Development of Programming Textbooks and Courses

Rudolf Pecinovský

ICZ plc, Na Hřebenech II 1817, 140 00 Praha 4
High School of Economics Praha, 4 sq. of W. Churchill, 130 67 Praha 3
rudolf@pecinovsky.cz

Abstract. The paper analyses various approaches to the development of textbooks and courses of programming or programming languages. It introduces their advantages and disadvantages and shows where is the difference between real programming textbooks and courses and the textbooks and courses that only present a programming language and its libraries. The paper specifies certain principles that should be observed in concept proposals of programming courses and textbooks. It reminds the early bird pedagogical rule according to which the lecture should be organized in such a way that the most important principles would be explained as soon as possible. At the same time it suggests not to concentrate unnecessarily on topics that are gradually automated but on the other hand, to teach from the very beginning also the basic architectural principles. At the conclusion it introduces various approaches to the development of accompanying programs and analyses their pros and cons. It introduces simultaneously a program that can significantly help in designing the accompanying programs which can be gradually improved by students during the course.

1 Introduction

It is generally known that the living languages and programming languages have much in common. If you want to learn a living language, you can come to a bookstore where a textbook of the given language, its grammar and bilingual vocabularies are sold. If you meet a textbook teaching how to write in the given language, it is mostly a textbook of business letter writing.

Naturally there are also textbooks that would really teach you to write – we can designate them as textbooks for future writers. However, these textbooks do not try to teach you writing in a particular language, but on the contrary they suppose you already know the language, and you need to learn "sophisticated writing". These "writing textbooks" mostly concentrate on a particular kind of text. Some of them try to explain you how to write professional guides/documentation, the others teach you to write screenplays or theatre dramas.

All the above mentioned kinds of living language textbooks have one common: already in their title they clearly declare what they want to teach you and they really try to do it. Unfortunately, it is completely different situation in programming textbooks and courses. If we have a look into a bookstore´s department of computer textbooks, we surely find there titles like "*Programming in Xyz*". But if we start to analyze their content, we discover that most of them do not teach programming itself. We could

call them textbooks of coding in particular programming language or simply textbooks of relevant programming language. Converted into the terminology of living languages textbooks, they could be called textbooks of grammar, including vocabularies, or simply language textbooks. Only exceptionally we could call them programming textbooks corresponding with the above mentioned writing textbooks.

This paper deals with basic features, which the textbook should have, so that we could call it a textbook of programming. At this occasion it introduces a tool that simplifies the development of such a textbook.

Also the courses that present themselves as courses of programming should have the same features. Anything that will be said concerning the textbooks is then equal/valid also for the courses.

2 Inappropriate features of programming textbooks and courses

If we remain at the previously mentioned parallel, we could liken the programming textbooks to textbooks of writing. Similarly we could liken various programming paradigms to different author styles or genres.

If we decide to create a textbook of writing, we certainly will not concentrate on details. We shall mention the grammatical rules only marginally and when we shall mention them, we surely will explain how they can help us in reaching the desired effect. Similarly we shall deal with the vocabulary and other "detailed" features of the language.

Contrary to it in textbooks of writing we will concentrate on the overall conception of the created work and how to reach the required effect. The future detective stories authors will learn how to increase the reader's excitement, how information that seems meaningless at first, but appears significant at the end, can be included into the text appropriately and further similar skills. It will be analogous with other types of literature.

Textbook conception is in contradiction to the declared principles

Let us now have a look at textbooks that proclaims themselves as textbooks of programming. We find out that they mostly concentrate just on details and try to show to the reader, how to fold up simple programs from separate detailed statements. They explain what the variables and their characteristics are like. Then they continue with explanation of particular programming constructs (statements, decisions, loops …) and teach the reader to make out simple methods. In case of textbooks of object oriented languages they finally explain the term of a class and show, how each class defines its own set of methods, alternatively how several classes can share some of their methods thanks to their inheritance.

Unfortunately, only exceptionally these textbooks rise above this basic level and try to explain something about the program architecture as well as principles if its design to the reader. This is left for advanced courses textbooks or they suppose the reader will learn it as a side effect. However, majority of readers will not read further textbooks of advanced programming and will rely on self-study through designing

and developing their own projects learning from their own mistakes and discovering the rules that their teachers (or books) should explain at the very beginning.

Let us learn from history and look how much time and effort was needed before the best programmers discovered such rules that consequently start to appear in advanced textbooks. But in fact, a majority of these rules belongs just into textbooks for beginners and more over into their introductory lessons.

Textbooks, which do not want to stay only at explanation of coding, often clarify that mostly the best way of solving big projects is the top-down design, when the basic goal is defined, divided into particular sub-goals. If a sub-goal is still complicated, we recursively apply the same rule further on. Finally we obtain a set of simple goals that are easy to be programmed.

However, it is only a theory for numerous readers, because their textbook demonstrated them just the reverse procedure: how to build the bigger complexes from simple components. They would need far more examples as well as time to absorb the top-down procedure, but this is usually not offered, because if these principles are explained, it is only within the closing summary framework.

Obstacles in learning new knowledge and skills

The above described method is even more improper due to the fact, that as soon as we complete the teens-age, we are not able any more to embrace new information without subconscious harmonizing it with previously accepted facts. If the new information does not fully correspond with our present knowledge, we subconsciously modify it so that it would get into accordance.

This fact often slows down the mastering of higher level abstraction in case the students did not thoroughly master the lower levels. As soon as their present knowledge and experience indicate similarity of the new information with the previously accepted one, the subconscious mind immediately modifies the new information to allow its simple interpretation through the old one. They do not realize that in fact they remember a little different information than the one offered by the teacher.

This subconscious modification of the received information is as more intensive, as the student's experience is deeper. Thus, the beginning programmer can learn new skills much easier than an old practitioner. That´s why a conversion of experienced programmers to a new paradigm is very often only formal one: the programmers start to use a new language, however their architectural thinking continues in routine rails. It was accurately expressed in a paper [4], saying that "*Real Programmer can write FORTRAN programs in any language*".

3 How to compose the programming textbooks and courses

In [1] we can find 14 pedagogical patterns that we should follow during development of our books and courses. The first of them (and one of the most important) is the *Early bird pattern* saying "*Organize the course so that the most important topics are taught first. Teach the most important material, the "big ideas", first (and often). When this seems impossible, teach the most important material as early as possible."*

If we want to teach our students (and readers) to think about a program in abstract terms, without any deliberations on some secondary details, we have to teach them this method from the very beginning, not at the textbook or course end.

Each time we notice that the understanding of an important principle might be impacted by a previous experience, we should re-organize the explanation in such a way, so that this important principle would be presented at first and only after that we should explain features that could – if presented prematurely – influence accepting of this key principle.

Let´s illustrate this situation at two examples:

Example 1: Design pattern *State*

Supposing we would like to show to our students that in case the behavior of an object differs depending on its state, then it is often more useful to define an independent class for each state describing the behavior of an object in particular state. The behavior of the whole multistate object is then described by a class which is defined according to the design pattern *State* and cooperates with the above described one-state classes.

If we introduce this design pattern after the students have mastered the conditional statements and switches, the students would prefer the non-object solution. It would seem to them as more simple and natural. Therefore they would prefer it even in cases, when using the *State* design pattern would be much suitable.

If we introduce this design pattern before we explain the conditional statements and switches, for certain time it may be the only solution for them, how to realize the needed decision-making. After they learn using the conditional statements, they will already have the design pattern *State* deep-rooted and will not consider it so eccentric. They will know already its advantages and they will be able to pick up the best solution at the particular case.

Example 2: Inheritance

There is a similar situation with inheritance. Too early explanation of inheritance may block (in some measure) a future acceptance of alternative solution of certain problems. It is better to introduce first the *Decorator* design pattern and demonstrate its usage at examples, where it is considerably better to use it compared to using of inheritance. When the students become acquainted with inheritance later on, they have already experienced with several projects using *Decorator* pattern and they will be less susceptible to inappropriate usage of inheritance.

In addition, the presentation of inheritance based on the knowledge of *Decorator* design patterns is mostly much easier to understand, because the basic principles of inheritance are explained on already-known constructions.

Early explanation of the architecture

We also could apply the early bird pattern in more general way. If we observe the evolution of development tools for programmers, we have to note how often the tools appear that are able to automate various actions. In addition, new programming lan-

guages appear with new powerful statements performing functions, listing of which took a whole page in earlier times.

Respecting these trends we should not back up teaching skills that are successively performed by various tools. We should rather concentrate on instilling such knowledge and skills that students can apply in architecture designs. We should incorporate their explanation as soon as possible in textbooks and courses, better said: they should be organized in such a way, so that we could present these principles at the soonest time.

It shows that the explanations should not be based on simple AHA examples, but that already from the beginning of the course the students should work on some relatively large project that will be subsequently supplemented and – if need be – modified. That might be the way how the students learn working in the regime similar to that one which they meet in their future practice.

5. Accompanying programs

The accompanying programs represent a special chapter. Many textbooks (e.g. [3]) and courses limit themselves to simple AHA programs, from which the students grasp the basic function of the demonstrated construct, however they often do not get a reasonable idea how to use it in a program. Therefore such programs are suitable only for students, who need only to learn working in a new programming language, because they already know programming as well as the presented paradigm and thus they are able to derive how to use the explained constructs in a developed program.

Contrary to it, other textbooks (e.g. [2]) prefer using of relatively complicated practical programs. Their students can see how to incorporate the explained constructs in a broad context of the demonstration program, but, on the other side, to understand the program requires great effort. This type of programs is therefore suitable for students, who already know programming basics and need to learn only working with new libraries and frameworks.

However, there are students remained, who are only entering the programming world. These students need programs that are sufficiently simple so that they should not get through a mass of surrounding “noise”, but on the other side sufficiently complex, so that they could see how to use the explained construct in practice. In proposing these accompanying programs, we can set out in two directions:

Separate programs for each explained topic

From the author´s point of view, the simplest way is to demonstrate the usage of each explained construct in a separate program. The problem is that the accompanying programs will be really very simple especially in starting lessons, which means they will be mostly AHA examples. Due to the fact that those are programs from starting chapters, the students will have enough opportunities to meet the presented construct in further programs and realize how to use it.

Gradually improved programs

From the student´s point of view it is much better when the lesson is demonstrated on smaller amount of gradually improved programs. When studying the usage of a new construct the student does not have to analyze the surrounding program in detail, because he/she knows it from previous lessons. He/she can work with relatively complex programs without being bothered by a "noise" of the surrounding code.

The second approach is better because besides presented constructs it enables to teach principles that are important in the development and subsequent management of real projects: code transparency, repeated using of already written code and automatic tests, using constructs facilitating future modification and management as well as other similar principles.

This approach has an unpleasant feature, i.e. that for each program part we should have as much source files as separate forms of the relevant part exist during the whole development of the project and that we should permanently verify their mutual consistence. In case we would realize in certain later project phase, that we should explain certain topics in a different way in some previous lessons, we would have properly modify a bigger number of source files and also check the mutual consistency. Simultaneously we should avoid using a construct that was still not presented.

Fortunately this problem can be solved by a program. To give you an example of a program that can help us: there is a *Cumulant* program ([5]) whose purpose is to support the projects definition that are consequently improved and cumulate further functions. This program defines special preprocessor comment statements that allow specifying more versions of destination class in one source code. In addition we can define a special file, specifying which of the source files should be (after an appropriate conversion) incorporated in particular project defining particular phase of the gradually developed program.

6. Summary/Conclusion

This paper analyzed different attempts how to write textbooks or design the courses of programming and programming languages. It divided the textbooks and courses on those which concentrate on teaching the language and its libraries, and those, which really tries to teach programming. Then it concentrated on books and courses trying to teach programming.

It specified some principles for this group of books and courses that should be respected in designing their content. It suggested to keep the *Early bird* pedagogical pattern and organize the explanation in such a way so that the most important topics would be presented as soon as possible. At the same time it recommended to skip over useless topics, the usage of which is progressively substituted by various automated systems and on the contrary, concentrate on basic architectural rules at a very beginning.

At the end the paper introduced different approaches to developing of accompanying programs and discussed their pros and cons. At the same time it drew an attention

to a program that can significantly help in designing the accompanying programs which can be gradually improved by students during the course.

Acknowledgment

This work has been created with a kind support of the grant *The Research of methodology how to teach programming and its possible improvement* announced by the RPF foundation.

References

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

[2] DEITEL Harvey M., DEITEL Paul J.: *Java: How to Program, 8th Edition.* Prentice Hall 2009, ISBN 978-0-136-05306-4.

[3] ECKEL Bruce: *Thinking in Java (4th Edition).* Prentice Hall 2006, ISBN 978-0-131-05306-4.

[4] POST Ed: *Real Programmers Don't Use Pascal*, Datamation 1983, For downoad at http://www.ee.ryerson.ca/~elf/hack/realmen.html.

[5] PECINOVSKÝ Rudolf: *Cumulant – Assistant supporting creation of teaching project stepwise cumulating functionality of the developed program.* Proceeding of Objects 2011. ISBN 978-80-554-0432-5.