Programming as a part of the Software Engineering education

Maksimenkova Olga
State University Higher School of Economics
Moscow, Russia
e-mail: omaksimenkova@hse.ru

Vadim Podbelskiy
State University Higher School of Economics
Moscow, Russia
e-mail: vpodbelskiy@hse.ru

Abstract — **Programming for the first-year undergraduates starts as a part of "Computer science" academic subject. Some traditional methods of teaching programming are popular in higher education in Russia nowadays. A different method, which is used as a part of Software Engineering education in our University, is described in this article.**

Keywords: software engineering; software engineering education; computer science

I. INTRODUCTION

Software engineering is a modern and complicated field of knowledge. Following the SWEBOK it contains such knowledge areas as software requirements, software design, software construction, software engineering process and so on. A place of our teaching interests laying in the software construction part. The term software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging [1]. As we can see coding is associated with the other fields of knowledge. In any training course, which is connected with software engineering, a teacher should provide and explain this connection at any stage of teaching.

The academic subject "Computer science" within the bounds of Software engineering education is a significant brick to train specialists in this field.

II. STRUCTURED PROGRAMMING LANGUAGES AS A BASE

"Computer science" as a course is traditionally widely connected with coding, algorithms and data structures [2]. Traditional higher education in Russia supposes one of the structured programming languages such C or Pascal to be taught to the first-year students.

Of course, such languages as C play a significant part in further education, because of that the most of modern programming languages (Java, C++, C# etc.) are based on theirs syntax. Some students, however, have an experience in structured programming from schools and special lyceums, but they should repeat lots of basic concepts at first time. Therefore, they lose interest to the main course and, for example, teachers need to prepare problems for all and for the advanced listeners to keep their interest. If advanced listeners had to solve easy problems in the beginning they couldn't have concentrated further on the new for them concepts of course.

III. OBJECT-ORIENTED PROGRAMMING LANGUAGES AS A BASE

The main idea of our course is to teach first-year students to use theory in the field of information technologies and programming skills to describe algorithms using one of the up-to-date programming languages. Graduating student should have "live" knowledge, which he can successfully use in his day-to-day activity.

Object-oriented technology is widely used in software design and development nowadays. So it should have been quite reasonable to use one of modern object-oriented programming languages as a base for course. That's why we chose C# for our first-year students. C# specification supposed C# to be a simple, modern, general-purpose, object-oriented programming language [3].

IV. TARGET AUDIENCE

"Computer science" is lectured during an academic year to the first-year undergraduates. The most of our students come to the University just after graduation from secondary schools, so we expect basic school knowledge in the field of Math and Computer science. In reality, only mathematical skills are more or less equal. As for basic skills of our students in computer science, they are quite different. Mainly just because of absence government curriculum for the Computer science in schools. Some of our students, for example, lyceums-graduates or programming competition winners have an experience in programming. The others have basic school knowledge or haven't got any special knowledge in "Computer science" at all. Anyway, each other should be involved and should be given a chance to be a success.

V. THEORY AND PRACTICE

The whole course of studies consists of two widely connected parts. First of them is the course of lectures and the second one is practical training.

Course of lectures is given to students during an academic year, which is approximately 86 academic hours, as it postulates in the curriculum. Students also take practical trainings, which are given during 88 academic hours following the curriculum.

Curriculum also specifies self-instruction during 204 academic hours for the "Computer science" course.

Course of lectures contains not only theoretical material but general practical examples with significant algorithms

and data-structures, as well. Theoretical concepts of structured programming and basic rules of object-oriented programming are also included into the course of lectures.

Practical trainings consist of set of C# examples and a list of problems to self-instruction. Theoretical material from the course of lectures meets its practical application on the practical trainings. Students consolidate knowledge and gain programming experience during these lessons.

We use multimedia means to represents material and make our lessons more interactive and efficient. Materials of practical trainings with a home task are sent to the students every week after lessons.

VI. TEST CHECKS

Test checks are represented by class written tests, home written tests, computer-based module-tests and yearly project.

Written tests involve one or two short problems to be done using Microsoft Visual Studio 20xx at class. Students write their programs for 40-60 minutes, it depends on difficulty of an introduced written test. We appraise written tests using special criteria. In them such characteristics as correspondence of a program with a task, functionality of a program, its failure stability and so on are taken into consideration. So, students not only take skills in programming and algorithmization, but learn basic maxims of development like constructing functionality programs in limited time.

Home written test is unassisted work of students. In which they should implement a program and provide it with simple documentation. Variant of a home written test is more complicated as opposed to class written test variant. It can contain, for example, wider class structure, more difficult collaboration between classes, user's interface design and data processing layer development. Results, which are reached during the home work, are a great help for students in yearly project (see "Yearly project" topic).

Our school year is divided into five modules – 7 weeks each. So students take a computer-based module-test five times a year during a test period. Each test consists of 30-40 questions of the different types. Usually we use multiple choice questions, multiple response questions and short-answer questions. Students are tested during 40 minutes. Topics of the test problems involve theoretical and practical questions of a current module.

VII. YEARLY PROJECT

Yearly project isn't completely included into the academic subject "Computer science". But it should represent a working application with a documentation package. In yearly project our University carries out an interdisciplinary approach in education. Such a project, obviously, do a lot of good for our students, because they gain not only programming experience, but obtain experience in such fields of software engineering as software requirements, software design, software construction, etc.

Within a "Computer science" academic subject we offer our student a simple method of how to develop a program easily. This method has grown up from an Agile software development methodology, to be more precise, from Extreme Programming. Extreme Programming, XP for short, is an Agile software development methodology that is made up of a collection of core values, principles, and practices that provide a highly efficient and effective means of developing software [4].

For sure, we didn't have a goal to teach first year students such complicated methodologies. We only used some main ideas of them to answer a hidden question: "How to develop a yearly question?" We provided our first-year undergraduates with a method which helps them to develop a huge application without lots of mistakes. We for ourselves called this method "Evolutional approach".

VIII. EVOLUTIONAL APPROACH

Every beginner programmer, like our first-year undergraduates, has a variety of troubles in making programs. Efficient but complicated software development methodologies are not suitable for him. Troubles in development may lie even in choosing algorithm or standard library. Sometimes, beginner programmer starts to develop an application without realizing what kind of means or structures will be needed for its implementation. Each beginner programmer is a software architect, developer and tester at the same time. Our main task, as tutors and teachers, is to help beginner programmers to solve all the problems in a way of implementation their applications.

Evolutional approach, following the traditional software development methodologies, gives next recommendations:

1. Development should be carried out by steps, each of those contains: design (including interface design), coding (including syntactic and semantic debugging) and testing.
2. Stages of development informally divide into two types: research stages and development stages, which give software new functionality according to a requirements specification.
3. In spite of type, the first stage provides creation of working software with the minimum functionality.
4. The analysis of a current version of software is carried out at the end of the each stage of development. A goal of analysis is to assess possibility of adding constructions for implementation new, additional requirements according to a requirement specification.
5. On basis of analysis results refactoring of a current version is taken place or project decisions are accepted, coding and testing are carried out.
6. Software, staying in up state, obtains new, or changes current functionality at the end of each stage.

Thereby, software every time stays in up state, and changes makes permanently in it. In the other words, software is evolving from step to step and its functionality is growing from minimum to wishful (given in requirement specification).

IX. MARKS

It should be said that in our University we use ten marks from 1 to 10. Marks less than 4 are unsatisfactory, 4 and 5 are satisfactory, 6 and 7 are good, 8, 9 and 10 are excellent marks.

Modular and total marks are accumulative. They are calculated using special weighting coefficients for each current mark. As far as all the marks are quite complicated we are giving, as following, formulas we have used this year:

A. *Module 1, 2, 5, current marks:*

$$M_i = min(\kappa_i, \tau_i), \text{ if } \kappa_i < 4 \text{ or } \tau_i < 4,$$

else

$$M_i = 0.5 * \kappa_i + 0.5 * \tau_i,$$

there i =1, 2, 5 – number of modules; κ – written test mark, τ – computer-based test mark.

B. *Total mark after module 2:*

$$\Theta_2 = 0.4 * M_1 + 0.6 * M_2,$$

there M_i – modular current mark, i = 1, 2.

C. *Module 3, 4, current marks:*

$$M_i = min(\kappa_i, \tau_i, \zeta_i), \text{ if } \kappa_i < 4 \text{ or } \tau_i < 4 \text{ or } \zeta_i < 4,$$

else

$$M_i = 0.3 * \kappa_i + 0.4 * \tau_i + 0.3 * \zeta_i$$

there i = 3, 4, κ – written test mark, τ – computer-based test mark, ζ – home written test mark.

D. *Total mark after module 5:*

$$\Theta_5 = 0.3 * M_3 + 0.35 * M_4 + 0.35 * M_5,$$

there M_i – modular current mark, i = 3, 4, 5.

E. *Total yearly mark:*

$$\Theta_y = min(\tau_t, \kappa_t), \text{ if } \tau_t < 4 \text{ or } \kappa_t < 4,$$

else

$$\Theta_y = 0.3 * \Theta_5 + 0.35 * \tau_t + 0.35 * \kappa_t,$$

there Θ_5 – total mark after module 5, τ_t – total computer-based test mark, κ_t – total written test mark.

As it is shown in formulas, each mark has a special threshold value of current marks. Usually written test mark is one of them, the others are depended on a structure of a module. If one of threshold values is unsatisfactory the whole accumulative mark should be unsatisfactory, even though all the others are satisfactory.

X. RESULTS AND STATISTICS

Described way of teaching programming within the bounds of "Computer Science" is taught by authors for almost three years.

Some statistics for second, fifth modules and yearly project of the last two years is following. We don't give statistics for a current year because of it incompleteness.

We didn't use computer-based modular test in 2007-2008 academic year Fig. 1.

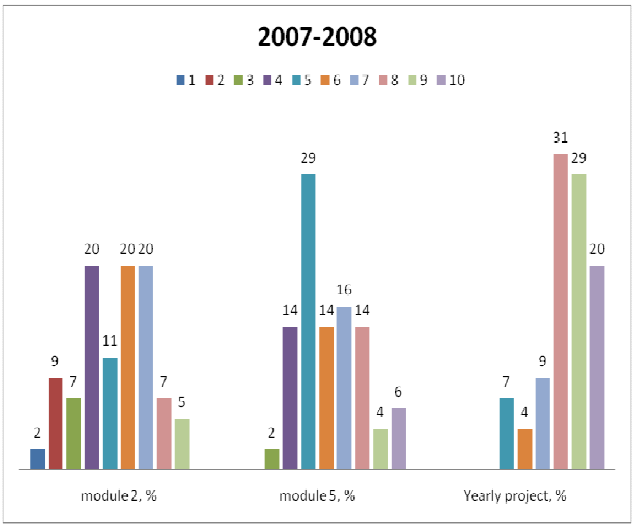

Figure 1. Statistics 2007-2008 acad. year.

We started using computer-based modular test as a test checks in 2008-2009 academic year Fig. 2.

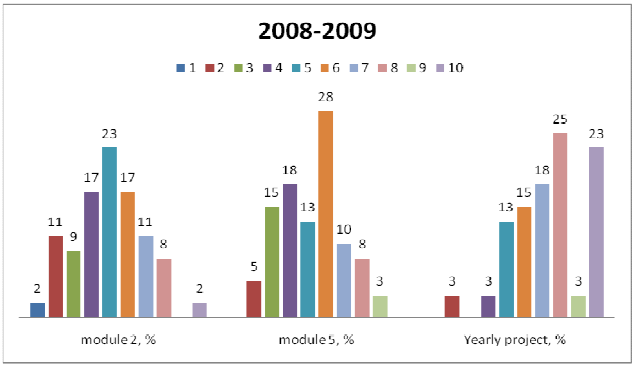

Figure 2. Statistics 2008-2009 acad.year

CONCLUSION

This work describes educational measures in the field of teaching programming as a part of Software engineering education within the bounds "Computer science". All these measures are successfully taken for three years in State

University Higher School of Economics by Software engineering department.

REFERENCES

[1] SWEBOK. Software Engineering – Guide to the Software Engineering Body of Knowledge (SWEBOK). First edition, - Geneva (ISO/IEC 19759: 2005(E)). – 173 pp.

[2] Lethbridge, T.C., Leblanc Jr., R.J., Kelley Sobel, A.E., Hilburn, T.B., Diaz-Herrera, J.L. SE2004: Recommendations for undergraduate software engineering curricula (2006) *IEEE Software*, 23 (6), pp. 21-25

[3] ECMA-334. C# Language Specification. 4th Edition / June 2006, - Geneva (ISO/IEC 23270:2006). – 553 pp.

[4] G. Pearman, J. Goodwill, "Pro .NET 2.0 Extreme Programming", Berkeley: apress, 2006, pp. 3 – 17.