Informatics in Education, 2021, Vol. 20, No. 3, 489–511

© 2021 *Vilnius University, ETH Zürich*
DOI: 10.15388/infedu.2021.20

Mediation of Knowledge Transfer in the Transition from Visual to Textual Programming

Tomáš TÓTH[1], Gabriela LOVÁSZOVÁ[2]

[1]*Department of Informatics, Faculty of Economics and Management, Slovak University of Agriculture in Nitra, Slovak Republic*
[2]*Department of Informatics, Faculty of Natural Sciences, Constantine the Philosopher University in Nitra, Slovak Republic*
e-mail: ttoth@uniag.sk, glovaszova@ukf.sk

Received: June 2020

Abstract. In education, we have noticed a significant gap between the ability of students to program in an educational visual programming environment and the ability to write code in a professional programming environment. The aim of our research was to verify the methodology of transition from visual programming of mobile applications in MIT App Inventor 2 to textual programming in the Android Studio using the Java Bridge tool as a mediator of knowledge transfer. We have examined the extent, to which students will be able to independently program own mobile applications after completing the transition from visual to textual programming using the mediator. To evaluate the performance of students, we have analysed qualitative data from teaching during 1 school year and determined the degree of achievement of educational goals according to Bloom's taxonomy. The results suggest that students in the secondary education can acquire advanced skills in programming mobile applications in a professional programming environment, when they have knowledge of visual programming in an educational programming environment, and a suitable mediator is used to transfer such knowledge into a new context.

Keywords: knowledge transfer, mobile applications, teaching programming.

1. Introduction

Programming is usually included in the educational content of Informatics teaching if the goal of teaching is not only to acquire computer skills, but also to learn the foundations of Informatics as a science. Hromkovič and Steffen (2013) reason why it is important for the teaching of Informatics in education systems to include real Informatics with its fundamental concepts. Such Informatics teaching contributes to the understanding the contemporary world; it contributes to the development of ways of thinking and problem solving and prepares for further study.

Programming, as one of the topics of Informatics teaching, makes it possible to understand the fundamental principle of computer science that information can be stored as data and can be automatically processed by computer programs. Programming has the characteristics of a fundamental concept as defined by Schwill (1994). Programming is widely applicable in many contexts both within and outside computer science and integrates a wealth of phenomena (horizontal criterion). Programming can be taught on almost every level of understanding (vertical criterion). In Slovakia, the theme of algorithmic problem solving and programming at the age-appropriate level is integrated into the curriculum at all levels of compulsory education (primary, lower secondary and upper secondary education). Algorithmic problem solving belongs to the field of ordinary intuitive thinking (criterion of sense) and programming teaching turns it to a more exact position. Another feature of programming fundamentality within the area of Informatics is the historical aspect (criterion of time). Programming can be clearly seen in the historical development of computer science and is relevant in the long term.

In the historical context of the programming development, we can see a paradigm shift from programming as mathematical science to programming as engineering science. Programming does not only or mostly mean putting together an abstract algorithm to solve a problem, but also its implementation, testing, installation, and therefore development of a real product, which is an executable program, whose behaviour corresponds with the required specification. A programming language and a programming environment play an important role in such a concept of programming.

In the literature overview (Tóth, 2017), we have reviewed 37 scientific articles dealing with programming teaching. Articles have repeatedly stated that programming is difficult to learn, but also to teach. Lack of experience with problem solving, problems with imagining of abstract concepts, with basic concepts of algorithmization, as well as problems with syntax and semantics of a programming language were mentioned as difficulties in the beginning of programming learning. The most common problems with programming teaching were related to inappropriate teaching methods, poor interaction in classrooms, lack of interest, motivation, or even student frustration.

For these reasons, it makes sense to look for teaching means and methods that are helpful in overcoming the encountered difficulties. This means increasing student engagement by linking programming with their interests and using educational programming environments that have been developed to simplify the process of algorithm implementation into the resulting executable program. Current trends support the use of visual programming environments (e.g. Scratch, Alice, etc.) as well as software solutions supported with tangible interfaces (App Inventor, micro: bit, OzoBlockly or other various tools for programming robotic kits).

Although visual programming environments and visual programming are considered appropriate in the beginnings of programming learning, in the long run, their advantages may turn into disadvantages for students over time. In our exploratory research (Tóth and Lovászová, 2018), we have found that the ease of programming of mobile applications in the App Inventor visual programming environment has been later limiting for students. Students were limited by possibilities of using and setting

properties of components. Modification of code compiled from graphic blocks is also less flexible, and due to dragging and dropping of graphic blocks also lengthier than modification of textual code. Moreover, clarity and readability of complex programs is worse. Students thought they had mastered programming in App Inventor and it was no intellectual challenge for them anymore. They demanded a more professional way of programming used in practice for professional application development.

The transition from visual programming to textual programming is a natural step for a student in programming learning in order to continuously acquire new programming knowledge and skills. Through this transition, the student makes progress from the basics of programming to the advanced programming skills. However, we have noticed that the transition of students from the App Inventor visual programming environment to the Android Studio textual programming environment is a non-trivial process. We have noticed a significant "gap" between these two methods of programming, which is a challenge for both the student and the teacher. While this has a negative impact on the students' performance, students' motivation was not impacted and students' interest in a professional way of programming remained.

Although several authors have already addressed the issue of the students' transition from a visual programming environment to a textual programming environment in the process of programming teaching, this research area is still not sufficiently covered.

Perkins and Salomon (1988) have dealt with the transfer of something that was learned in one context to another context. They specify two types of transfers, namely:

- **Low road transfer** – in the case of a new context that has several obvious similarities to the old context, the new context almost automatically activates behaviour patterns that were satisfactory in the old context.
- **High road transfer** – it is based on the purposeful abstraction of knowledge and skills from one context, when applied in another, more distant context, where the similarity is not entirely clear.

The authors also deal with the possibilities of how to encourage such transfer for students. They call this process mediated transfer. They specify two techniques that support such a transfer:

- **Hugging** – is creating such connections between contexts, when teachers introduce a new educational situation which is similar to some of the previous ones (they try to "hug" it as closely as possible – to get close to it) and it is also related to the already existing experience of students and thus creates conditions for low road transfer.
- **Bridging** – is a process, when a teacher points out parallels between elements of content and facilitates the process of abstraction necessary for the high road transfer.

Cheung *et al.* (2009), who see a "gap" between a visual programming environment and a textual programming environment in the teaching of high school students at grades 11–13, have dealt with the issues of the transition phase. The authors state that students often find a visual programming environment too restrictive on the one hand, and a textual programming environment too difficult on the other. They address this gap

by creating a custom textual-graphical hybrid programming environment BrickLayer. A program is created by dragging and dropping graphic blocks in the BrickLayer programming environment. But at the same time, the syntax of the textual representation of the created program is displayed in real time. Students therefore can also see what happens in the textual form of the created program in an interactive way when arranging blocks. The results of the authors show that the textual-graphical hybrid programming environment has a positive impact on students' learning experience.

A similar problem has also been encountered by Dolgopolovas *et al.* (2017), in whose research students had difficulties when transitioning from visual programming in the MIT App Inventor 2 development environment to textual programming in the C programming language. According to them, the problem is with the different paradigms that are used in these environments (event-driven programming vs. structured programming), the different teaching approaches that are traditionally used (game creation vs. the mathematically-oriented approach) and differences in the use of different programming concepts.

Dann *et al.* (2012) have addressed the transition phase by creating the Alice 3 programming environment and a plugin for the NetBeans IDE. Alice 3 provides several features to support students in the transition to the Java programming language. One of the features displays the generated code in the Java programming language alongside with an algorithm created in the Alice 3 programming environment. It also allows exporting the code from the Alice 3 programming environment into the NetBeans IDE.

Armoni *et al.* (2015) investigated the transition from the Scratch visual programming environment to professional textual programming languages (C#, Java). They have found that programming knowledge as well as knowledge acquired by students who have learned Scratch has facilitated learning of advanced educational topics. However, difficulties arose when concepts were implemented differently in programming languages (defining the data type of a variable, using full and incomplete branching, and bounded loops). Despite these facilitations, and also the challenges at the end of the teaching process, there were no significant differences in the success of these students in comparison with students who have not learned Scratch. On the other hand, students have shown a higher level of motivation and self-sufficiency.

Hsu and Ching (2013) have stated that when working with the MIT App Inventor 2 programming environment, more advanced students missed the ability to edit textual code in the Java programming language (i.e. the native programming language used for the development of mobile applications for the Android OS) through the MIT App Inventor 2 programming environment.

Chadha and Turbak (2014) have created an extension for MIT App Inventor 2 called TAIL (A Textual App Inventor Language). TAIL is an isomorphic textual programming language with a visual programming language of the MIT App Inventor 2 programming environment. It enables bidirectional conversion between graphic blocks and text fragments. TAIL improves the usability of MIT App Inventor 2 by simplifying readability, writing, sharing and copying of programs thanks to the textual representation of the program. At the same time, it can also help in the transition from a visual programming language to a textual programming language.

Wagner *et al.* (2013) present their approach to students' transition from a visual programming language in the MIT App Inventor 2 environment to the Java textual programming language. For this transition from visual programming to textual programming, they used the Java Bridge library, which allows using textual programming to create programs that are equivalent to programs created in App Inventor. The authors have noted that using a visual programming language first and then displaying a direct mapping to an equivalent version in the Java programming language code has helped students to understand application programming in the Java programming language.

2. Visual vs. Textual Programming

In textual programming, a program is created by writing textual code, which can be challenging for students. When writing a program, they must focus not only on the content of the solution – an algorithm, but also on the syntactically correct formal notation of the algorithm in the programming language. However, the disadvantage of more demanding program creation can be overcome by students' feeling that they are working with a professional tool. Textual programming can be motivating for students with higher expectations and needs.

In visual programming environments, a program is created by arranging pre-prepared graphic blocks that make it impossible to compose a syntactically incorrect program. A student does not have to know language commands (they are selected from a menu) and does not have to solve problems with possible syntax errors. All this simplifies the application development process. This makes programming more accessible for the general public and not just for a narrow circle of professional programmers.

2.1. *App Inventor vs. Android Studio*

Our area of interest is mobile application programming. Programming for a mobile platform increases engagement of students, because it is connected to the object of their interest – a mobile device. It is motivating for students to create a mobile application that they can use in their own smartphone also outside the classroom and show it to their relatives and friends.

Currently, the standard way to develop mobile applications for the Android operating system is native development using the Java or Kotlin textual programming language, Android SDK and the Android Studio integrated development environment. This option is associated with professional application development, requires advanced development skills and therefore exceeds the skills of students beginning with programming.

Visual programming using the MIT App Inventor 2 development environment created specifically for educational purposes is easier to use for beginners. It is a popular cloud solution that is used online through a web browser. App Inventor is a visual development environment providing all the advantages as well as disadvantages resulting from visual programming described above. A more detailed analysis of the MIT App

Inventor 2 programming environment, its possibilities and limitations in comparison with the Android Studio development environment are discussed in more detail in the article (Tóth, 2019).

2.2. *Java Bridge – A Mediator in Moving from Visual to Textual Programming*

The problematic transition of students from the development of mobile applications in App Inventor to professional development using textual programming can be simplified by using the Java Bridge support tool.

The term Java Bridge includes Java Bridge Library and Java Bridge Code Generator:

- **Java Bridge Library** is a Java programming language library designed for creating applications for the Android OS. Its purpose is to simplify textual programming of applications for the Android OS in comparison with the Android SDK. The Java Bridge library uses the same terminology as MIT App Inventor 2 – there is a Java class for each component. Students can use this library directly in the Android Studio programming environment, where they can use textual programming to create applications for the Android OS (AppInventor, 2020a).
- **Java Bridge Code Generator** is a version of the MIT App Inventor 2 programming environment that allows students to create an application just like in the MIT App Inventor 2 programming environment and then generate an equivalent version of the application in the Java programming language. The Java Bridge code generator is intended to help students who already know how to create applications by arranging blocks in the MIT App Inventor 2, when moving to textual programming. Generated applications use the Java Bridge library and can be edited in the Android Studio programming environment (AppInventor, 2020a; AppInventor, 2020b).

With a correct proposal of teaching methodology using these tools, a smooth transfer of student knowledge from one context (visual programming in App Inventor) to another (textual programming in the Java programming language and the Android Studio development environment) can be mediated (Fig. 1).

The methodology has three phases:

- **Visual programming.** Students program their applications visually in the MIT App Inventor 2 educational programming environment.

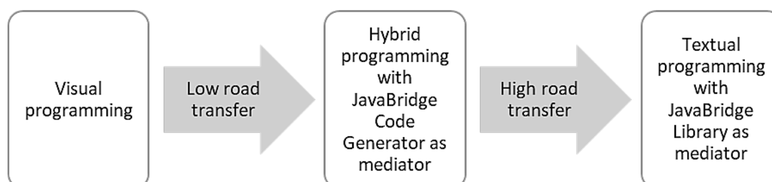

Fig. 1. Methodology of knowledge transfer from visual to textual programming of mobile applications.

- **Hybrid programming.** Students program by arranging graphic blocks in the Java Bridge Code Generator environment. They transfer the generated Java code to a prepared blank project in Android Studio. They get to know the textual equivalent of a visual code, and the Java programming language. They experiment with minor modifications to the textual code – changes to command parameters, analogical code extension. They gain experience in application development in the Android Studio environment.
- **Text programming.** Students program directly by writing text into a prepared blank project in Android Studio. They use an analogy to programming in App Inventor using classes from the Java Bridge library.

A low road transfer of knowledge takes place during the transition from the visual programming phase to hybrid programming. Students explore their program created by visual programming in a new context of textual programming. They experiment with textual program editing and find out how they can apply their knowledge in a new context of a textual programming environment.

Full transition to textual programming requires high road transfer of knowledge. It is based on the abstraction of knowledge from the old context. Students abstract their knowledge of the principle of mobile application creating in a visual programming environment and transfer it to writing a textual program. They directly write textual code without the need to create an equivalent program from graphic blocks first. The parallel between programming in App Inventor and Android Studio using the Java Bridge library is still used.

3. Case Study

3.1. *Research Objectives*

The aim of the research was to determine the effectiveness of the methodology of transition from visual programming of mobile applications in the MIT App Inventor 2 educational environment to textual programming in the professional environment of Android Studio using Java Bridge as a mediator of knowledge transfer. The research question we have studied was, to what extent students are able to independently create a mobile application using textual programming after completing the transition from visual to textual programming using a mediator.

The research design is based on the evaluation of students' ongoing and final performance in order to determine whether the teaching strategies used are appropriate to achieve the goals of education. Such an approach can also be found in other studies examining the effectiveness of certain teaching methods for reaching results, for example (Alexandron *et al.*, 2016), in which the authors study "how high-school students understand the concept of operative nondeterminism after learning the language of live sequence charts".

We have verified the methodology of transition from visual to textual programming of mobile applications in an optional hobby programming course for secondary education students during one school year.

3.2. *Sample*

A total of 21 students took part in the course, of which only 14 students regularly took part in the lessons; we excluded the remaining 7 students from the research sample. The students were aged 12–18 years, 13 male and 1 female. Students chose to attend the course as their leisure activity and participated in the course outside of compulsory education.

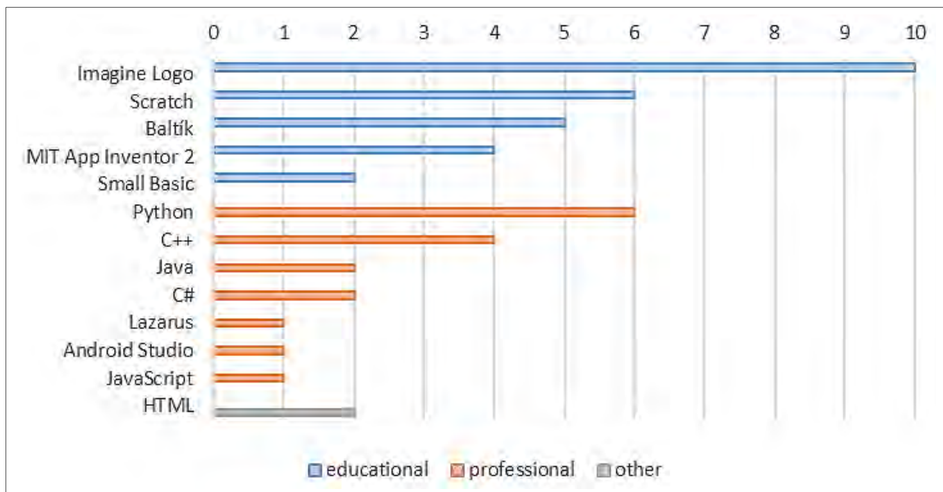

Fig. 2. Programming languages or environments with which students have experience.

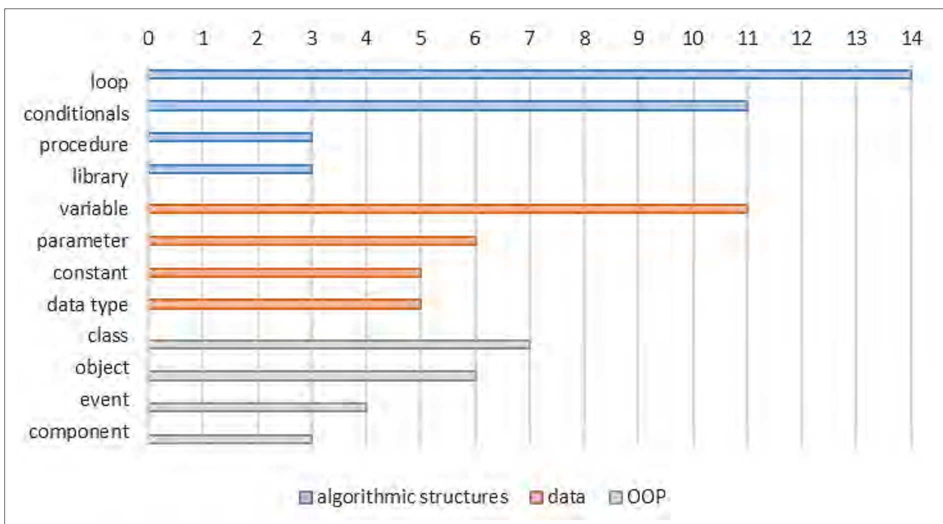

Fig. 3. Concepts from programming that students have already encountered.

In the entry questionnaire, they stated a positive attitude towards Informatics and programming. They all stated they enjoyed programming and 57% stated they would like to become a programmer in the future. All students already had experience with programming since in Slovakia, programming is a mandatory part of education already in lower secondary education, as a part of the Informatics curriculum. Most of the students stated in the questionnaire that they program 1 or 2 years, one quarter stated 3 or more years of programming experience.

In the questionnaire, students reported experience with educational programming environments for children used in schools in lower secondary education, older students already had experience with programming in other than educational programming environments (Fig. 2). When asked which concepts of programming are known to them, students mentioned in the questionnaire concepts from the area of algorithmic structures, work with data, object-oriented and event-driven programming. The questionnaire did not ascertain the level of understanding of the concepts. The answers are shown in Fig. 3.

3.3. *Instruments and Procedures*

The research was conducted during the school year 2018/2019. The course took place once a week and comprised two school lessons (total of 90 minutes). The course was led by one lecturer, who was also a researcher. The course schedule was divided into 3 stages in accordance with the proposed methodology in Fig. 1:

1. Visual programming stage.
2. Hybrid programming stage.
3. Textual programming stage.

The detailed schedule is shown in table (see Fig. 4). Ideas for applications that were programmed in the course are taken from the books Wolber *et al.* (2014) and Michaličková (2016), or the ideas of the teacher and the students themselves were used. Games but also “useful applications” that students can use in their real lives were themes for the applications.

The visual programming stage and the textual programming stage were concluded with an independent project of students according to their own idea or were chosen from a list. In the project, the students had to show what they had learned in the previous period and what application they are able to create independently. The table shows the complexity of solved projects (simple or complex) according to the number and complexity of specific teaching goals and the highest level of educational goals according to the revised Bloom Taxonomy (RBT).

During the lessons, in which the level of objectives according to the RBT was labelled as Applying, all students worked on the same assignment, which was selected with regard to the objectives to be achieved. The teacher used the work on the project to explain the principles of mobile application creation in a given programming environment, to present various programming techniques, the method of application debugging, testing, and compiling. The aim was to understand and apply the knowledge

<table>
<tr><th>Week</th><th colspan="3">Way of programming</th><th>Theme</th><th>Difficulty</th><th>Level of RBT</th></tr>
<tr><td>1</td><td rowspan="9">App Inventor (visual)</td><td colspan="2" rowspan="9"></td><td>Hello, World (V1)</td><td>Simple</td><td rowspan="4">Applying</td></tr>
<tr><td>2</td><td rowspan="3">Catch the egg (V2)</td><td rowspan="3">Simple/complex</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td rowspan="5">Project (V3)</td><td rowspan="5">Complex</td><td rowspan="5">Creating</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td></tr>
<tr><td>9</td></tr>
<tr><td>10</td><td rowspan="17"></td><td rowspan="9">Java Bridge Code Generator (hybrid)</td><td rowspan="17">Android Studio + Java Bridge Library (textual)</td><td rowspan="3">Hello, World (H1)</td><td rowspan="3">Simple</td><td rowspan="12">Applying</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td></tr>
<tr><td>13</td><td rowspan="3">ChatBot (H2)</td><td rowspan="3">Simple/complex</td></tr>
<tr><td>14</td></tr>
<tr><td>15</td></tr>
<tr><td>16</td><td rowspan="3">MoleMash (H3)</td><td rowspan="3">Complex</td></tr>
<tr><td>17</td></tr>
<tr><td>18</td></tr>
<tr><td>19</td><td rowspan="8"></td><td>Hello, Purr (T1)</td><td>Simple</td></tr>
<tr><td>20</td><td rowspan="2">Roll the dice (T2)</td><td rowspan="2">Simple</td></tr>
<tr><td>21</td></tr>
<tr><td>22</td><td rowspan="5">Project (T3)</td><td rowspan="5">Complex</td><td rowspan="5">Creating</td></tr>
<tr><td>23</td></tr>
<tr><td>24</td></tr>
<tr><td>25</td></tr>
<tr><td>26</td></tr>
</table>

Fig. 4. Course schedule.

when solving the assignment by creating a functional application. In addition to personal assistance of the teacher, students had printed handouts containing requirements for application functionality, a preview of the application graphical user interface (GUI), an outline of the solution in the form of a list of sub-tasks, an outline of the project structure with a list of components to be used, and tasks extending the original assignment. The use of printed learning materials supported the students in active independent work on projects and allowed differentiation according to students' abilities. During the lessons, in which the students worked on their own individual projects, the level of RBT goals is higher (Creating). Students did not have a formal description of the final product and a sketch of the solution, they had to analyse the problem themselves, to propose the application structure, and how to implement its functionalities. They used handouts for projects from previous lessons, information from the Internet and individual help from the teacher as teaching material.

The whole teaching process was constantly reflected and corrected based on the results of formative assessments. Continuously throughout the course, we were surveying, how students understand the taught concepts and how they perceive the course management (e.g. using handouts, etc.). The formative assessment was carried out using the Socrative voting system. Students answered questions in an interactive questionnaire aimed at understanding the curriculum, but also to express their opinion on teaching methods. The questionnaire was filled by all students at the same reserved time during the lesson. After answering each question by all students, their anonymized answers were projected on the screen and discussed with the lecturer.

The total achieved results of students were verified and evaluated on the basis of solution of their own individual project. Such summative evaluation took place at the end of the visual programming stage and at the end of the textual programming stage. The creation of both projects ended with its presentation by the author via a data projector on a screen in front of other students.

3.4. *Methods of Data Collection and Processing*

We have used several methods of qualitative research for data collection: unstructured observation, participatory observation, informal interviews with students, problem-solving interviews, focus groups, questionnaires, field notes and product collection (student-created applications). In this way we have obtained data of three types:

1. Data from observations and interviews – after each lesson, the data were converted into text in the form of protocols containing field notes from teaching.
2. Data obtained in writing from questionnaires.
3. Data obtained in the form of collected products – code of applications created by students.

Methods of data collection and processing to assess student performance are listed in Table 1.

Table 1

Student performance assessment

	Operationalization	Assessment	Data collection	Data processing
Applying	understand the principles of mobile application creating, apply the techniques of programming, debugging, testing, application compiling according to an example	Formative – to optimize the teaching process	observation, individual consultations, ongoing questionnaires, product analysis	Qualitative, Quantitative
Creating	analyse a problem, design an application structure and how to implement its functionalities, create a functional application	Summative – to determine the degree of achievement of educational goals	product analysis	Quantitative

We have analysed in detail the source code of applications created by students, and we have quantified the data (have converted it into a numerical form). The complexity of the project was quantified as the sum of programming and technical complexity. The programming complexity is represented by activities related to the design of the application structure and the creation of program code to ensure all the functionalities of the application according to the specified requirements. Into the technical complexity we have included activities related to the implementation of the solution in the given programming environment, the result of which is the creation of a real product (mobile application running on a mobile device).

For each project, specific goals were formulated in detail in the form of student performance, the achievement of which was expected from the solution of the project. On this basis, it was possible to assign numerical values to individual student projects according to which goals and to what extent the students managed to meet. We have quantified the performance of students, or the success of learning, from such numerically evaluated student projects.

4. Research Results

4.1. *Qualitative Findings*

Qualitative findings are based on the evaluation of data from observations and quizzes to determine the level of achievement of goals at lower levels of RBT (memorization, comprehension, application) in the phases, when students worked with the help of a lecturer on the same projects and used handouts with sketches of the solution process. Using a questionnaire, we have also found out how students themselves evaluate their ability to program mobile applications after completing the course.

Visual programming phase: In the phase of work in the App Inventor visual programming environment, we have observed very fast progress. Several students had previous experience with visual block programming in the Scratch environment. They have noticed similarities between the two environments:

> *"App Inventor is similar to Scratch. We have to arrange blocks as well."*

Scratch programming experience has helped students to understand and apply knowledge in a new context in App Inventor. An example is the creation of custom procedures. Scratch programming experience, where there was no way to define custom procedures in version 2, was essential to understand the importance of using custom procedures:

> *"Several times, I programmed something in Scratch, then I wanted to change it, and I had to change it everywhere, which was difficult. Using a procedure would made it easier."*

At this stage, we did not observe problems with memorizing and understanding the syntax (programs are created by selecting, dragging and dropping blocks from a menu), or with algorithmic problem solving. Students were able to work independently on the basis of their previous knowledge of programming and according to instructions in handouts, which contained a rough outline of the solution without details. Students' solutions therefore differed in details, even though they worked on the same assignment. The ongoing formative assessment through interactive quizzes also did not reveal problems in understanding the principles of programming in App Inventor. Problems that occurred were mainly of a technical nature and were not related to programming:

- Problems with exporting files with application source code, and building the application.
- Problems with differentiating a source code file (aia) and an application installation package (apk).
- Problems with remembering how to upload files to cloud storage.

Again, the findings from the preliminary research were confirmed, i.e. that the feeling of success is no sufficient motivation for programming mobile applications in App Inventor, although there were still many things that the students could learn in it. When expressing their relationship to visual programming in App Inventor, students were rather reserved or directly expressed their ambition to program in a professional environment, because they considered App Inventor to be *"childish"*.

Hybrid and textual programming phase. The transition to textual programming through the generation of textual code from visual code made it possible for all students to move to textual programming. However, with further progress, there was a more significant differentiation between students:

- Some could use an analogy for direct text extension of the generated code.
- Others still helped themselves by generating textual code from the visual code.
- Some still preferred visual programming.

At this stage, we observed a lower degree of independence of students' work. Only some students knew how to proceed independently. The higher complexity was not caused by the textual notation of a program, but rather by the complexity of the development environment. When writing a textual code, the occurrence of syntax errors was sporadic. The problems were in particular:

- With orientation in the file structure of a project in Android Studio.
- With application building.
- With application startup on a mobile device.

When solving a complex assignment in week 16–18 (see Fig. 4), some students evaluated textual programming in Android Studio as too demanding in relation to their abilities, and some expressed even frustration from failure:

> *"Difficulty of App Inventor is closer to my intellectual level. Android Studio is difficult."*
>
> *"I am lost in Android Studio."*

When the teaching method changed from independent programming according to a handout to the frontal explanation of the lecturer (gradual solution with explanation), all students managed to complete the application and the mood in the classroom also improved.

At the end of the course, the students evaluated the difficulty of programming in Android Studio on a five-point scale as follows: mostly neutral (7/12), 1 student said that programming in this environment is rather difficult (1/12), others no longer considered programming in Android Studio difficult: rather not (2/12) or not at all (2/12). When evaluating what is difficult on mobile applications programming in Java and Android Studio, students reported the Java language syntax (*"the Java language and its lengthy and complex syntax"*) and work with project structure (*"rather lengthy than demanding: preparation of a project and Java files, subsequent editing and occasional clutter.*")

Self-assessment: In the final questionnaire, we investigated the attitudes of students towards various issues related to the completed course. In a self-assessment of their ability to program mobile applications using textual programming, students stated:

- Ability to program the application completely independently: 4/14.
- Ability to program an application with the help of:
  - a more experienced person (e.g. teacher) 7/14
  - Internet: 7/14
  - code generation: 5/14
  - handout: 5/14

No student has reported the inability to program a mobile application with textual programming in the self-assessment. Out of the forms of help that were a part of the teaching methodology, the students mostly specified the help of the teacher as the condition of success, followed equally by the generation of textual code from visual code using a mediation tool, and teaching material – handout. From observations of students' work and interviews, we have noticed that they used mainly the information from the Internet as another form of help. The students stated this despite the fact that information found on the Internet did not usually deal with the use of the Java Bridge library. Students did not know how to distinguish this sufficiently or apply it in their own projects. The lecturer had to draw the students' attention to this fact. Nevertheless, exactly half of the students (7/14) stated in the final questionnaire that the information found on the Internet was helpful.

4.2. *Quantitative Findings*

Quantitative findings are based on the evaluation of the complexity and goal fulfilment rate in programming products created and submitted by the students in the course. The methodology of data processing is specified in Section 3.4 Methods of Data Collection and Processing.

Each stage (visual, hybrid and textual programming) consisted of creating three applications, so the students worked together on 9 projects. We categorized the projects (see also Fig. 4) to:

- Visual / Textual – according to the method of programming, to projects created using visual (projects V1, V2, V3) and textual programming, including hybrid programming (projects H1, H2, H3, T1, T2, T3).
- Applying / Creating – according to the level of educational objectives, to projects requiring understanding and application of knowledge (projects V1, V2, H1, H2, H3, T1, T2), and to projects requiring independent creation, including problem analysis and solution proposal (projects V3, T3).

Table 2 shows the results of the evaluation of students' products from individual categories from three points of view:

- Assignment difficulty – overall / programming,
- Solution success rate – achievement of goals in percentage,
- Weighted performance – the product of the coefficient of project complexity (ratio of project complexity to average complexity) and solution success rate.

The assignment difficulty and the solution success rate are shown in more detail in Fig. 5 for each project.

The Fig. 5 shows that:

F1 Applications created at each stage of the programming method (visual, hybrid, textual) had increasing difficulty score, and after the transition to a new programming method, the difficulty of the project was reduced.

F2 The success rate of projects decreased with increasing difficulty, with the exception of the first project in the hybrid programming stage (H1), where the solution success rate decreased despite the decrease of difficulty.

F3 The technical difficulty of the projects is the lowest for visual programming (on average 5.58), the highest for hybrid programming (on average 11.33), and has decreased again for textual programming (on average 8.37).

Fig. 6 shows in more detail and compares the solution success rate for programming projects V3 in App Inventor and T3 in Android Studio, which were independently created by the students according to their own ideas.

Table 2

Results of quantitative evaluation of student products

	Assignment difficulty (total / programming)		Solution success rate (%)		Weighted performance	
	Applying	Creating	Applying	Creating	Applying	Creating
Visual	18.5 / 12.5	21.92 / 17.17	81.11	77.78	0.8071	0.8623
Textual	20.2 / 10.2	18.00 / 10.88	71.04	55.04	0.7008	0.5012

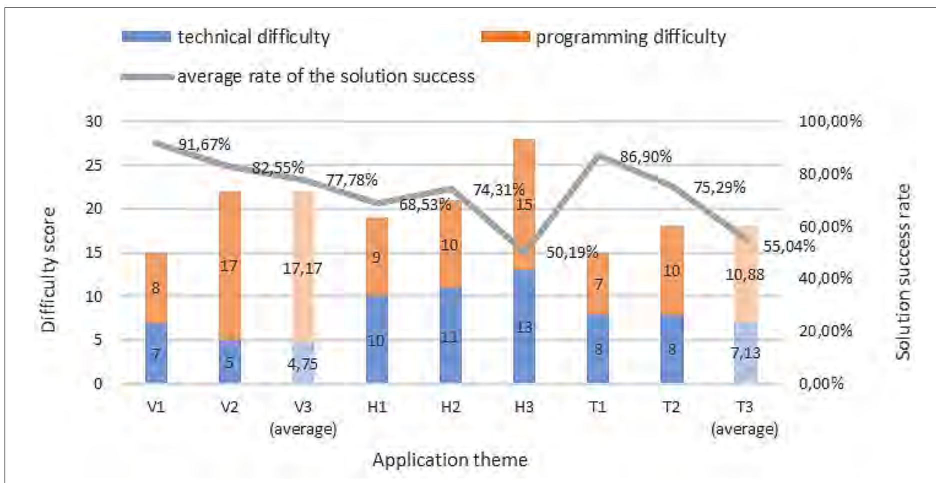

Fig. 5. Difficulty of application topics vs the average solution success rate of applications.

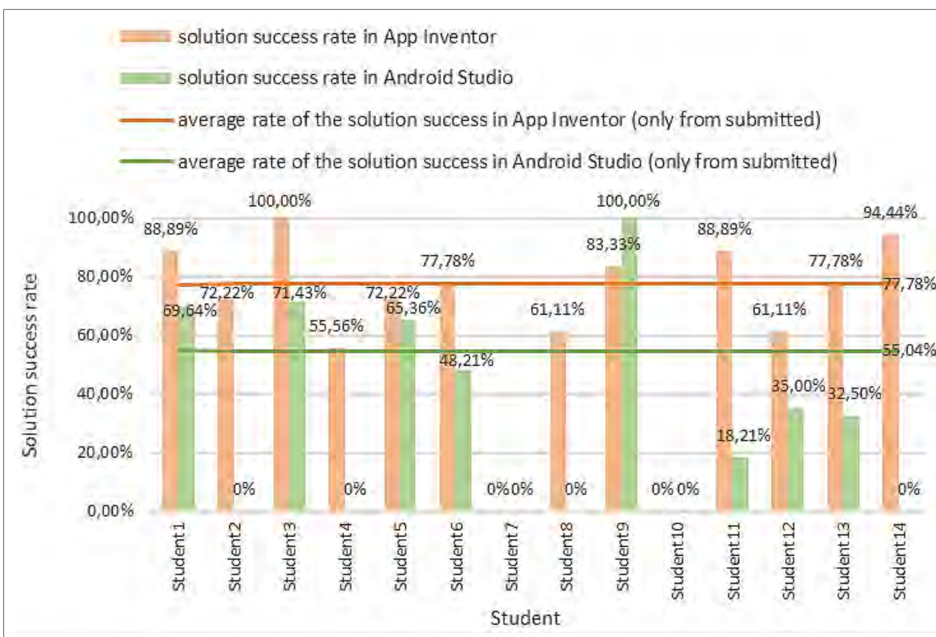

Fig. 6. Solution success rate of individual projects in App Inventor and in Android Studio.

The Fig. 6 shows the following findings:

F4 The solution success rate for application creation using visual programming is higher than using textual programming for most students (with the exception of Student9), also on average.

F5 There were more successfully completed and submitted projects for visual programming than for textual programming (12 vs. 8).

F6 If we consider 50% fulfilment of the goals to be a success, all students who submitted the project were successful in visual programming, half of the students (4/8) were successful in textual programming – one slightly and three significantly below the success limit.

A similar trend can be observed when comparing the difficulty score achieved by students' own projects according to their own ideas in App Inventor (V3) and Android Studio (T3) (Fig. 7):

F7 A standalone T3 project created in Android Studio has a higher score of technical difficulty than an independent project V3 created in App Inventor for all students.

F8 In contrast to F7, the programming difficulty of the T3 project is lower than the programming difficulty of the V3 project for all students with one exception (Student9).

F9 The total difficulty score of the T3 project is lower for each student than in the V3 project, with one exception (Student9).

F10 The average score of the achieved total difficulty is higher in the visual programming stage than in the textual programming stage.

The synthesis of both indicators of student performance evaluation (application difficulty and solution success rate) represents the weighted student performance as

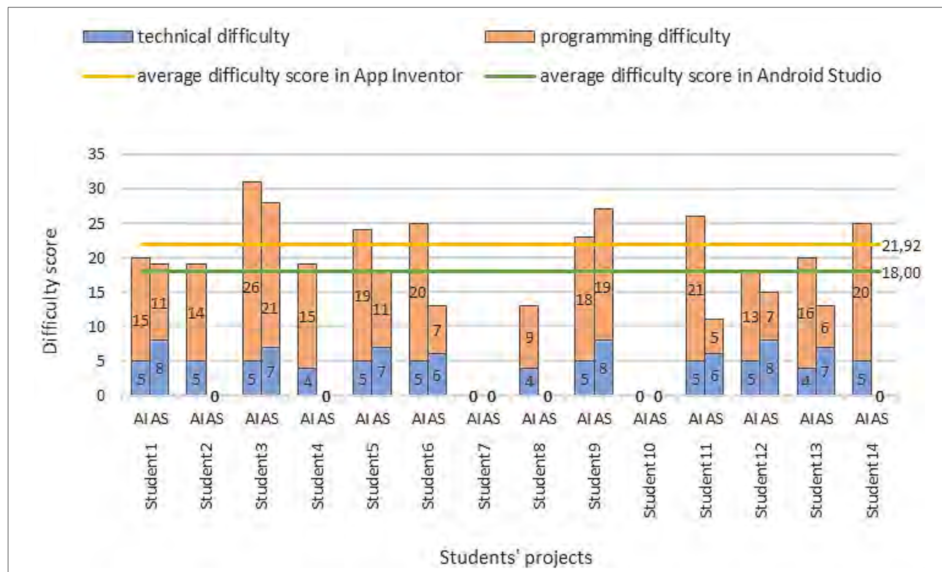

Fig. 7. Comparison of difficulty of individual projects created in App Inventor and in Android Studio.

the product of both indicators. When examining the weighted performance (Fig. 8), we came to the following findings:

F11 Significantly higher values of weighted performance are in the visual programming stage than in the textual programming stage (including hybrid programming): 0.8071 vs. 0.7008.

F12 In all three stages, there was an increase in weighted performance after programming the initial application (biggest in the visual programming stage 0.2231) and a subsequent decrease when programming complex projects (biggest in the textual programming stage 0.1843). We observe a significant decrease below the level of the weighted performance of the initial application (T1) in the textual programming stage.

F13 If we observe a decrease in weighted performance during the transition from one stage to another, we observe a significantly higher decrease in weighted performance between visual and hybrid programming (decrease of 0.2036) than during the transition to pure textual programming (decrease of only 0.0515).

F14 The achieved value of the average weighted performance of students for each initial application in all three stages (V1, H1, T1) has only a slight deviation (0.6956 vs. 0.6587 vs. 0.6594).

Other important findings can be observed in the measured values of average weighted performance in the case of individual student projects (V3 and T3):

F15 Students achieved **significantly higher** average weighted performance in independent programming of own applications in the visual programming stage

Fig. 8. Average weighted performance of students with respect to the coefficient of difficulty of application created.

(V3) compared to the average weighted performance in the textual programming stage (T3).

F16 The average weighted performance achieved in individual projects in the visual programming stage (V3) is **higher** than the average value of the weighted performance that students achieved in applications programmed together with the lecturer during the first stage (V1, V2).

F17 The average weighted performance achieved for individual projects in the textual programming stage (T3) is significantly **lower** than the average value of the weighted performance that students achieved in applications programmed together with the lecturer during the second and third stage (H1-T2).

F18 While the average weighted performance in the case of an individual project in the V3 visual programming stage is only slightly lower than the average weighted performance achieved in previous application V2 programmed together with the lecturer (the difference of 0.0564), the decrease in the average weighted performance in the textual programming stage between the T3 individual project and the previous application T2 created together with the lecturer is more pronounced (the difference of 0.1843).

5. Discussion

The aim of the research was to find out to what extent students are able to independently create a mobile application using textual programming after completing the transition from visual programming in App Inventor to textual programming in Android Studio using the Java Bridge mediator, and thus how successful the methodology of mediated knowledge transfer using a mediator was.

The use of the App Inventor visual programming environment in the introduction to programming of mobile applications can be considered a suitable choice. Visual programming environments do not burden programmers with so many technical aspects of development – the lowest technical complexity was measured in the visual programming stage (finding F3). In this programming environment, students can make rapid progress in creating functional mobile applications. This is evidenced by the high increase in the average weighted performance of students in programming at the beginning of the course (finding F12) and the high weighted performance also when creating an individual project (finding F16).

The transition to textual programming represents an increase in the overall complexity of programming. The aim of the used teaching methodology was to bridge the gap between the complexity of visual and textual programming by including the stage of hybrid programming. Research results show that this transition is still challenging. The difficulty of the transition between the visual and hybrid programming was reflected by the increase in the technical complexity of programming in the hybrid programming stage (finding F3). The reason was the introduction of another programming environment into teaching – Android Studio, while still using the visual programming environment. At this stage, there was also a significant decrease in weighted performance

(finding F13). However, the benefit of using a mediator was subsequently reflected in the transition to textual programming, where the decrease in weighted performance was only minimal. The success of bridging the gap between the visual and textual programming is confirmed by finding F14. In all three stages, the weighted performance in programming of initial applications was maintained at approximately the same level.

We have evaluated the ability to independently design and create a mobile application based on the results achieved in the creation of individual projects (V3, T3). Using visual programming, students achieved significantly higher average weighted performance than using textual programming (finding F15). This result in favour of visual programming was influenced by both components of weighted performance – difficulty and success rate. Students were able to create more complex projects with higher overall and programming difficulty (findings F8, F9), and were more successful in completing their projects (findings F5, F6). The difference in students' performance can be explained by higher technical difficulty of programming in Android Studio (finding F7). The authors of projects programmed in Android Studio, which had less than a 50% solution success rate, were the youngest students aged 12, 13 and 14 years. It turns out that textual programming is especially challenging for students in lower secondary education. On the other hand, in applications that these students programmed in a text-based way together with the lecturer, they managed to achieve a solution success rate in the range of 83.75%–100%. Although students were able to emulate the work of a lecturer in jointly created applications, non-specific knowledge transfer could not be fully achieved and students could not independently apply knowledge in textual programming. Since each student is an individual personality, this finding may not generally apply to all students at this age. This is also confirmed by exceptions in our sample – more than half of the solution success rate for a 14-year-old student and less than half of the rate for a 17-year-old student.

The results obtained in our research match the final statement of the researchers Wagner *et al.* (2013) who found that when programming teaching starts with a visual programming language and then equivalent Java code is presented using direct mapping (pointing to context), it can help students to understand programming techniques in programming mobile applications in the Java programming language. Moreover, we extend this result with other specified facts.

The limitation of our research is the group of research participants – students with whom the research was carried out. It is a selection of students with increased interest in computer science and programming. This fact may affect the achieved results, which cannot be generalized.

6. Conclusions and Implications

The aim of our research was to apply the methodology of transition from visual programming of mobile applications in the App Inventor educational programming environment to textual programming in the Android Studio professional programming environment using Java Bridge as a mediator of knowledge transfer on a sample of

students. We were interested in the extent to which students were able to create mobile applications using text programming after completing the transition from visual to textual programming using a mediator.

Based on quantitative evaluation of the collected data we found that the weighted performance of students in the examined sample, which combines difficulty and success rate, was on average 0.7 in the case of programming according to the instructions and using analogy, and on average 0.5 in the case of independent creative programming. In terms of the ability to independently create mobile applications using textual programming in Android Studio, large differences in students' performance, both based on quantified evaluation of created projects and self-evaluation of students' own programming skills were noted. Nevertheless, we can state a certain (lower or higher) level of success and a positive attitude towards textual programming in the Android Studio for all students.

Regarding the comparison of students' visual and textual programming skills, the degree of understanding the principles of mobile application creation and the degree of ability to analyse a problem, creatively design a solution, apply programming techniques and create a functional application was significantly higher in the case of visual programming in App Inventor than in textual programming in Android Studio. In general, students were able to deliver higher performance using visual programming and App Inventor.

In spite of that, we consider the methodology of transition from visual to textual programming using a mediator to be successful. Students were able to design non-trivial applications in Android Studio using analogy with programming in App Inventor. Through the Java Bridge library, students were able to write textual code containing the component hierarchy and terminology known from App Inventor.

Java Bridge Code Generator was helpful in bridging the gap between visual and textual programming. On the other hand, it complicates the transition to some extent by using two programming environments at the same time. Students perceived hybrid programming and combined work with both programming environments as complicated and hindering. With gradual improvement of skills in textual programming and with the intent to simplify their work with multiple programming environments, they naturally moved to textual-only programming. Every student who subjectively perceived that he/she had already mastered textual programming made effort to make this natural transition. This applied to 50% of students. A smoother transition to textual programming was also made possible by the use of the Java Bridge library as a mediator in textual programming (this is not a standard development method used in professional development). However, in the case of the youngest students (12–14 years old), we have noticed that textual programming is difficult for them and they preferred rather visual programming.

The results of our research can be considered the first step in exploring knowledge transfer in the transition from visual programming in App Inventor to text programming in Android Studio. We focused on supporting the transition through the Java Bridge library and the Java Bridge Code Generator. The next step would be to continue research focused on the subsequent transition to the native development of mobile applications without the use of the Java Bridge library as a mediator.

Acknowledgments

The article was created thanks to the support of KEGA project 018UMB-4/2020 Implementation of new trends in computer science to teaching of algorithmic thinking and programming in Informatics for secondary education.

References

Alexandron, G., Armoni, M., Gordon, M., Harel, D., (2016). Teaching nondeterminism through programming. *Informatics in Education*, 15(1), 1–23.

AppInventor. (2020b). *App Inventor Java Bridge*. `http://www.appinventor.org/jbridge`

AppInventor. (2020a). *Java Bridge Programming*. `http://www.appinventor.org/jBridgeIntro`

Armoni, M., Meerbaum-Salant, O., Ben-Ari, M. (2015). From Scratch to "Real" programming. *ACM Transactions on Computing Education*, 14(4), 25.

Chadha, K., Turbak F. (2014). Improving AppInventor usability via conversion between blocks and text. *Journal of Visual Languages and Computing*, 25, 1042–1043.

Cheung, J., Ngai, G., Chan, S. And Lau, W. (2009). Filling the gap in programming instruction: a Text-enhanced Graphical Programming Environment for Junior High Students. *ACM SIGCSE Bulletin*, 41(1), 276–280.

Dann, W., Cosgrove, D., Slater, D., Culyba, D. And Cooper, S., (2012). Mediated transfer: Alice 3 to Java. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education – SIGCSE '12*, 141–146.

Dolgopolovas, V., Jevsikova, T., Dagienė, V. (2017). From Android games to coding in C-An approach to motivate novice engineering students to learn programming: a case study. In: *Computer Applications in Engineering Education*, 1–16.

Hromkovič, J., Steffen, B. (2011). Why Teaching Informatics in Schools Is as Important as Teaching Mathematics and Natural Sciences. In: *ISSEP 2011: Informatics in Schools. Contributing to 21st Century Education*, 7013, Berlin, Heidelberg, 21–30.

Hsu, Y. -C., Ching, Y. -H. (2013). Mobile app design for teaching and learning: Educators' experiences in an online graduate course. *The International Review of Research in Open and Distance Learning*, 14(4), 117–139.

Michaličková, V. (2016). *Programovanie Mobilných Aplikácií v Prostredí MIT App Inventor 2*. Univerzita Konštantína Filozofa v Nitre. ISBN 978-80-558-1101-7.

Perkins, D. N., Salomon, G. (1988). Teaching for transfer. In: *Educational Leadership*, 22–32.

Schwill, A. (1994). Fundamental ideas of computer science. *Bull. European Assoc. for Theoretical Computer Science*, 53, 274–295.

Tóth, T. (2017). Current trends in teaching of introductory programming: A literature review and research directions. In: *Proceedings of 10th International Conference of Education, Research and Innovation, ICERI 2017*, Seville, Spain, 4852–4862.

Tóth, T. (2019). App inventor vs professional application development: A comparative study. In: *Proceedings of 10th International Conference of Education, Research and Innovation, ICERI 2019*, Seville, Spain, 4447–4456.

Tóth, T., Lovászová, G. (2018). On difficulties with knowledge transfer from Visual to Textual programming. In: *DIVAI 2018 – The 12th International Scientific Conference on Distance Learning in Applied Informatics. Conference Proceedings*. Wolters Kluwer ČR, a. s., 379–386.

Wagner, A., Gray, J., Corley, J., Wolber, D. (2013). Using app inventor in a K-12 summer camp. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education – SIGCSE '13*, 621–626.

Wolber, D. Abelson, H., Spertus, E., Looney, L. (2014). *App Inventor 2: Create Your Own Android Apps*. O'Reilly Media.

T. Tóth is a recent graduate of doctoral studies (PhD.) in the field of Theory of Teaching Informatics. The results presented in this article are the part of his research carried out within the dissertation thesis. He currently works as Assistant Professor at Department of Informatics, Slovak University of Agriculture in Nitra, Slovak Republic. His scientific interest is focused on the software development, teaching programming and creating websites.

G. Lovászová is an Associate Professor of Informatics Education at Constantine the Philosopher University in Nitra, Slovakia. Her academic interest areas are methodology of teaching Informatics, mobile technology integration into education, and professional competencies of future Informatics teachers.