Order of explanation should be Interface – Abstract classes – Overriding

Rudolf Pecinovský Jarmila Pavlíčková
University of Economics Prague
Department of Information Technologies
Churchill sq. 4
130 00 Prague 3
rudolf@pecinovsky.cz pavjar@vse.cz
+420 603 330 090 +420 224 095 460

ABSTRACT
Most textbooks about object-oriented programming start explaining the OOP triumvirate, *inheritance – abstract classes – interface*, with inheritance and overriding, then continues with abstract classes and ends with interfaces and their implementation. The article explains why the *Design Patterns First* methodology changes this order of explanation. It suggests explaining the interface at the beginning of course. After some experience it should follow by explaining abstract classes and inheritance without method overriding and after more experience by explaining inheritance of standard classes and method overriding. The article gives reasons for this changed order, expresses its benefits and accompanies this explanation with some examples of students' assignments demonstrating usage of these principles.

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

General Terms
Design, Languages

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

1. INTRODUCTION
Almost all introductory textbooks on object-oriented programming we have met start the explanation of object-oriented constructions with an explanation of inheritance between standard classes including method overriding. They continue by explaining abstract class as a special sort of class, which can have methods that are only declared and whose implementation is left to child classes. The explanation of these principles ends with introducing the concept of interfaces, which are however, often presented as Java's substitution of C++'s multiple inheritance. The only exception we have met is in [4] (and naturally in [9]), where the explanation starts with interfaces and only then continues with the explanation of inheritance and the principles of method overriding.

This standard order of explanation breaks the *Early Bird* pedagogical pattern ([2]) that says: "*Organize the course so that the most important topics are taught first.*" In addition it complicates the introducing and especially exercising of two important object oriented design principles:

- Programming to an Interface, not an Implementation
- Favour object composition over class inheritance

It is not too wise to start with the explanation of inheritance and after some chapters to continue by suggesting using this construction as seldom as possible and then using it with special care. Moreover everywhere it is possible to favour usage of the interface and composition (not counting textbooks whose authors forget to mention it).

Similarly it is not best to attempt to put the explanation of interface concept near the end of programming part of the textbook[1] and then declare that the usage of interfaces belongs to the main OOP principles. If we want to put some knowledge deep into the student's blood, we should place the explanation of the subject as early as possible to give students enough time and opportunity to absorb it.

2. THE CHANGED ORDER OF SUBJECT
In our courses we modified the standard *Object First* methodology presented e.g. in [1] into slightly different methodology which we name *Design Patterns First* ([6]). This modified methodology orders the explained subjects in such a way as to introduce the key principles as soon as possible and so offer sufficient opportunities to learn, practise and absorb them.

2.1 Preliminary explanations
As we described in [6], we start in a similar way to [1] – this means with interactive experiments with classes and their

[1] We notice that programming textbooks are divided in two parts: the programming part, which explains the basic programming rules, and the technology part, which presents the libraries and its classes and methods.

instances using *BlueJ*. We then continue by introducing basic syntax rules and construction of the first class – the empty class, which means the class with an empty body. This class we step by step equip with methods and attributes.

2.2 Interfaces

Immediately after these first steps with a very simple class we present a problem, which cannot be solved with our present knowledge and which cannot even be solved with the deeper knowledge of advanced structured programmers (for details see [6]).

Then we introduce the concept of interface. We explain two meanings of this word and clarify that **`interface`** in a program is formal notation of general interface. Now we can present the first design pattern: *Servant*[2], which helps us to solve our task.

We continue with the explanation of some design patterns (*Crate/Messenger*, *Observer*, *Simple factory*, *Singleton*, *State*), implementation of several interfaces together, inheritance of interfaces, algorithmic constructs (conditional statements, cycles), collections, arrays and then further design patterns (*Iterator*, *Factory method*, *Bridge*, *Strategy*).

In the introduction to the standard collection library we do not mention the existence of abstract classes. In our explanation we make do with only the relation *interface – implemented class*.

During all these explanations the students meet with classes, which implement interface(s), work with such classes and design such classes of their own. In such a way they have sufficient opportunity to absorb this principal OOP construct and include it into their thinking.

2.2.1 Assignment: Calculator's CPU

This part ends with an assignment to design the CPU for a calculator, whose class-diagram is in Figure 1. Every student has his/her own set of keys, which when pressed should cause his/her CPU to react and answer. The key method declared in interface **`CPU`** obtains a **`String`** argument with the pressed key title and should return the new text on display.

The instance of **`GUI`** should have a constructor with parameter of type **`CPU`** where it expects reference to its CPU. The constructed instance asks the CPU for its key titles and prepares a window with them. Then it repeatedly waits for a key press and asks the CPU what should be written on the display after the press of the given key.

The class **`Version`** is an assignment generator. Its constructor takes as an argument the assignment number and the constructed instance can then return the list of requested keys and the list of test steps.

The class **`TestStep`** behaves as GUI. First it asks the given CPU for its assignment number, then asks class **`Version`** for the appropriate instance and after information from the **`Version`** instance it checks the behaviour of the tested CPU. (For more details about this project see [6].)

[2] *Servant* is design pattern almost identical with *Command*. The only difference is the way we discover that it can solve our problem.

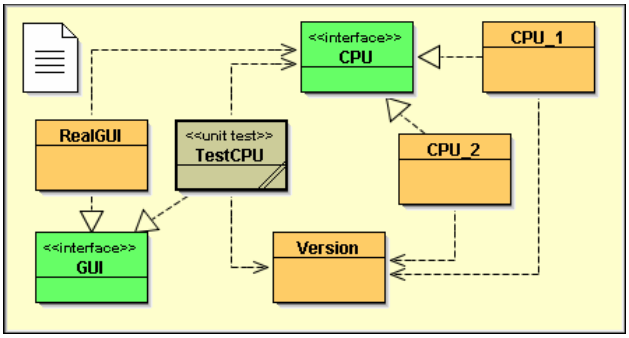

Figure 1: The Calculator

2.3 Abstract classes

In the previous part we introduced some basic classes and interfaces from the collection library. This library is an excellent starting point for explaining abstract classes. We present them as some hybrid between interfaces and standard classes. The type, which wants to have advantage of both: from standard classes it wants the ability to define fields and implement methods, and from interface the ability to declare abstract methods, which need not be implemented.

We can take an abstract class for an interface that wants to offer to its implementing classes some usable implemented methods, which the children will share. Such a class can serve as a special kind of adapter allowing its children to implement only some of the methods requested by implemented interface.

We explain that abstract classes necessarily need some children, because they (similar to interfaces) cannot have any instances of their own. We show, how we can define the classes, which are children of abstract classes. These children can inherit the implemented method in a similar way to that between parent and child interfaces. Instances of child type can offer inherited methods in the same way, as if it declared them itself.

At the same time the child of abstract classes should implement all parent's abstract methods according the same rule, as the class implemented the interface with the same abstract method. From the abstract methods point of view the inheritance from abstract class parent is the same as the implementation of an interface with the same abstract methods – their demands are the same.

From the child-class point of view there is one significant difference: the class can implement an arbitrary number of interfaces, however it can have only one class parent.

We add the information that we can define the child classes not only for abstract classes, but also for standard, concrete classes. The rules are the same without obligation to implement some abstract methods.

Now we introduce the design pattern *Template method* and show some its applications in the collection library. For use in students' next assignment we explain also the design pattern *Command*.

Till now we haven't mentioned the fact that children classes can define their own version of some parent methods. Now we discover that such definitions are possible, however they need deeper understanding of inheritance and therefore we will return to this subject in some future lesson.

2.3.1 Assignment: Adventure game

After all these explanations we continue with the project, in which every student should define its project with an adventure game of his/her own. All these projects should fit into the given common framework (see Figure 2) with the set of abstract classes, whose child the students should define in their projects. Each of these adventure games can be controlled from the common GUI, which is part of the framework.

The work has two steps. The first is to hand in an iterable class, which instances should offer an iterator returning the sequence of instances of class `TestStep`, which is also part of the framework. The obtained sequence should serve as the script defining the author's "artistic intention". By preparation of this script students can utilize the class `GameTest`. One of its methods can simulate playing the game with given test steps, the other tests the prepared game program by the given test steps.

This subtask is useful from several points of view:

- Some students start to plan a very bombastic game, which they are evidently not able to program. Others design the script too simple. The teacher looks at the script and guides both these extremes into the "golden middle way".
- This subtask is the way to force students to separate the design of "game idea" from design of "game program". In the second leg the students have only one goal: to fulfil the handed in test sequence (the script).
- The students again meet several classes implementing the same interface but performing very different tasks. In our framework the classes `GUI` and `GameTest` pass themselves off as GUI, but only the class `GUI` is a real GUI while the key task of the class `GameTest` is to test game or simulate the script.

2.4 Method overriding

In the last period of the introductory course we explain the principle and functioning of method overriding and especially its treachery in some situations (e.g. in constructors). We show, how inheritance spoils the encapsulation and thoroughly explain, where class inheritance should be and where it should not be used.

We return once more to design pattern *Template method* and explain, that it is often usable to define some implicit version of the used methods, which may, but need not be overridden by children.

Thanks to the late inclusion of this subject we can take advantage of the experience of the students and show examples based on programs, which the students have met.

3. OBTAINED BENEFITS

3.1 Better acceptance of OOP principles

Because the students meet the basic design principles almost from beginning and can see their application, they very soon accept these principles into their thinking. It is especially observable by young pupils, because they have no problem accepting new information despite being unable to anchor it into their present knowledge.

Students, who meet the interface in the late part of their course, have problems with the full acceptance of them and with realizing their sense and importance. It is especially true for experienced programmers, for whom the interface is for a long time something strange and who don't like to use it in their programs.

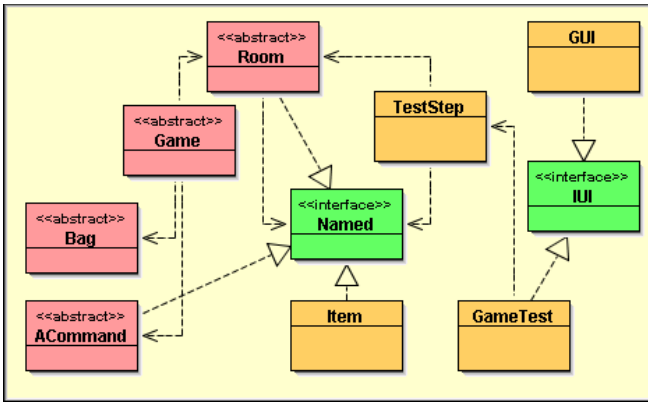

Figure 2: The project adventure

3.2 Design patterns from beginning

Thanks to the early inclusion of interfaces we can introduce design patterns almost from the beginning (the first one is introduced in the second lesson and in the homework from this lesson the students should use it in their program).

During the course students have enough opportunities to meet design patterns in various situations and absorb their spirit. They can discover the usefulness and the advantages of their usage in programs. Most students really learn to use design patterns in their programs and indeed use them continuously.

3.3 Levelling the starting level

The next benefit comes from the structure of student groups. Our experience with teaching of OOP comes from four types of courses in which we teach:

- out-of-school courses for interested children between 12 and 16 years in the Centre for Technical Hobbies Vysehrad (Prague),
- courses of object oriented programming at the High school of Informatics and Management in Prague,
- university introductory courses of programming at the University of Economics Prague,
- courses for professional programmers switching to OOP and Java from structured programming (mostly in Pascal or PHP) organised by Amaio Technologies Inc.

In all these types of courses we meet the same problem: the different entry levels of our students. Some of them are absolute novices; others are experienced structured programmers, who want to convert from classical structured programming to the object oriented one.

We have discovered that the best solution to this problem is to put as soon as possible tasks that are not solvable by simply using the knowledge of classical structured programming. We say that we need as soon as possible to "saw down" the branch of old skills from under the experienced programmers. They then are not able

to try to use their old knowledge to solve the presented problems and their only way to solve them is to use the newly explained object oriented techniques. In such a way we soon bring all our students to a similar level.

One of OOP constructs, which allow us to put the experienced programmers into "another programming cosmos", where they cannot use their old habits, is the interface. When we put them in front of a problem, whose solution is based on usage of an interface, they first try to solve it by the "old good way", and when they discover that this way doesn't lead to the solution, they start to try to solve it by the recently explained OOP way.

3.3.1 Some experience

In courses for professional programmers this switching takes whole first day. When the students discover, that the task is not solvable by known techniques, they feel lost at first. However after penetrating OOP, they are entranced.

3.4 Easier preparation and validation of assignments with many variants

A pleasing consequence of explaining interfaces in the starting lessons is, that it puts into our hands powerful instruments for designing student's assignments. From the very first lessons we can prepare assignments to design classes, which should implement some interfaces. The instances of these classes should then be put as arguments of particular methods, which will use these instances and in such a way they will also test the correctness of handed in solutions.

So that we can give every student a slightly different task despite preparing only one simple framework in which all the tasks should run. On the other side the possibility of incorporating interface into assignments brings also an easier validation of handed in homeworks. We needn't start and test every solution, because we can prepare a simple program, which finds all the classes implementing a given interface test them and return result in some convenient form.

By using interfaces we can always fulfil the rule "*Use large projects*" (presented e.g. in [5]). From the very beginning we explain to our students that their most frequent task in their professional lives will not be designing and developing a completely new program, but it will be acquainting themselves with some complicated project and adding or modifying some feature. This kind of task therefore they should meet during their studies.

4. CONCLUSION

In this paper we have noticed that the classical order of explanation of key object-oriented constructions explaining first the inheritance with overriding, second the abstract classes and finally the interface is not the optimal one. It breaks the *Early Bird* pedagogical pattern and complicates the introduction and especially the exercising of important object oriented design principles.

We have shown the order that is suggested by *Design Patterns First* methodology, and explained why we believe that the suggested order of explanation is much better. We have described some benefits of this approach and we have presented two assignments for our students to solve.

We have verified in practice that using this methodology we can teach even young people from as young as 12 years. These very young students perceive the matter very quickly, mostly with fewer conceptual problems than experienced professional programmers, because they don't need to incorporate the new information into the present knowledge and habits.

This methodology is used in a broad spectrum of courses from the courses for young children through the high school and universities to courses for adult professional programmers. In all the mentioned courses we have better results than we had before applying this methodology.

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

[3] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. ISBN 0-201-30998-X.

[4] HORSTMANN, C.: *Big Java, 2nd Edition*. John Wiley & Sons, Inc., © 2005. ISBN 0-471-69703-9

[5] KÖLLING, M. ROSENBERG, J.: Guidelines for Teaching Object Orientation with Java, *Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001)*, Canterbury, 2001.

[6] 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.

[7] PECINOVSKÝ Rudolf: Aplikace metodiky Design Patterns Fist. *Objekty 2006 – sborník příspěvků desátého ročníku konference*, ČZU, Praha 2006. ISBN 80-213-1568-7.

[8] PECINOVSKÝ Rudolf: Výuka programování podle metodiky Design Patterns First. *Tvorba softwaru 2006 – sborník přednášek*. ISBN 80-248-1082-4.

[9] PECINOVSKÝ, R.: *Myslíme objektově v jazyku Java 5.0* (Object Thinking in Java 5.0), Grada Publishing, 2004. ISBN 80-247-0941-4.

[10] SCHMOLITZKY, A.: "Objects First, Interfaces Next" or Interfaces Before Inheritance. In *Proc. OOPSLA '04 (Companion: Educators' Symposium), (Vancouver, BC, Canada, 2004)*, ACM Press.

[11] SCHMOLITZKY, A.: Teaching Inheritance Concepts with Java. *Proceedings of International Conference on Principles and Practices of Programming In Java (PPPJ 2006)* Manheim, 2006