Let's Modify the Objects-First Approach into Design-Patterns-First

Rudolf Pecinovský
Amaio Technologies Inc.
Education Department
Trebohosticka 14
100 00 Prague 10
+420 603 330 090
rudolf@pecinovsky.cz

Jarmila Pavlíčková
Luboš Pavlíček
University of Economics Prague
Department of Information Technologies
Churchill sq. 4
130 00 Prague 3
+420 224 095 460
pavjar@vse.cz
pavlicek@vse.cz

ABSTRACT
Design patterns have already gained great importance in both design and implementation of object-oriented software in many diverse areas of applications. In order to get the ideas of design patterns firmly established, they should be taught right from the beginning of a course. This paper outlines how the presently used Objects-First approach can be extended and changed into the Design-Patterns-First approach. The outline of the first five lectures of our university course, which is structured according to this approach, is presented. Some examples of design patterns suitable for the very first lecture of such introductory course are also included.

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
When we first experienced the *Objects First* approach to the teaching of computer programming, we were very excited. However, during practical application of this methodology in the classrooms we have soon realized that the main rules of modern OOP, especially the rule to program to an interface, not an implementation and the usage of design patterns, should be taught at the very beginning of a course.

ITiCSE'06, June 26–28, 2006, Bologna, Italy.

This modified sequence of the concepts presented to students of introductory courses was first published in a book [13], written by one of the authors of this paper. This book has quickly gained large acceptance as a textbook for beginners and as a stimulating reading for professionals and enthusiasts. The positive feedback from the readers of this book and from the teachers who have used it in their courses has confirmed to us, that we are going in the right direction.

The first mentioned rule of OOP is to program against the interface and not against the implementation. The Java interface allows us to formalize this rule and it often allows implementing it, too. The explanation of interfaces in our courses has been moving closer and closer to the introductory lectures. In our current course it takes place in the second lecture, just after the first examples of coding.

The second key rule is to use design patterns wherever it is feasible. We have also felt the need to introduce this concept as early as possible in the lecture course. Currently, we teach the design patterns to our students already at the end of the first lecture; even before their first experience with the code!

We have tested this approach in three different types of courses:

- Continuing education classes for students aged 12 to 16.
- Introductory courses of programming at the University of Economics in Prague.
- One- or two-week courses for professional programmers.

These experiences have demonstrated that if we modify the *Objects first* approach into the *Design Patterns First* we will improve the results in teaching of OOP paradigm.

Let us now introduce the outline of the first few lessons of our typical course, taught according to this new methodology. Maybe the presented examples will not be exactly the *Killer Examples* [20]; however in our introductory lectures they have proved themselves very useful.

2. FIRST LESSON

2.1 Introduction of objects
During the first lesson the students discover the concept of objects and classes. We introduce the first project (see Figure 1) to them and then we explain the interactions between instances. Here, at the very beginning, it is quite important to show that objects in

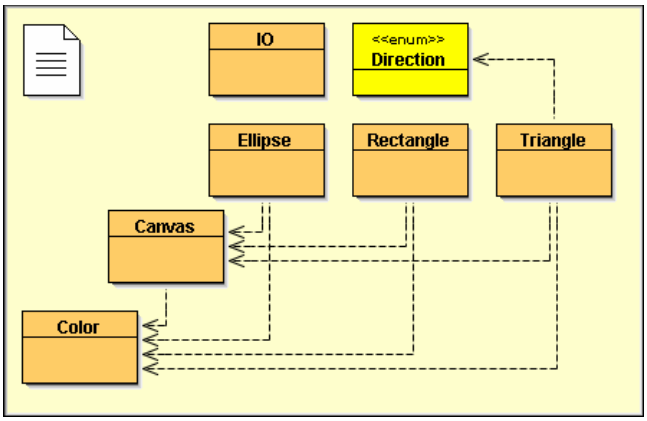

Figure 1: The first project

the program are not merely the representation of real life "objects". In object oriented programming the objects may represent also properties, events, activities or other concepts. Thus in our first example project the students meet the objects representing colours and directions.

2.2 Creating of the Test Class

If we use an appropriate integrated development environment, IDE, e.g. popular *BlueJ*, the students can do something more than just to create the instances of existing classes straight from the beginning. They can define a new class of their own – the unit test class. The *BlueJ* watches their activities and creates programs, which can reproduce them as test methods of this class.

2.3 Introducing of design patterns

During the first lesson we first introduce the design patterns to the students. At this moment we cannot in detail describe their coding, but we can show some tasks which are solved by the use of design patterns.

In our sample project of the geometric shapes we explain that there are classes with different constraints on the number of instances:

- no instance – the auxiliary class `IO`,
- just one instance – the class `Canvas`,
- the known number of known instances – the class `Direction`,
- number of instances controlled by the class itself – the class `Color`,
- arbitrary number of on the fly created instances – the shape classes `Ellipse`, `Rectangle` and `Triangle`.

We explain to our students that the design patterns are programmatic equivalent of mathematical formulae. The programmers who are familiar with them are able to solve the problem much quicker and safer than those who do not know them.

3. SECOND LESSON

The second lesson is devoted to creation of the first handwritten programs and to discussion of a design pattern which we call *Servant* (it is in reality another type of the design pattern *Command*).

3.1 The first textual program

We follow up the initial play with the objects by a construction of an empty class (we name it `Empty`) which we supplement by hand step-by-step, adding the functionality from the first lesson. We write a constructor painting a circle and then we write methods `lightOff()` and `lightOn()`. Now we discover the necessity to define an attribute to remember the colour of this light.

Then we explain that classes should be named according to their purpose and we rename our class simulating a light to `Light`. Finally we add a method `blink()` which switches the light on for a while and then switches it off again.

3.2 The design pattern *Servant*,

Up-to now, there are no significant differences between the new method of teaching and the standard methodology. The differences begin really at this moment. We pose a problem to define a method with a parameter `blink(int)` which would flash the light given number of times. The challenge is that we want to be able to implement this method even though our students at this stage do not know how to write conditional statements or loops.

Some more experienced programmer amongst our students may now suggest solving the problem using a cycle. A teacher would object that such a solution would not be able to flash the lights independently on other actions – for example when we want to move the car whilst the lights are blinking continuously.

Thus we introduce the design pattern *Servant*, based on the following philosophy: When we want some objects to perform a common action, it is often not the best solution to define this action as a method in every class. In many situations it is a more profitable solution to define a specialised class, the *Servant*, which can serve all these objects and provide the common functionality.

To implement its service, the servant requires that the served objects have some abilities. To be served, they have to explicitly declare that they are able to fulfil the requirements of the servant.

3.3 Introducing interface

Now we introduce the interface as a construction, which formalises the servant's requirements placed on the served objects and also a way how the served objects declare that they fulfil these requirements.

Finally we show an example of an interface `Command` with a method `command()` and a servant class `Repeater` with a method `repeat(int,Command)`. Its first parameter means how many times the method `command()` of its second parameter should be executed.

4. THIRD LESSON

In the third lesson we continue to work with interfaces and introduce design patterns *Crate*[1] and *Observer*.

---

[1] The design pattern, which we call *Crate*, is often called *Messenger* (e.g. [6]). We do not use this name, because real life messengers are active and carry their shipments hidden. However our object is passive (it is being sent by one method to another) and its content is publicly visible and therefore accessible for everyone on the way – similar as in the real crate.

4.1 Design pattern *Crate*

We define the method `setPosition(int,int)`, which moves the "light" into a given position.

Now we suggest that we will teach our objects how to continuously move towards a destination position. It is the right task for some servant. We try to describe the requirements that such a servant should place on the served objects and we conclude that the moved object should not only be able to set their final position, but they also should be able report their present position.

When we try to define the method `getPosition()`, we meet a problem that Java methods only allow a single return value. So how can a method return more items? As an answer we introduce a design pattern called *Crate* comprising a class `Position`, an interface `Movable` and a class `Mover` and show how it can continuously move the flashing lights.

4.2 Design pattern *Observer*, event driven programs

By experimenting with the mover we will discover a big disadvantage of the current project: the moving objects erase the objects over which they travel. Therefore we open a new project, in which the present `Canvas` is replaced by a class `ActiveCanvas`. However, `ActiveCanvas` is not the real canvas – it is an active agent managing the correct painting of multiple overlapping objects. Every object which wants to be visible should register itself with `ActiveCanvas`.

`ActiveCanvas` is willing to register only those objects which are able to paint themselves using a given paint-tool[2] on demand. The formal declaration of this demand is the interface `Paintable`. In other words: `ActiveCanvas` is willing to register only objects implementing the interface `Paintable`.

Here we make a note that this mechanism is an implementation of the *Observer* design pattern. We point out that the painted object doesn't know a priory when it will be asked to repaint itself. This action will be made as a response to the need for the `ActiveCanvas` to repaint the whole picture. On this note, we conclude the lesson by explaining that this is the fundamental principle of the event driven programming.

5. FOURTH LESSON

So far we have defined only simple classes performing simple tasks. It is now time to do some more complicated programming. Thus, for a while, we leave the realm of graphic objects and present a familiar application *Calculator*.

5.1 One class – one task

We explain to our students that every class should fulfil only one principal task. Therefore the calculator in our example is not defined in one class, but its definition is divided among two classes:

- the class `GUI` implements the graphical user interface and

[2] Our students have often problems with the term *graphical context*. Therefore we substitute it for the term *paint-tool* and wrap the class `Graphics` into class `PaintTool`.

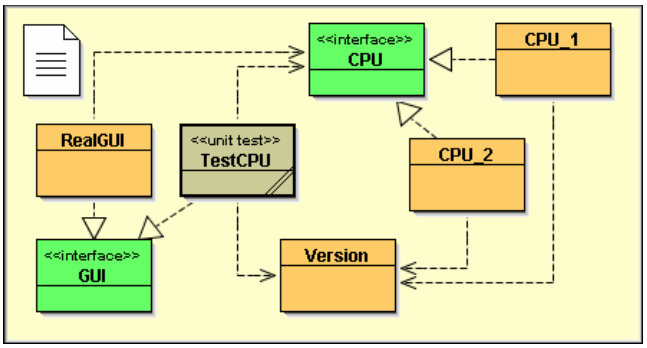

Figure 2: The Calculator

- the class `CPU` is responsible for the computing capabilities of the whole calculator.

5.2 Programming to an interface

After introducing the project we announce that students will receive a ready made class `GUI` and their task will be to create their own class `CPU`. Each student will get his or her own assignment. Because we do not want to prepare individual versions of GUI for every assignment, we prepare one universal GUI, which will cooperate with all CPUs by replacing the original class `CPU` by an interface that each student in the class should implement. This interface declares two methods:

- `public List<String> getOperations()`
  which returns the list of button labels and
- `public String process( String command )`
  which gets the label on the pressed key and returns the new content of the display.

5.3 Design pattern *Bridge*

The GUI comes to know about its CPU from the parameter of its constructor:

```
public GUI( CPU cpu )
```

The GUI remembers given CPU in its attribute and will communicate with the CPU through it.

In such way we reach the independence of GUI on used CPU. We explain students that such solution is described by the design pattern *Bridge*.

5.4 Test class acting as GUI

Next step will show that the CPU class can cooperate with many GUIs. To formalize it we rename our class to `RealGUI`, define the tag interface `GUI` and implement this interface by `RealGUI`. Now we define the unit test class `TestCPU`, which implements this interface, too. We explain that this class will test the correctness of their solutions to save our time and effort.

5.5 Version generating class

To test the student's solutions correctly, the `TestClass` have to discover what the tested assignment is. We introduce the class `Version` (see Figure 2) with two public methods:

- `public List<String> getOperations(int i)`
  which returns the list of button labels of `i`-th assignment and
- `public List<TestStep> getTestSteps(int i)`
  which returns the list of single test steps, where instances of `TestStep` have two attributes: the label of pressed button and the string with the right answer to display.

In addition we extend the interface `CPU` with the method

```
public int getAssignmentNumber()
```

which should return the number of solved assignment.

5.6 Design pattern Strategy

In the final part of this lesson we show that our application can still be improved. We can make it to work in different modes: with real numbers, complex numbers, matrices, vectors, fractions etc. We outline the possible solution of such a task.

6. FIFTH LESSON

After the excursion into calculator application we return back to the realm of graphic objects with an objective to improve our flashing cars. Up to now our cars were able to move only in one direction. We will try to teach them how to turn and continue to move in another direction.

However there is one problem here, our cars move by using the service of the `Mover` objects. But the car must not turn before it reaches the final position. So far the cars were not able to recognize that they have reached the final position without repeated queries about their actual position.

6.1 Inheritance of interfaces

One of the possible solutions is to use the *Observer* design pattern and to ask the observer to tell the car that it has reached its destination position. The problem is that the interfaces for mover and for observer are different. However, we cannot simply add the *Observer's* method to *Movable* interface, because there may be some objects, which don't want to receive these messages.

Now, we explain the interface inheritance and its rules. We define the interface `Multimovable`, instances of which will be movable and will be able to get the message about reaching their destination. Then we extend the mover by adding the possibility to send finishing messages to `Multimovable` objects.

6.2 Design pattern *State*

The cars are ready to turn at the right moment. However, they cannot yet turn. There are two possible solutions to this problem:

- to include multiple `if` or `switch` statements into the implementation of the car class or
- to use some more object oriented solution.

We show to our students that the behaviour of the cars turning in diverse directions differs substantially. This difference is so large that it is reasonable to consider diversely turned cars as instances of different classes.

We define such classes together with the common car class which will represent the cars to the outside world (see Figure 3). Then we explain that this solution implements the design pattern *State*.

6.3 Hiding the details, design pattern Proxy

Next scenario: cars should be running on a track, made from a sequence of segments. The car asks the current segment about the following segment and then it asks the following segment about its start and moves itself to this start. Then the car asks for the direction of this segment, takes an appropriate turn and then continues this sequence over again.

The problem is that the track segments have to cooperate not only with cars, but also with the builder of the roadmap. The builder has other demands on the segments. It wants to create segments, connect them together etc. The cars should not know about these abilities, so that they could not change the roadmap during the race.

One of the possible solutions is to define interface which will publish only the segment's methods intended to be used by the cars and hide the remaining methods. Cars will communicate only with instances of this interface. This interface will serve as a proxy of segments.

7. … AND SO ON

Next lessons continue up in a similar way, we keep introducing new and new design patterns to our students and teach them the rules they should follow when they use them.

8. CONCLUSION

This paper has indicated how to modify the *Objects-First* teaching approach in order to change it into the *Design-Patterns-First* approach. It presented the contents of the first few lessons together with the examples that have been used in the classroom. We have shown that this approach is not only logically possible, but that with an appropriate set of examples, it can infix the object oriented paradigm in the minds of the students very soon and therefore very effectively.

We have verified in practice that using this methodology we can teach even young people from an age as low 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 (see [16]).

We have observed that when this new methodology is used in the courses for the experienced professional programmers converting to OOP, the early introduction of interfaces removes the ground under their feet and therefore they are not seduced to use their previous experience to solve new problems (see [15], [16]). They have no other option than to use the new, object oriented constructs and to start "object oriented thinking". This approach significantly increases the course effectiveness and compresses the time needed for converting the participants to the new paradigm.

University students at our lecture courses have profited from both of these advantages, mentioned above: the beginners obtained deeper skills in an understandable way and those already trained in procedural programming are not able to heavily abuse their old bad habits. The teachers have benefited from the more even levelling of students, because the advanced students soon recognize that they are in the entirely new programming world and don't try to solve the exercises in the old way.

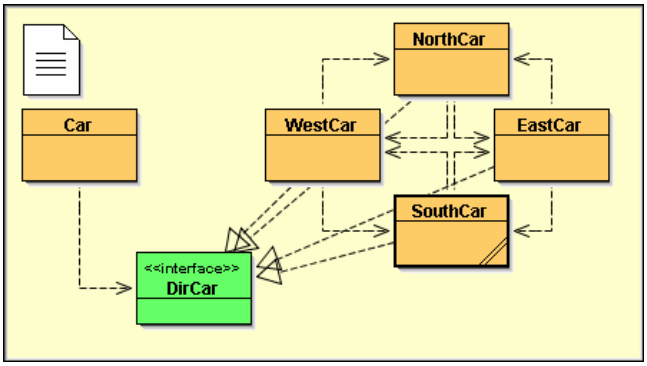

Figure 3: Turning cars

The teaching of computer science at the University of Economics in Prague is focused to analysis and design of business information handling systems. The new methodology equips our students with much more advanced level of basic programming skills and deeper knowledge and understanding of fundamental design principles.

In all these three groups the early introduction of design patterns has allowed teaching of the key principles almost from the onset of the course. The students "live" in the "design pattern environment" during the entire course and these principles become firmly established in their minds.

9. REFERENCES

[1] Astrachan, O., Geoffrey B., Landon C., Garrett M. *Design Patterns: An Essential Component of CS Curricula*, ACMSIGCSE Bulletin v 30 n 1, 153-160.

[2] Alphonce, C., Ventura, P. Object-Orientation in CS1-CS2 by Design, *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002)*, pages 70-74.

[3] Alphonce, C., Ventura, P. Using graphics to support the teaching of fundamental object-orientation principles in CS1, 2003. *Proceedings companion of the 18th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, ACM Press.

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

[5] *Computing Curricula 2001, Computer Science Volume.* http://www.sigcse.org/cc2001/

[6] Eckel, B. *Thinking in Patterns*. http://www.bruceeckel.com

[7] Freeman, E., Freeman, E. *Head First Design Patterns*. O'Reilly, 2004. ISBN 0-596-00712-4.

[8] Horstmann, C. *Object-Oriented Design & Patterns*. John Wiley & Sons, Inc., 2004. ISBN 0-471-74487-5

[9] Lewis, T. L., Rosson, M. B., Pérez-Quinones, M. A. What do the experts say? – Teaching Introductory Design from an Expert's Perspective, *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, March 03-07, 2004, Norfolk, Virginia, USA

[10] Metsker, S. J. *Design Patterns Java Workbook*. Addison Wesley, 2002. ISBN 0-201-74397-3.

[11] Nevison, Ch. Wells, B. Teaching Objects Early and Design Patterns in Java Using Case Studies, *Proceedings of the 8th Annual Conference on Innovation and Technology in computer Science Education* (ITiCSE 2003).

[12] Dung Nguyen, Wong, S. Design Patterns: Pedagogical Foundations For Object-Orientation. A workshop presented at *University of Wisconsin System Computer Science Education Workshop*, University of Wisconsin, October 13, 2000. http://exciton.cs.oberlin.edu/uwisconsin/

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

[14] Pecinovský, R. Začlenění návrhových vzorů do výuky programování (Incorporating design patterns into programming pedagogy). *Proceedings of the conference Objects 2005*, pages 26 – 42. VŠB – Technical University of Ostrava, 2005. ISBN 80-248-0595-2.

[15] Pecinovský, R. Jak efektivně učit OOP (How to teach OOP effectively). *Proceedings of the conference Software Development 2005*, pages 174 – 182. VŠB – Technical University of Ostrava, 2005. ISBN 80-86840-14-X.

[16] Pecinovský, R.: Jak při výuce Javy opravdu začít s objekty (How to really start with objects by teaching Java). *Proceedings of the conference Objects 2004*, pages 241 – 259. Czech University of Agriculture, Prague, 2004. ISBN 80-248-0672-X.

[17] Shalloway, A., Trott, J. A. *Design Patterns Explained – A new Perspective on Object-Oriented Design (2nd edition)*. Addison-Wesley, 2004. ISBN 0-321-24714-0.

[18] Ventura, P., Alphonce, C. Teaching OOD and OOP through Java and UML in CS1 and CS2. Position paper presented at the *Fifth Workshop on Pedagogies and Tools for Assimilating Object Oriented Concepts*, October 2001. http://www.cs.umu.se/~jubo/Meetings/OOPSLA01/Contributions/PVentura.html

[19] Wick, M. R. Kaleidoscope: Using design patterns in CS1. *Proceedings of the Thirty–Second SIGCSE Technical Symposium on Computer Science Education*, pages 258–262, 2001

[20] *4th "Killer Examples" for Design Patterns and Objects First workshop* http://www.cse.buffalo.edu/faculty/alphonce/KillerExamples/OOPSLA2005/