International Journal of Engineering Education Vol. 32, No. 1(B), pp. 521–531, 2016
Printed in Great Britain

0949-149X/91 $3.00+0.00

Using Games to Help Novices Embrace Programming: From Elementary to Higher Education*

SAŠA MLADENOVIĆ and DIVNA KRPAN
Faculty of Science, University of Split, Teslina 12, 21000 Split, Croatia. E-mail: divna.krpan@pmfst.hr, sasa.mladenovic@pmfst.hr

MONIKA MLADENOVIĆ
Elementary School "Blatine-Skrape", Ul. na Križice 2, 21000 Split, Croatia. E-mail: monika.mladenovic@gmail.com

Learning programming is difficult and it presents a great challenge for both students and teachers. The goal is to increase success rate for novice programmers. Students lose their confidence and motivation when they encounter difficulties such as programming environment, language syntax knowledge, problem understanding and debugging. Programming languages are artificial and require high level of abstraction, which is not easy for young students. It is also common knowledge that many adults have more difficulties with learning programming than children. In order to make the difference, we introduced game making course for university undergraduate novice programmers and examined the effect of making games on attitude and motivation for elementary and high school children as well. The objective of using games in this context is to make students learn something serious and difficult like programming while doing something fun like making and playing games. Teachers with a strong understanding of the subject matter they teach are more likely to produce successful students.

Keywords: novice programmers; Scratch; visual programming languages; gamification

1. Introduction

Research on learning and teaching, including education itself, advanced along with the other changes in society throughout the history. Development of technology influenced industry job requirements, which consequently affected education as well [1]. One of the common job requirements today is computer literacy or fluency [2] which includes more than basic skills on how to use computational technology, it implies creative thinking supported with technology. Computational thinking should be considered as one of the fundamental skills for students in 21st century (just like reading, writing, arithmetic, etc.) [3], and programming should be the skill that student must acquire [4]. However, there is a trend of decreasing interest in learning computer science [5], although there is the opposite trend in job requirements [6].

Learning and teaching programming is often considered difficult. Novice programmers quickly lose motivation and give up. University introductory programming courses have high dropout rates [7]. In order to address issues with learning and teaching programming it is important to consider three parts of learning and teaching process: (i) who are the students, (ii) what we teach and (iii) how we teach [8]. Students learn programming from elementary school to university. Novice programmers of different age have different skills, knowledge and abilities, so learning and teaching programming should be adapted accordingly. After defining knowledge and skills suitable for students' age (what to teach), teachers should decide how to teach. For example: which paradigm to choose (object oriented or procedural), teaching strategy or combination of strategies (problem solving, projects, game based learning . . .). "What" and "how" at the university level are often affected by industry job requirements, but for younger students it should be adapted to their characteristics and environment.

The use of technology, problem solving skills and complex communication are some of the most important requirements for future jobs [9]. Because of that, programmers today are expected to be creative. Educational requirements are also affected since students grow up surrounded with technology: computers, smart phones, video cameras, tablets and other devices of digital age. Such students are referred to as "digital natives" [10]. They prefer learning through play rather than doing "something serious". Learning and teaching programming often remains unchanged although students and their learning environments are significantly different. Students today have fast access to vast amount of information; they have multitasking and visual abilities and also want quick and frequent content interactions. Those are characteristics of game-based learning [11-13]. Such students think differently and often solve problems by "trial and error" method, while experimenting and researching [14].

There is not enough research on the influence of game-based learning or *gamification* on learning

* Accepted 11 August 2015.

outcomes [15]. Some research results show that game based programming with storytelling increased motivation for programming for girls in high school [16]. Game-based learning might be the interesting ''buzzword'' in recent researches but it is important to carefully consider how to incorporate it into educational process to avoid pitfalls [17]. We should consider playing and designing games.

1.1 Computer games

Games in general are highly adaptable for different technologies, and there are a large number of game types such as: puzzles, logic games, strategies, social, etc. [18]. From now on in the article, when we refer to games we will consider *computer games* since for the last four decades, computer games are replacing traditional games [19]. There is no unique definition of game, but there are different interpretations of the concept [20]. Juul [21] defines two dimensions of the game: *game* (rules of the game) and *player* (game-player relation). Second dimension defines player requirements such as: rules, outcomes, evaluation of outcomes, consequences in the real world etc. [21]. Playing games increases the hypothesis setting, testing and evaluation cycle [22]. Very simple games become too easy and boring, but too complex games tend to discourage player.

Positive effects of the games include possible development of useful skills and interest in game-based learning and designing games for educational purposes [19]. Games vary on primary function of the game, whether it was developed with the purpose of entertainment, learning or serious game. Educational games should have third dimension: *knowledge* which represents relation between player and educational purpose of the game. Costikyan defined the game simply as: ''*without a goal, it is a toy*'', so according to that, game is actually ''a toy'' with the set of rules [23]. In order to keep the players interested, game should also be fun. There is a formal taxonomy which includes set of rules: sensation, fantasy (make-believe), narrative (good story or drama), challenge (overcoming of obstacles), fellowship (social aspect), discovery, expression (self-discovery) and submission (game as pastime) [24].

1.2 Programming

Learning and teaching programming presents great challenge for students and teachers [25]. It is considered difficult for learning and understanding, especially for novice programmers [26–28]. Many children in elementary schools have negative opinions of computer science [29]. Programming is computer supported problem solving, debugging, development of logic and computational thinking which incorporates development of problem solving strategies (not necessarily in programming area). Learning programming involves more than just acquiring new skills and knowledge: students practice what they learned and apply it on new problems [30]. Although there are many researches about programming, there is no unique or the best method. As we already stated in the introduction, one must consider: who, what and how to teach.

Some researches indicate there are different factors that influence success in learning programming such as: students' attitude, motivation and interest [31]. Without consideration of different students' interests, teaching programming might discourage most of the students from pursuing further learning in the computer science [32].

There is far less enthusiasm for programming today than in 1970s and 1980s. Smart phones now are more powerful than computers back in 1970s, students grow up surrounded with different technologies and their expectations are different, also programming is not so strongly dependent on mathematics and mathematical abilities [33]. Programming is often perceived as activity for smaller part of the population. There are some factors that influenced programming enthusiasm in negative manner [34]: complex programming tools (especially difficult for children), activities and assignments that did not match children's interests and debugging process was complex. Classic programming environments such as BASIC are not visually attractive for children and require syntactically and semantically correct program. Since program cannot run if it is not syntactically correct, very often students focus more on learning syntax than semantics [35].

Abstraction is the foundation of mathematics, science and engineering. According to Piaget's theory of cognitive development, there are four stages of human cognitive development: sensorimotor (0–2 years), preoperational (2–7 y.), concrete (7–11 y.) and formal operational stage (>12 y.), [36]. Research studies show that only 34 % of the adolescents reach last stage, and results are not much better for adults either [37]. The results imply that schools should adapt curriculum to students' cognitive abilities, and encourage development of abstract abilities. Children can learn better about things that are tangible and accessible. Combined with the experience, they gain ability to understand abstract concepts, reason and generalize. Concrete experiences are the most effective in the relevant context (real world situations) [14, 38]. Sometimes students simply recite terms which they do not actually understand and teachers overestimate students' ability to understand abstraction. Programming languages are artificial and require

high level of abstraction as well as the programming activity itself [39].

1.2.1 Introductory programming languages

Novice programmers need to learn concepts and techniques, not a specific programming language. However, one language should be chosen in order to have the concrete experience. Some languages are better suited than others in different aspects, and for example, some authors recommend language with simple syntax, quick feedback and support for structured programming (referred to visually distinguishing block statements) [40]. Learning basic concepts is the foundation for building more advanced skills [41]. Very often, language choice does not depend on pedagogical choices but instead on the industry. For example, although programming language LOGO might be good for novices at the university level, most universities decide against it.

Papert developed LOGO guided by the principle of thinking from concrete to abstract [42]. Students write programming instructions and immediately receive the result visually. Papert argued that programming language should have *low floor* (easy start), *high ceiling* (opportunity to create more complex projects) and *wide walls* (engagement of students with different interests) [34, 43]. The success of LOGO inspired development of the whole family of LOGO-like languages that produced large variety of novice environments oriented on the turtle graphics and geometry, for example: Scratch, Alice, Greenfoot, GameMaker, Kodu, NXTG (Lego-Mindstorms). Such languages are called: *mini-languages* [44]. They are small, simple and intuitive, which is especially convenient for novice programmers. In most of mini-languages students control main character or sprite. The character can be virtual (such as turtle in LOGO) or real (Lego robot).

Visualizations have been used for long time in computer science education as the important approach for understanding abstract constructs [41]. Visualization might help students to learn some concepts better. Mostly, they are concerned about algorithm animation and not basic program structure and execution. Visual programming languages enable students to experience programming from concrete to abstract. Difficult abstract concepts are represented through concrete events and concept visualization. Such programming languages provide aforementioned necessary interaction for digital natives, enable context programming, and they are visually attractive. Some of the visual programming languages have visual representation of programming instructions: Scratch [45] or its dialects (Snap! or BYOB [46]). Programming instructions are in the form of puzzles, removing syntax issues. Novices find learning syntax confusing and overwhelming which might discourage them [47]. General-purpose languages include constructs which are sometimes difficult (because of abstraction) and although similar to natural English language, they might have different meaning.

Visual programming languages history goes back to 1960s [48]. Since visual programming languages field matured, some form of classification emerged, and we will mention only two most interesting categories: (i) purely visual programming languages, (ii) hybrid programming languages. Other categories are not mutually exclusive. In pure visual programming languages, the programmer works with icons or other graphical representations in order to create a program which is also executed in the same visual environment. There is no translation in any form of textual language. Other important category is approach of combining visual and textual elements. It is noted that after some time programmers overgrow visual interface and become more comfortable in writing complex programs in textual form. Novice environments tend to remove or simplify syntax and introduce visible immediate results or motivating contexts (such as making robots or games). By doing all that, it is possible to attract wider audience into programming and to allow students to focus on logic and structures [47]. Designers tend to make such environments closer to general-purpose languages, in order to make the transition easier.

1.3 Game-based programming and learning

Programming languages mentioned above are potential environments for making and playing games. In order to connect learning programming and playing games, there is a natural idea of learning programming while creating games. Students find learning more interesting, and by shifting the focus from learning syntax to semantic structure, students are not even aware they learn problem solving, debugging, and making scenarios; instead they think they are simply making games. Encouraging students to learn one thing while doing something else Pauch called *head fake* [49]. Researches in this area are often conducted in the summer camps or clubs because there is no wide accepted school study programmes including game-based learning [50]. There are many researches which showed how computer games programming increases motivation, and that students learn programming concepts through different environments, for example: using Alice [51–53], Scratch [50, 54, 55] and Kodu [56]. Using these environments increases students' computer fluency [53] and by using Scratch as first programming language students are learning basic programming concepts [57].

Longitudinal research was conducted in Finland with students of age 12–16 years [58]. During three years researchers monitored students who attended one-week summer programming courses. Most of the students pursued programming further and their interest in computer science increased. Girl students were equally interested as boys.

Research with undergraduate art students was conducted during course of 3D game programming [59]. The final result of the course was dozens of games—team projects (teams consisted of 3–6 students). They used tools such as Macromedia Director, Flash, Java, Virtools and Quest3D. Games were often better than those developed by computer science students.

The study where teachers applied *game-first approach* demonstrated that students acquired basic programming concepts better, there was decreased dropout rate in computer science and increased overall satisfaction with learning [60].

1.3.1 Mediated transfer

Research studies show the increase of students' problem solving skills while playing, as well as the transfer of problem solving skills to different problems in the game, but it was difficult to transfer them out of the game [61]. Curtis & Lawson found moderate transfer of problem solving skills [62], but it seems that transfer level was low [63].

There is a lack of *mediated transfer* or approach where skills and knowledge acquired during gaming should be transferred in the real world (like learning programming concepts, which was actually the objective of the research) [64]. There are two techniques of mediated transfer: *bridging* (teacher helps students to create the bridge from the context in which programming concept was learned into other contexts) and *hugging* (teacher creates learning situation in which the transfer is expected). Teacher should help with the transfer from the gaming into learning outcomes. There is an example of study conducted with the high school students and game *Crystal Island* where students learned microbiology concepts. They made notes of hypotheses and discoveries acquired during game play [65].

Meta-analysis conducted on 17 studies emphasizes the necessity for teacher's support during concept transfer through different contexts [66]. Game-based learning will not replace the teachers. Li & Watson present three approaches to game-based learning of programming [67]:

- Authoring-based approach
- Play-based approach
- Visualization-based approach

In the first approach student's learning activity is the game development (based on constructivist theory). Students get game assignment and maybe partial implementation. The developed game is also student's reward. Partial implementation is usually provided for students when programming environments are too complicated and it takes too long for students to develop basic game structure before they experience actual progress. Block based languages such as Scratch, allow students to focus on the problem solving instead on syntax errors. In the play-based approach student develop programming strategies to complete specific task or win. The coding is the smaller part of the solution. Visualization in this context only demonstrates code execution in visual context and students are not able to create games or play.

Sometimes teachers expect too much from games (that students will actually get all knowledge and skills as expected). Regardless of the optimism and high expectations from game-based learning some authors noted there is a scarce of quality evidence to support them [19]. Connolly conducted extensive literature review on vast number of research papers. Majority of the papers contained speculations on the game potential, description of theories and game development but there were no empirical evidence considering impact on learning outcomes, so those papers were discarded from further investigation. Subject area of the games in rest of the papers was mostly concerned about entertainment. There was some evidence that games designed with entertainment purpose could be used in education. Learning outcomes could be classified into two categories: *softer* (emotions, motivation and attitude) and *harder* (knowledge and skills acquisition). In the next section we will explain the experience of game-based learning with the elementary school students, which was mostly oriented on softer category of the outcomes. As stated before, game-based learning is typically situated outside the classroom. Some of the reasons are predefined curriculums and lack of textbooks. High schools in Croatia have diverse curriculums, and when programming is concerned, it ranges from nothing to ''hardcore programming'' in C. there is the intention of introducing game-based programming to high school students as well, encouraged by the research described in the next section. We argue that carefully implemented game-based learning of programming is appropriate for novices of different ages (from elementary school to university) regardless of the environment. The research described in this paper was conducted with elementary and university undergraduate students. Since we did not have an opportunity to conduct the research with high school students using visual programming environment, we examined the literature to provide a closer insight.

1.4 Learning computer science concepts in high school

Meerbaum-Salant et al. developed a set of Scratch-based learning materials for middle school students (similar age of lower high school grade students in Croatia) and conducted research during school hours. Since the course was not required by the curriculum, authors considered that some students and teachers were probably not committed enough [57]. Middle school students should be provided with the learning materials but the routine textbook is not adequate for Scratch since it is in the contrast with the constructivist philosophy. The textbook in this case is more like a guideline. Very often programming courses and textbooks follow language specific features. The authors follow principles:

- Teaching specific concepts instead language feature,
- Programming construct are introduced as needed,
- Project-based presentation
- Visual appearance is optional

Research sample consisted of 9th grade students (14–15 year), and authors used mixed analysis (qualitative and quantitative), and also mixed taxonomies (SOLO and Bloom). Students were introduced to the concepts of loops and concurrent programs very early. Type I concurrency occurs when two sprites execute scripts in parallel, while Type II concurrency occurs when one sprite executes more than one script in parallel. The results showed that most students were able to understand computer science concepts although they had difficulties with the concepts of initialization, variables and concurrency.

Besides learning of computer concepts, Scratch also influenced habits of programming [68]. Such aspect was not expected during previous experiment but students demonstrated two habits: bottom-up programming habit and extremely fine-grained programming which takes top-down approach to the extreme (decomposing into very small logical units). That seemed to be the side effect and the problem with the habits is that they tend to be persistent. Students used instructions in the way that seemed natural to them, and the introduction of the scenario-based programming should be considered and prepared more carefully [69]. Teacher should be aware of the students' tendencies as well as different design principles.

2. Research results

Appropriate programming language for young students should be as simple as possible and preferably visual since visual programming environment encourages learning by exploration [68]. While teaching programming to the elementary school students in the past, we experienced difficulties with text based programming languages such as BASIC (6th grade students). On the other hand, younger students had no problem solving more complex assignments in LOGO. However, LOGO programming language and assignments that students are supposed to solve are also very much connected with mathematics. For that reason, students in this small study were presented with short Scratch introduction, as part of the normal school hours. The interest was more in affective results because we were not able to differ much from standard curriculum.

2.1 Elementary school students with Scratch experience

The term *informatics* is often used in the Croatian educational system instead of *Computer science*, and we will use it further in this article. The Croatian National Educational Standard (CNES) introduced standard for learning and teaching informatics as elective course from 5th to 8th grade in elementary schools with the standard curriculum. In the school year of 2005/2006, CNES elements were experimentally introduced in 5% of elementary schools in the Republic of Croatia, and next school year (2006/2007), all elementary schools had started implementing the CNES. Programming is part of the curriculum (average of 13 *school hours* per school year). School hour in Croatian educational system (elementary school, high school and university) equals 45 minutes.

Students in Croatian elementary schools mostly learn LOGO as the first programming language, normally in lower grades from 5th to 6th and have an opportunity to switch to BASIC or continue learning LOGO (recently there is also emerging trend of introducing Python). As we already stated before, this approach of learning programming is usually not the part of the curriculum, and therefore we were obligated to adapt assignments and tests to cover concepts students were expected to learn by the CNES. Study was conducted on the sample of 24 elementary school students in the 7th grade (12–13 years old). Students were given the assignments to solve in Scratch instead of LOGO. There were 5 test assignments, and the results are presented on the Fig. 1.

Two of the students were on different study program and were excluded from these test results. Assignments covered instructions for moving forward, rotation, loops and nested loops. Assignment 1 was simple recognition of instructions (remembering category of the Bloom's revised taxonomy [70],

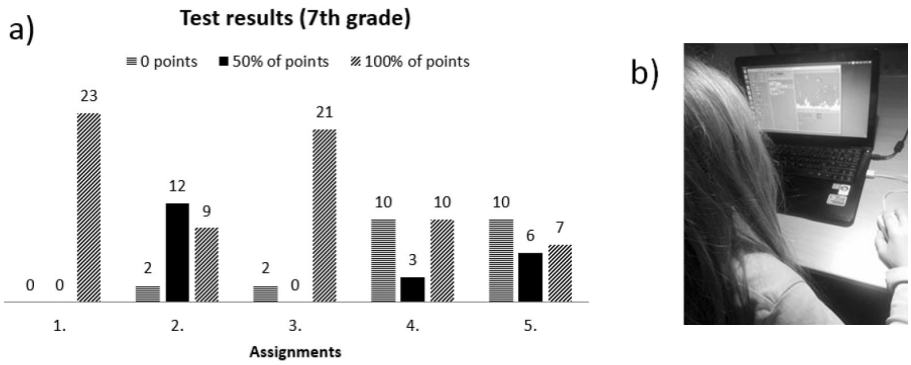

Fig. 1. (a) Test results for 7th grade, (b) Student programming in Scratch.

and since all Scratch instructions were in Croatian, all students solved it. Students had more difficulties with the assignment 4 (nested loops) and 5 (write small program). 29% of the students had no previous programming experience, and 80% of students stated they liked Scratch more. Some of the students commented they liked block instructions more since they did not need to remember instructions. They graded their preference for each programming language with grades from 1–5 and average for Scratch was 3.8, and LOGO 2.8.

It is important to note that informatics in Croatian elementary and high schools is elective course and if students are not interested enough or consider it too difficult, they drop out and there are many students that do not enroll at all. High school students that enroll, mostly attend classes twice a week for one school hour. Some schools teach informatics only during one school year (first or second year), and students learn mostly office applications (word editors, spreadsheet tables etc.) with very little programming if any at all. As the result, some students that attended informatics classes might have at least a three-year gap before university. Schools with ''mathematics'' in their title teach four years of informatics with more programming lessons and students from those schools are in fact more successful in programming courses at the universities.

Next part of this paper refers to the research we conducted with the university undergraduate first year students during their first semester. Motivation for the research emerged from years of experience in teaching undergraduate students during introductory programming courses at the Faculty of Science, University of Split (FOS).

2.2 Scratch goes to higher education

First year students at the FOS have same problems as most novice programmers, and most of them enroll introductory programming courses as absolute beginners. They are different based on their study majors: mathematics (M), physics (P), technical science (T), informatics (I), engineering physics (eP), and also combinations (MI, PI, IT). All students learn Python as their first programming language. Python is well accepted with our students ever since its introduction at the FOS [71]. However, approach of the game based learning was considered in order to increase motivation for novice undergraduate programmers for one group of students (I—study major in Informatics), but there was again a problem of programming language choice and students' lack of knowledge and skills. We decided to use Scratch as visual and syntax-free language, and later its dialect Byob (developed at Berkley University). Byob was a direct modification of the Scratch source code and after it was completely rewritten, it is now known as Snap! project. Research on the effect of game based learning using Scratch and Byob with undergraduate students at the FOS is actually ongoing research which started at 2009 and during five years we monitored students' progress. Using programming environment mostly considered for young students (elementary and high school students) with the adults (undergraduate students) might be considered unusual, but we also inherited problems from previous stages of students' education and the question was, why not? We expected to engage students by making programming experience more fun and accessible, and to make them more confident. The problem is somewhat more challenging since students with major in informatics study to be teachers. There are actually two introductory programming courses at the FOS: *Programming I* (P1) in the first semester and *Programming II* (P2) in the second semester. In 2009 we introduced new course: *Advanced P2 lab* for group I with major in the computer science. Students learned Python as additional language, but although they found it interesting, we noted that learning new syntax did not help. Next year, curriculum was reorganized and new course was introduced: *IT Project 1* (IP1) in the first semester along with the P1. During the course IP 1 students learn programming while making

games in visual programming environment. Each lab assignment consists of making small game. After first year programming in Scratch (2009/10), students found environment somewhat limiting and we switched to Byob because of its additional feature: Build Your Own Block.

2.2.1 Data collection

We conducted an *ex post facto* study, and type of the selected sample was accidental [72]. Accidental sampling is based upon convenience. During three years (2010–2013) there were a total of 727 students enrolled in the course P1 and 784 students enrolled in the course P2. We focused on the students that enrolled courses for the first time since they had no previous knowledge of the course content. After data refinement process, total of 510 students remained. All students that enroll P1 also enroll P2. Introductory programming courses, as already mentioned, were enrolled by groups: IT, I, M, MI, eP and PI. Comparison of the results for groups at the FOS confirms that students with major in mathematics have higher percentage of success or pass rate (Fig. 2).

We focused our interest on groups IT and I since students with mathematics as their major had higher success rates and groups with the major in physics consisted of small number of students. Group I, it was selected as an experimental group (introduction of course IP1 as research variable) and IT as a control group, selected retrospectively [72]. Collected data included: high school grades average (HSA), P1 grade (final grade), P2 grade (final, midterm and final practical exam) and attendance.

Further data refinement reduced sample population to 452 students since 57 students had missing data (HSA). Finally, sample population used for the experimental group I and control group IT consisted of the 202 students through 3 years (6 semesters).

Students in course P1 learn structured programming, and we considered that students should be more prepared for course P2 and introduction to the object-oriented approach since Byob is object-oriented and event-driven environment. While conducting informal interviews with the random students and observing their performance during first year with Scratch, it was established that some of them found the environment quite limiting and expressed frustration. Those students were perspective with highest final grades on all three observed courses. Next year, Byob was introduced and we received no more such complaints, since students were actually able to make their own blocks, but after designing larger project the script area became crammed with blocks. Students expressed the need for text-based interface.

2.2.2 Data analysis

We set the null-hypotheses for the groups IT and I:

H1: There is no statistically significant difference in the high school final grade average between the control and experimental group.

H2: There is no statistically significant difference in the P1 final grade between the control and experimental group.

H3: There is no statistically significant difference in the P2 final grade between the control and experimental group.

First, we tested normal distribution on all variables with Shapiro-Wilk W test. Test was conducted for all three years combined and separately.

Since all the results for HS average variable were normally distributed ($p > 0.05$), it was appropriate to perform the T-test for groups IT and I. Grade distribution for P1 and P2 fits power law distribution in accordance with previous research [73]. Experimental group achieved statistically significant better results ($p = 0.021$), hence we reject null-hypothesis H1 and conclude there is statistically significant difference between IT and I in the

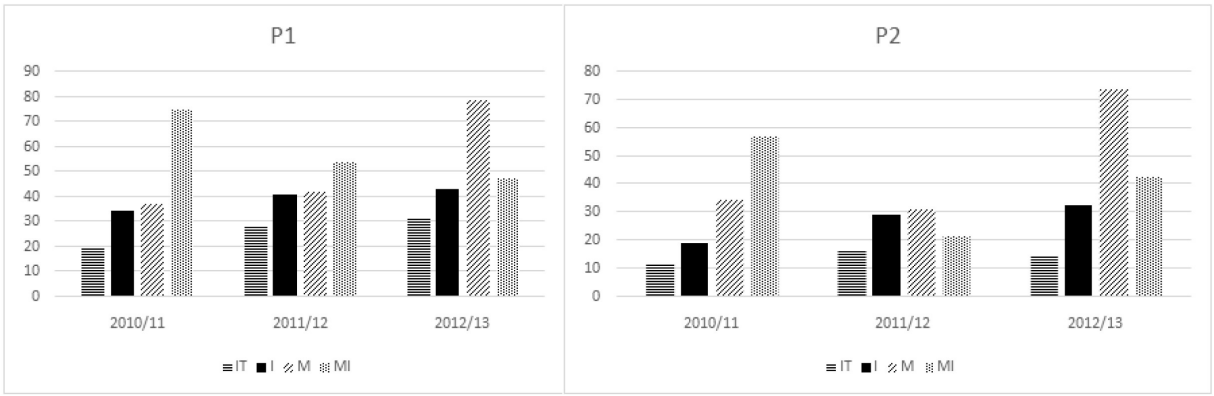

Fig. 2. Students that passed courses P1 and P2.

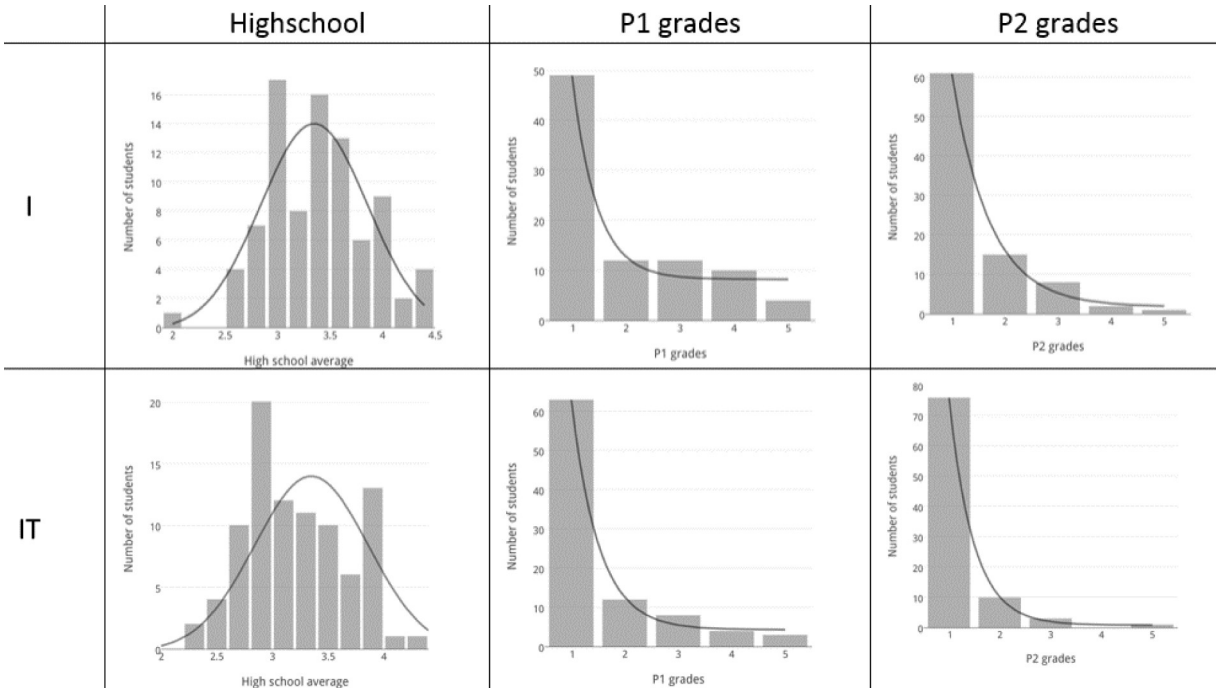

Fig. 3. Grade distribution for control and experimental group.

HS average. Data for the P1 and P2 grades were not normally distributed, and consequently not appropriate for T-test, we performed nonparametric Mann-Whitney U test. Both, H2 and H3 were rejected ($p < 0.05$).

Further, we separated groups for each school year and tested H1, H2 and H3 for IT and I in each year. HS average was normally distributed. T-test result for first year ($p = 0.434$), second year ($p = 0.07$) and third year ($p = 0.12$) showed there was no statistically significant difference between groups I and IT for each year.

Data for P1 and P2 grades was not normally distributed and we conducted Mann-Whitney W test with the results:

- First year: there is statistically significant difference between IT1 and I1 on both P1 ($p = 0.045$) and P2 ($p = 0.0002$) grades, although small one for P1.
- Second year: there is no statistically significant difference between IT2 and I2 for P1 ($p = 0.292$) and there is a difference for P2 ($p = 0.012$).
- Third year: there is no statistically significant difference between IT3 and I3 for P1 ($p = 0.073$) and there is a difference for P2 ($p = 0.002$).

Variables HS average and P1 grades were considered as initial testing. First midterm exam in P2 course consisted of C# console applications, and final exam was based on graphical user interface (GUI) development. Since GUI is event driven as Scratch and Byob, results were more thoroughly examined for year 2012/13. After data refinement (considering attendance records as well), the same hypothesis testing was conducted on small sample (IT = 32, I = 34 students). T-test showed there was no statistically significant difference in initial conditions (HSA) between groups, and by Mann-Whitney U test, there was no statistically significant difference in midterm console exam. The difference was established in the final test results based on GUI applications ($p = 0.049$).

In the next section, we will interpret the results in more detail and conclude on the importance of hypotheses testing.

3. Discussion

Most of the researches about game-based learning or gamification in education have something in common: increase in interest and motivation for different age of students. Since we were able to design the whole course in Scratch, the study with the undergraduate students was more extensive. The study investigated the influence of the introduction of game-based learning on the final grade of the programming course in C#. During previous analysis it was established that experimental group performed better in the course P2 than control group when we observed each year separately, but those differences were weak and students with mathematics as their major were still better Students in the course P1 learn procedural programming, while Scratch and its dialect are event driven and

object-oriented. Since course P2 is an introduction to object-oriented programming, we consider IP1 and P2 more related to each other, which explains reasons for slightly better students' performance. Interviews with the students during first year led us to switch from Scratch 1.4 to Byob where students were able to develop their own blocks. Further it was established that better students were creating more complex games and overgrew the visual environment with higher ambitions and interest. Weaker students were not left behind and they gained more confidence and still considered the course fun.

Undergraduate students study to become teachers of informatics, and if they learn poorly, they will be poor teachers to children that might eventually become undergraduate students at the faculty. Someone must break out of the loop. Positive experiences encourage us to think further. With the introduction of Byob the demands on students were slightly higher. However, we noticed that although students for example, used events with Scratch and Byob, they did not transfer the experience on GUI applications programming and although each sprite is actually an object with attributes and actions capable of cloning (making an instance), they did not connect those concepts with classes and objects in C#. Consequently, we focus future research plans on mediated transfer teaching techniques from visual programming environments and game-based learning of programming in Scratch 2.0 to the C# programming language (Fig. 4).

We are developing the framework where students will start learning programming while making games in Scratch and transfer programs and learned concepts in C# programming environment. Student would be provided with the previously designed game environment (some of the methods for Sprites and its behavior used in Scratch 2.0), and they would be able make similar games in C#.

4. Conclusion

Research on the related work demonstrates that problem with the teaching basic programming skills to undergraduate students is universal. Approach with the introduction of the visual programming environment in order to help novices is often observed while dealing with children as novice programmers, but the use Scratch or its dialects in university courses is not uncommon. While experimenting with different versions of visual programming languages during three semesters, we conclude there is a positive influence on the object-oriented introductory programming course P2. Game-based learning for novice undergraduate students must be expandable since students quickly outgrow initial set of instructions (blocks), and the best of them seem unable to further express themselves. Better students tend to make more complex scripts that get crammed in the script area and according to their comments, they find it difficult to edit. They miss textual environment with the ability to search through code while considering visual environment as ''not serious programming''. Students should be able to advance further and simply continue their work in some ''serious'' object-oriented programming language such as C# which is used in the next course P2. There is a missing link between Scratch and C# since weaker students do not transfer knowledge or connect concepts they learned in Scratch with the concepts they learn in C# (for example click event for sprite in Scratch is the same concept as click event for button in C#). Students are often intimidated by complex syntax and are not able to make games as they did in Scratch. The idea (Fig. 4.) is to create the environment that is able to transfer programs written in Scratch 2.0 (in visual form) to the C# programming environment (in textual form), which would enable students to continue programming in C# using the predefined set of classes and methods that follow the text written on Scratch blocks as much as possible (according to the C# syntax). Such new and improved visual environment closely connected to C# would provide them an important head start and easier transfer to the overwhelming set of classes and methods. Students would be able to continue programming games they started in Scratch during next course, but they would do it in C# instead.

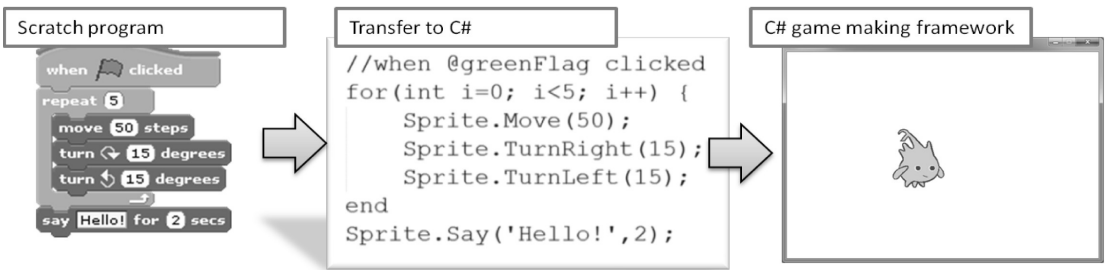

Fig. 4. Mediated transfer from visual programming to C#.

References

1. C. M. Reigeluth, Instructional Theory and Technology for the New Paradigm of Education, *RED, Revista de Educación a distancia*, **32**, 2012, p. 30.
2. K. Williams, Literacy and Computer Literacy: Analyzing the NRC's "Being Fluent with Information Technology", *Journal of Literacy and Technology*, **3**(1), 2003, pp. 1–20.
3. J. M. Wing, Computational Thinking, *Communications of the ACM*, **49**(3), 2006, pp. 33–35.
4. S. Uludag, M. Karakus and S. W. Turner, Implementing IT0/CS0 with Scratch, App Inventor for Android, and Lego Mindstorms, *Proceedings of the 2011 conference on Information technology education*, 2011, pp. 183–190.
5. P. J. Denning and A. McGettrick, Recentering Computer Science, *Communications of the ACM*, **48**(11), 2005, pp. 15–19.
6. C. Litecky, B. Prabhakar and K. Arnett, The IT/IS Job Market: A Longitudinal Perspective, *Proceedings of the 2006 ACM SIGMIS CPR Conference on Computer Personnel Research: Forty four Years of Computer Personnel Research: Achievements, Challenges & the Future*, 2006, pp. 50–52.
7. S. Garner, P. Haden and A. Robins, My Program is Correct But it Doesn't Run: A Preliminary Investigation of Novice Programmers' Problems, *Proceedings of the 7th Australasian conference on Computing education*, 2005, pp. 173–180.
8. M. Pedroni, *Teaching Introductory Programming with the Inverted Curriculum Approach*, Diploma thesis, Department of Computer Science, ETH, ETH, Eidgenössische Technische Hochschule, Zurich, 2003.
9. F. Levy and R. J. Murnane, *The New Division of Labor: How Computers Are Creating the Next Job Market*, Princeton University Press, 2012
10. M. Prensky, Digital Natives, Digital Immigrants, *On the Horizon*, 2001, pp. 1–6.
11. J. S. Brown, Growing Up Digital: How the Web Changes Work, Education, and the Ways People Learn, *USDLA journal*, **16**(2), 2002, p. n2.
12. M. Prensky, Digital Game-based Learning, *Computers in Entertainmen*, **1**(1), 2003, p. 21.
13. D. Oblinger and J. Oblinger, Is It Age or IT: First Steps Toward Understanding the Net Generation, *Educating the Net Generation*, **2**(1–2), 2005, p. 20.
14. S. Turkle and S. Papert, Epistemological Pluralism and the Revaluation of the Concrete, *Constructionism*, 1991, pp. 161–192.
15. P. Wouters, E. D. van der Spek and H. Van Oostendorp, Current Practices in Serious Game Research: A Review from a Learning Outcomes Perspective, *Games-based learning advancements for multisensory human computer interfaces: techniques and effective practices*, 2009, pp. 232–255.
16. C. Kelleher, R. Pausch and S. Kiesler, Storytelling Alice Motivates Middle School Girls to Learn Computer Programming, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2007, pp. 1455–1464.
17. S. Papert, Does Easy Do It? Children, Games, and Learning, *Game Developer*, **5**(6), 1998, p. 88.
18. J. Kirriemuir and A. McFarlane, Literature Review in Games and Learning, *Futurelab, A Graduate School of Education, University of Bristol*, 2004,
19. T. M. Connolly, E. A. Boyle, E. MacArthur, T. Hainey and J. M. Boyle, A systematic literature review of empirical evidence on computer games and serious games, *Computers & Education*, **59**(2), 2012, pp. 661–686.
20. J. Smed and H. Hakonen, *Towards a Definition of a Computer Game*, Turku Centre for Computer Science Turku, Finland, 2003.
21. J. Juul, The Game, the Player, the World: Looking for a Heart of Gameness, *DIGRA Conf.*, 2003.
22. R. Van Eck, Digital Game-Based Learning: It's Not Just the Digital Natives Who Are Restless, *EDUCAUSE Review*, **41**(2), 2006, p. 16.
23. G. Costikyan, I Have No Words & I Must Design: Toward a Critical Vocabulary for Games, *Computer Games and Digital Cultures Conference Proceedings*, Tampere, 2002, pp. 9–33.
24. R. Hunicke, M. LeBlanc and R. Zubek, MDA: A Formal Approach to Game Design and Game Research, *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004, pp. 04–04.
25. I. Milne and G. Rowe, Difficulties in Learning and Teaching Programming—Views of Students and Tutors, *Education and Information technologies*, **7**(1), 2002, pp. 55–66.
26. A. Gomes and A. J. Mendes, Learning to program—Difficulties and solutions, *International Conference on Engineering Education—ICEE*, 2007.
27. L. E. Winslow, Programming Pedagogy—A Psychological Overview, *ACM SIGCSE Bulletin,* **28**(3), 1996, pp. 17–22.
28. A. Robins, J. Rountree and N. Rountree, Learning and Teaching Programming: A Review and Discussion, *Computer Science Education*, **13**(2), 2003, pp. 137–172.
29. B. Violino, Time to Reboot, *Communications of the ACM*, **52**(4), 2009, pp. 19–19.
30. M. Hassinen and H. Mäyrä, Learning Programming by Programming: a Case Study, *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling*, 2006, pp. 117–119.
31. N. F. A. Zainal, S. Shahrani, N. F. M. Yatim, R. A. Rahman, M. Rahmat and R. Latih, Students' Perception and Motivation Towards Programming, *Procedia-Social and Behavioral Sciences*, **59**, 2012, pp. 277–286.
32. A. Forte and M. Guzdial, Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses, *IEEE Transactions on Education*, **48**(2), 2005, pp. 248–253.
33. J. M. Wing, Computational Thinking and Thinking about Computing, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences,* **366**(1881), 2008, pp. 3717–3725.
34. M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver and B. Silverman, Scratch: Programming for All, *Communications of the ACM,* **52**(11), 2009, pp. 60–67.
35. C. M. Lewis, How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch, *Proceedings of the 41st ACM technical symposium on Computer science education*, 2010, pp. 346–350.
36. J. Piaget, The Origins of Intelligence in Children, *Journal of Consulting Psychology*, **17**(6), 1953, p. 467.
37. E. Fusco, Matching Curriculum to Students Cognitive Levels, *Educational Leadership*, **39**(1), 1981, pp. 47–47.
38. W. Dann and S. Cooper, Education Alice 3: Concrete to Abstract, *Communications of the ACM*, **52**(8), 2009, pp. 27–29.
39. R. Moser, A fantasy adventure game as a learning environment: Why learning to program is so difficult and what can be done about it, *ACM SIGCSE Bulletin*, 1997, pp. 114–116.
40. L. Grandell, M. Peltomäki, R. Back and T. Salakoski, Why Complicate Things? Introducing Programming in High School Using Python, *Proceedings of the 8th Australian Conference on Computing Education*, 2006, pp. 71–80.
41. K. Ala-Mutka, Problems in Learning and Teaching Programming, *Codewitz Needs Analysis*, 2012.
42. S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc., 1980.
43. M. Guzdial, Programming Environments for Novices, *Computer Science Education Research,* 2004, pp. 127–154.
44. P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko and P. Miller, Mini-languages: A Way to Learn Programming Principles, *Education and Information technologies,* **2**(1), 1997, pp. 65–83.
45. About Scratch, http://scratch.mit.edu/about/, Accessed 20.01.2015.
46. Snap! (Build Your Own Blocks), http://snap.berkeley.edu/, Accessed 20.01.2015.
47. C. Kelleher and R. Pausch, "Lowering the Barriers to Programming: A Survey of Programming Environments and Languages for Novice Programmers," DTIC Document 2003.
48. M. Boshernitsan and M. S. Downes, *Visual Programming Languages: A Survey*, Citeseer, 2004.
49. R. Pausch and J. Zaslow, The Last Lecture: Really Achieving

Your Childhood Dreams, *Given at Carnegie Mellon University*, 2007,
50. J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick and N. Rusk, Programming by Choice: Urban Youth Learning Programming with Scratch, *ACM SIGCSE Bulletin*, 2008, pp. 367–371.
51. L. Werner, S. Campe and J. Denner, Children Learning Computer Science Concepts via Alice Game-programming, *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 2012, pp. 427–432.
52. L. L. Werner, S. Campe, and J. Denner, Middle School Girls + Games Programming = Information Technology Fluency, *Proceedings of the 6th conference on Information technology education*, 2005, pp. 301–305.
53. D. Werner, J. Denner, M. Bliesner and P. Rex, Can Middle-Schoolers use Storytelling Alice to Make Games, *Results of a Pilot Study, Orlando*, 2009.
54. D. J. Malan and H. H. Leitner, Scratch for Budding Computer Scientists, *ACM SIGCSE Bulletin*, **39**(1), 2007, pp. 223–227.
55. A. Wilson, T. Hainey and T. Connolly, Evaluation of Computer Games Developed by Primary School Children to Gauge Understanding of Programming Concepts, *6th European Conference on Games-based Learning (ECGBL)*, 2012, pp. 4–5.
56. A. Fowler and B. Cusack, Kodu Game Lab: Improving the Motivation for Learning Programming Concepts, *Proceedings of the 6th International Conference on Foundations of Digital Games*, 2011, pp. 238–240.
57. O. Meerbaum-Salant, M. Armoni and M. Ben-Ari, Learning Computer Science Concepts with Scratch, *Computer Science Education,* **23**(3), 2013, pp. 239–264.
58. A.-J. Lakanen, V. Isomöttönen and V. Lappalainen, Life Two Years After a Game Programming Course: Longitudinal Viewpoints on K-12 Outreach, *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 2012, pp. 481–486.
59. M.-H. Tsai, C.-H. Huang and J.-Y. Zeng, Game Programming Courses for Non Programmers, *Proceedings of the 2006 international Conference on Game Research and Development*, 2006, pp. 219–223.
60. S. Leutenegger and J. Edgington, A Games First Approach to Teaching Introductory Programming, *ACM SIGCSE Bulletin*, **39**(1), 2007, pp. 115–118.
61. S. Egenfeldt-Nielsen, Overview of research on the educational use of video games, *Digital kompetanse*, **1**(3), 2006, pp. 184–213.
62. D. D. Curtis and M. J. Lawson, Computer Adventure Games as Problem-Solving Environments, 2002,
63. S. Egenfeldt-Nielsen, Third generation educational use of computer games, *Journal of Educational Multimedia and Hypermedia*, **16**(3), 2007, pp. 263–281.
64. W. Dann, D. Cosgrove, D. Slater, D. Culyba and S. Cooper, Mediated Transfer: Alice 3 to Java, *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 2012, pp. 141–146.
65. K. Ash, Digital Gaming Goes Academic, *Education Week,* **30**(25), 2011, pp. 24–28.
66. F. Ke, A Qualitative Meta-Analysis of Computer Games as Learning Tools, *Handbook of research on effective electronic gaming in education*, **1**(2009), pp. 1–32.
67. F. W. Li and C. Watson, Game-based Concept Visualization for Learning Programming, *Proceedings of the third international ACM workshop on Multimedia technologies for distance learning*, 2011, pp. 37–42.
68. O. Meerbaum-Salant, M. Armoni and M. Ben-Ari, Habits of Programming in Scratch, *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 168–172.
69. M. Gordon, A. Marron and O. Meerbaum-Salant, Spaghetti for the Main Course: Observations on the Naturalness of Scenario-Based Programming, *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, 2012, pp. 198–203.
70. L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, P. R. Pintrich, J. Raths and M. C. Wittrock, A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Abridged Edition, *White Plains, NY: Longman*, 2001.
71. D. Krpan and I. Bilobrk, Introductory Programming Languages in Higher Education, *MIPRO, 2011 Proceedings of the 34th International Convention*, 2011, pp. 1331–1336.
72. R. Kumar, *Research Methodology—A Step-by-Step Guide for Beginners*, Sage Publications, London, Thousand Oaks, 2012.
73. H. Aguinis, The best and the rest: Revisiting the Norm of Normality of Individual Performance, *Personnel Psychology*, **65**(1), 2012, pp. 79–119.

Saša Mladenović is vice dean for education at the Faculty of Science, University of Split (Croatia). He teaches number of doctoral, graduate and undergraduate courses at the Department of Informatics. He received his Ph.D. in computer science at the Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture in Split (FESB). From 1999 to 2006 he was acting as Technical manager of the Toll collection system department of Ecsat, Croatia—the company responsible for software development at the Transportation department of CS group Designer, integrator and operator of mission critical systems, France. His research interests include: problems in teaching programming, interoperability, intelligent technologies like ontology and multi-agent systems, especially engineering applications of intelligent technologies. He is IEEE member since 1996.

Divna Krpan is a lecturer at the Department of Informatics at the Faculty of Science, University of Split. She holds a degree in Mathematics and Informatics. The title of her graduate thesis was ''Knowledge evaluation in e-learning systems''. She teaches number of undergraduate courses in programming. Her main interests include research on improving teaching and learning of computer programming for novice programmers at the introductory undergraduate programming courses.

Monika Mladenović is Computer Science teacher in elementary schools Blatine-Skrape and Spinut, external Associate Teaching assistant at the Faculty of Science, University of Split (Croatia) and external Associate software developer at the Clinical hospital Split for developing and maintenance of hospital business information system. At the Faculty of Science she teaches basics of computer science and she acts as methodical mentor for graduate students. She is also PhD candidate at the Faculty of Science, University of Split (Croatia) with the research area of using computer games for teaching programming. She is ACM member since 2011.