, that the text fixture be really created in a separate method and not in the test methods and so on. In addition the students can watch several animated solutions on the web and they can use it as hints for creating their own classes. This assignment has the three advantages:

- It is very simple and therefore the students have no problem to prepare it. Almost all students submit it.
- The checking program immediately shows where the student made a mistake, thus the student knows whether the solution contains errors.
- At the first look the assignments have nothing to do with programming. Most students creatively prepare the solution of their own without tempting to copy a solution from one of their colleagues.

A consequence of the third advantage is that the most of the pictures significantly differs one from another. The assignments in the following lessons require some modifications of the submitted class. The differences between particular classes are sufficient to not allow solving the assignment by a simple copying the solution from a colleague.

4. OBJECTS IN DETAIL

We continue with more advanced topics. We show to students how to work with objects, how to pass them as parameters, how to obtain an object as a return value and some other techniques.

We end up this stage with inspecting objects. We introduce attributes as places, where objects save information about their state. Simultaneously we explain the difference among attributes and properties: attributes are internal tools needed for performing the object's task successfully, properties represent information we can get or set. Several properties may correspond with appropriate attributes, whereas other properties are computed. Conversely, values of some attributes may be obtained and/or set through the corresponding properties (getter and setter methods), while other attributes serve for object's internal needs only.

Each explained subject is demonstrated on examples. Students can examine everything on their objects and continuously watch their behavior.

5. INTERFACE

So far the explained subjects have been similar to the subjects explained in the courses using the *Object First* methodology ([1]). Our approach differs in the style of explanation of some subjects, however, the set of the explained subjects is almost the same. Now we start to differ significantly. The methodology *Design Patters First* respects the *Early Bird* pedagogical pattern ([3]) that says: "*Organize the course so that the most important top-*

ics are taught first." Therefore it place the explanation of interface (and following explanation of design patterns) as soon as possible ([7]).

First, we reveal that each object has two faces: its interface and its implementation. The interface describes everything the other objects know about it, the implementation defines how it is made, that the object can do what it can do. (Jak to dělá, že umí to, co umí.) In many situations it is not important, how the particular object ability is fulfilled. The only important information is that the object has this ability, in other words that it understands the particular message.

We explain that Java introduces a special kind of data type – the interface. It declares only the required abilities of its instances but it has no implementation. However for creating an instance something must be done, thus some implementation must exist. This contradiction is solved by implementing the interface by classes. When a class announces that it implements a given interface, it promises that its instances fulfill all the requirements announced by the implemented interface. In such a case the instances of this class can pass oneself off as instances of the implemented interface.

After this theoretical introduction we import a prepared interface **`IShape`** together with the class **`Mover`** into our project. The reason for importing the class **`Mover`** is its methods using instances of the interface **`IShape`** as their parameters (they move them smoothly).

We show that all three graphic classes can response to the messages (they have the methods) declared in the interface **`IShape`**, however until they explicitly implement the interface, the **`Mover`**'s methods do not accept their instances as parameters (at least in Java). As we draw the implementation arrows in the class diagram (see figure 2), the instances start to be acceptable and can be smoothly moved.

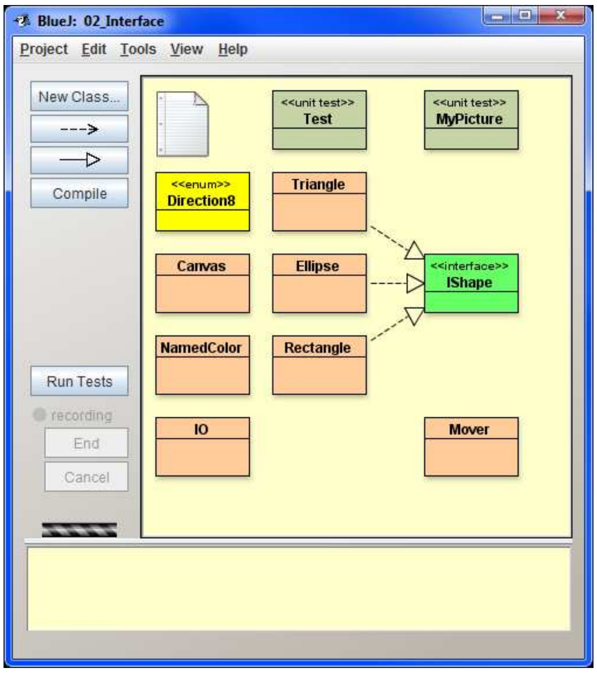

Figure 2: Project after import of the interface and the method

Again we explain everything without writing any line of source code and even without showing any source code. All the time we work only with the class diagram. The interactive mode allows us to demonstrate everything visually and students can immediately do their experiments with the classes and their instances.

6. INHERITANCE OF INTERFACES

We continue with an explanation that the interface **`IShape`** has too high demands towards the implementing classes. The parameters of the **`Mover`**'s methods need not understand all the messages demanded by **`IShape`**. We import a simpler interface **`IMovable`** demanding only methods for getting and setting a position of the object and a new version of the class **`Mover`**, whose methods do not need parameters that are instances of **`IShape`**. Methods of the new **`Mover`** work with instances of interface **`IMovable`**.

Then we explain principles of inheritance of interfaces and show that instances of **`IShape`** comprise actually only a subset of instances of more general interface **`IMovable`**. Therefore we adjust the interface **`IShape`** as a child of the interface **`IMovable`**. Subsequently we import other interfaces and classes and create the inheritance hierarchy of interfaces shown on figure 3. We use each imported type in a special example to exercise the explained subject. It should be emphasized, that we explain here only the inheritance of interfaces, we do not mention anything about inheritance of classes ([6]).

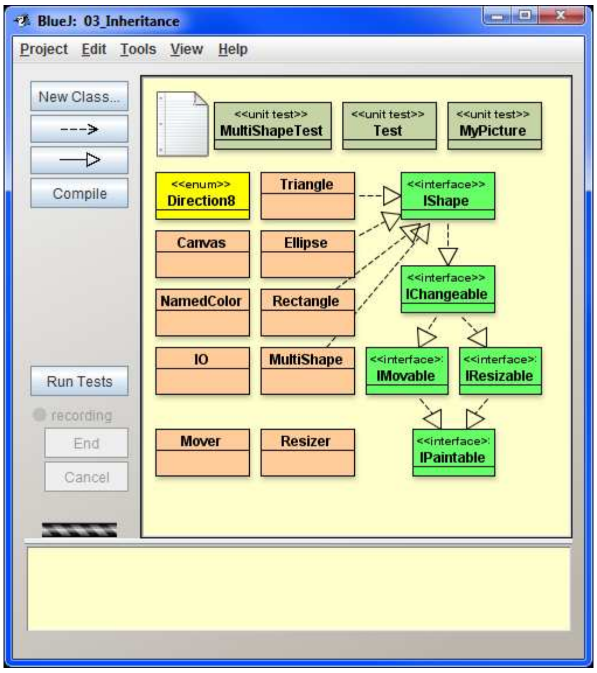

Figure 3: Project after import of the interface and the method

7. DESIGN PATTERNS

Now, the students' knowledge is sufficient for explaining the principles of design patterns and for starting using several of them. We explain the general principles of design patterns and introduce the patterns used in the opened project. Then we point out the disadvantages of the present project: the moved and/or resized objects rub out the underlying and/or overlaying parts of other objects. We describe the reason of this behavior and sketch the necessary steps for its enhancement.

We explain problems connected with an ad hoc solution and introduce the *Mediator* and *Observer* design patterns, which help us to solve the problem in a simple and elegant way. Then, we open a new project, where this solution is employed (figure 4). We import all our classes from the previous project into this project and check, what should be changed to put them to work. After small modifications everything works.

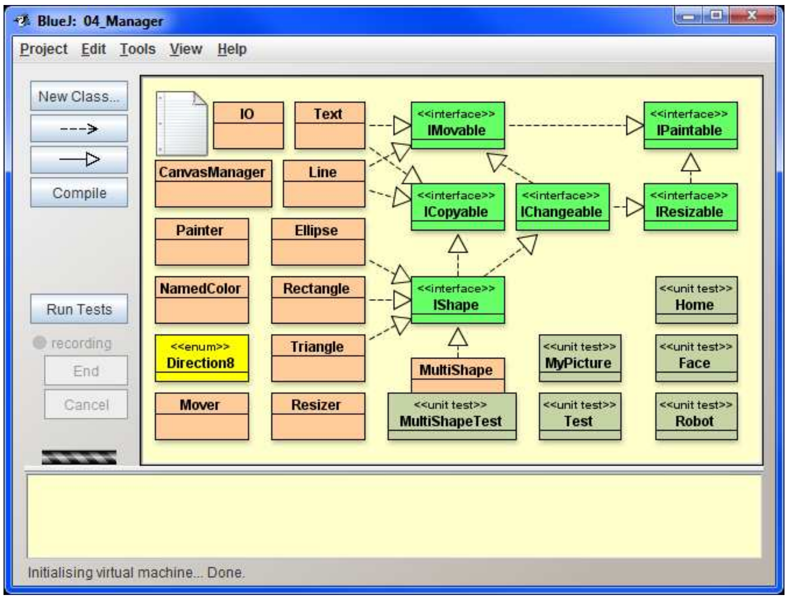

Figure 4: Project using a canvas manager

8. REVISION BY WRITING A CODE

Switching to the project with the `CanvasManager` finishes the first phase of our introductory course. As I have said, we return now to the original project and explain how to write everything we have still made into the code. Students revise everything what they have learned. Now, they know many associations and consequences, so the revision is more effective.

9. CONCLUSION

The paper showed that the interactive mode of the *BlueJ* development environment can be used not only for demonstration of basic objects properties and behavior and for quick preparing the test classes, but that we can use it also for explanation of more advanced features of OOP. It described whole first stage of our courses. In this stage we use only the interactive mode and do not write any line of code. The paper illustrated that the interactive mode allows us to explain clearly even such advanced topics as interface and its intent in the program. As students understand the nature of interface, we can continue with explanation of basic design patterns. We introduce the patterns used in the actual project and show how the behavior of the project can be improved by incorporating further design patterns. By the way students learn basic TDD principles and start understanding the main features of unit tests ([2]).

10. REFERENCES

[1] BARNES, D., KÖLLING, M. *Objects First With Java: A Practical Introduction Using BlueJ (2nd edition)*. Prentice Hall, 2004. ISBN 0-131-24933-9.

[2] BUCHALCEVOVÁ, A. Where in the curriculum is the right place for teaching agile methods? *Proceedings 6th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2008)*. Prague : Copyright, 2008, p. 205–209. ISBN 978-0-7695-3302-5.

[3] BERGIN, Joseph: Fourteen Pedagogical Patterns. *Proceedings of Fifth European Conference on Pattern Languages of Programs.* (EuroPLoP™ 2000) Irsee 2000.

[4] PECINOVSKÝ R.: *How to improve understanding of object constructs using a slightly modified explanation*. http://edu.pecinovsky.cz/papers/2010_DI_How_to_improve_understanding.pdf

[5] PECINOVSKÝ R. Using the methodology Design Patterns First by prototype testing with a user. *Proceedings of IMEM 2009*, Spišská Kapitula.

[6] PECINOVSKÝ R.: Early Introduction of Inheritance Considered Harmful. *Proceedings of Objekty 2009*, Hradec Králové.

[7] PECINOVSKÝ Rudolf, PAVLÍČKOVÁ Jarmila: Order of Explanation Should be Interface – Abstract Classes – Overriding, *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education*, University of Dundee 2007.

[8] PECINOVSKÝ Rudolf, PAVLÍČKOVÁ Jarmila, PAVLÍČEK Luboš: Let's Modify the Objects First Approach into Design Patterns First, *Proceedings of the Eleventh Annual Conference on Innovation and Technology in Computer Science Education*, University of Bologna 2006.