THE DESIGN, DEVELOPMENT AND EVALUATION OF A VISUAL PROGRAMMING TOOL FOR NOVICE PROGRAMMERS: PSYCHOLOGICAL AND PEDAGOGICAL EFFECTS OF INTRODUCTORY PROGRAMMING TOOLS ON PROGRAMMING KNOWLEDGE OF GREEK STUDENTS

IOANNIS VASILEIOU VASILOPOULOS

A thesis submitted in partial fulfilment of the requirements of Teesside University for the degree of Doctor of Philosophy

SCHOOL OF SOCIAL SCIENCES AND LAW
TEESSIDE UNIVERSITY
TEESSIDE, UNITED KINGDOM

May 2014

Declaration

While registered as a candidate for the degree of Doctor of Philosophy the author has not been registered for any other award with any other university or institution.

No part of the material in this thesis has been submitted for any degree or other qualification at any other institution by the author or, to the best of his knowledge and belief, by any other person. The thesis describes the author’s original work.

Abstract

This thesis reports a research project that aims to improve the teaching and learning of introductory programming from a pedagogical and psychological viewpoint. Towards this aim, seven principles for designing educational programming tools for novices were identified by reviewing literature regarding novices' difficulties and using a theoretical framework defined by the psychological theories of Constructivism and Cognitive Load Theory. This set of design principles was not only theoretically identified, but its pedagogical impact was also empirically tested. For this reason, Koios, a new programming tool, was designed and developed as a manifestation of the combined set of principles. Empirical studies were conducted by a way of a quasi-experimental design in two different Greek secondary-education institutions. The independent variable was compliance with the set of the seven principles. Students' level of programming skills (procedural knowledge) was the dependent variable, while the quality of their mental models in the domain of introductory programming (declarative knowledge) was the potential mediator. The effect of compliance with the set of principles on students' programming skills and mental-model quality was explored via Koios' evaluation. Declarative- and procedural-knowledge measurements, as well as a practical test, were used to collect data, which were analysed using ANOVA and hierarchical multiple regression. The major conclusions drawn from this study are: (a) compliance with the set of design principles does not affect the development of novices' procedural and declarative programming knowledge, (b) a programming tool that highly complies with this set facilitates novices in the application of their procedural programming knowledge during program creation and (c) programming tools, declarative and procedural knowledge are independent components in learning to program. However, it was also concluded that the two knowledge types and a programming tool that highly complies with the set contribute significantly to novices' programming performance. This study contributes to knowledge by theoretically identifying and empirically testing a set of design principles for educational programming software, and by producing and scientifically evaluating a programming tool as an embodiment of this set. Through this evaluation, the suggestion of Koios as a practically useful programming tool for novices seems to be well supported.

Acknowledgements

First and foremost, I would like to acknowledge the invaluable help of and express my gratitude to Professor Paul van Schaik, the Director of Studies of this research project. Paul maintained a high level of professionalism in our working relation and provided continuous support, constant guidance, useful ideas, and thoughtful comments. This was a unique learning experience and without his excellent supervision and expertise I would not have made it this far. I really hope that this work reflects the level of supervision provided by Paul. I would also like to thank my second supervisor Professor Mike Lockyer for his ideas and suggestions. However, this project would not have started without the consulting and guidance of Professor Georgios Antonopoulos, who continued to be a helpful advisor and supporter throughout this research project. Special thanks to Dr. Aggeliki Prayati for her support and important advice throughout this project. I would also especially like to thank Ioannis Zafeiropoulos, the visual artist who contributed to the redesign of the dialogue windows of Koios Version 1. A special thanks to Vasileios Kerchoulas for making the necessary arrangements regarding the first round of data collection. Naturally, I would like to attribute my thanks to a number of fellow teachers for facilitating the execution of empirical study and participating in the rating task, especially Konstantina Avgeri, Dimitra Therianou, Vasileios Theocharis, Alexis Prasoulis, Ioannis Toskas, Dimhtrios Rigas, Aggelos Verykios, Vera Gerasimatou, Konstantina Plessa and Alexis Lazanas. Last, but not least I would like to thank Vasili, Frida and Kosta for their support.

Thank you all.

Abbreviations

ANCOVA	Analysis of covariance
ANOVA	Analysis of variance
B	Unstandardised regression coefficient
CLT	Cognitive Load Theory
CS	Computer science
CSE	Computer science education
DV	Dependent variable
F	F-test
GUI	Graphical user interface
HCI	Human-computer interaction
HMR	Hierarchical multiple regression
IDE	Integrated development environment
IDEs	Integrated development environments
IT	Information technology
IV	Independent variable
M	Mediator
M	Mean
n	Number of cases
N	Total number of cases
OO	Object-oriented
p	Significance probability
PL	Programming language
PLs	Programming languages

PSA	Pathfinder scaling algorithm
R	Pearson's product moment correlation coefficient
R^2	Squared multiple correlation coefficient
SD	Standard deviation
SE	Standard error
SPSS	Statistical Package for the Social Sciences
t	t-test
VIF	Variance inflation factor
WYSIWYG	What you see is what you get
α	Probability of Type I error
β	Probability of Type II error

Table of Contents

List of Appendices

List of Tables

List of Figures

Chapter One

Introduction

The limits of my language mean the limits of my world.

(Ludwig Wittgenstein, Tractatus Logico-Philosophicus (1922)

Contents

1.1 Overview

The first chapter of this thesis aims to introduce the reader to the research project reported in this thesis. Hence, this chapter discusses the research background that this study was based on, the research hypotheses proposed and explored by this study and the research strategy I employed to test the validity of these hypotheses. Finally, a brief summary of this project's outcomes is presented. This chapter concludes with an outline of the thesis' structure.

1.2 Teaching and Learning Programming

The notion of programming in computer science (CS) involves the conception and implementation in a programming language (PL) of the steps required for guiding a computer system to perform a specific task or to produce a specific output or result. Programming is interwoven into the evolution of CS, because it provides the means not only to exploit new advances in hardware, but also to implement evolutionary ideas and research findings in software. Furthermore, the degree of computers' penetration in science, telecommunications and entertainment renders their use an inextricable part of contemporary everyday life. The full exploitation of the capabilities computers offer, as well as the customisation of these capabilities to individual needs, requires sufficient knowledge of the way computers operate. Programming assists people to familiarise themselves with technology, improves computer literacy and provides a more detailed view and clarification of the operations performed by computers. Furthermore, a computer in the hands of a programmer can be a powerful tool for producing creative and innovative applications, regardless of whether their intended use is personal, professional, scientific or artistic. It is, therefore, not surprising that programming instruction has been included in many, CS and non-CS tertiary courses as well as in secondary education.

However, teaching and learning programming is considered a difficult and challenging activity by both instructors and students. The causes of this difficulty as well as the ways to deal with these have been an active research area since 1970 (1981). A research approach employed within this context, is to study the teaching and learning of programming from a psychological point of view. This approach introduced the field of the psychology of programming (Hoc, Green, Samurçay & Gilmore, 1990; Soloway & Spohrer, 1989; Weinberg, 1971). Thus, research in this field aims at helping novices to cope with the potentially demanding and frustrating task of programming by exploring the way humans learn and revealing the underlying cognitive processes that are taking place.

Since the early 1970s, the following research topics emerged as the most dominant in the psychology of programming (Robins, Rountree & Rountree, 2003; Winslow, 1996).

- Common mistakes made by novices.
- Skills expert programmers have acquired.
- Characteristics of programming languages (PLs) and integrated development environments (IDEs) that are designed exclusively for teaching introductory programming.
- Design (or specification) of the proper pedagogy and curricula for CS courses in primary, secondary and tertiary education.
- Selection of the appropriate programming paradigm for teaching.

- Application of knowledge-acquisition theories in the context of programming instruction.
- Definition of ‘learning to program’ and the kinds of skill that must be developed through it.

This is not a rigid categorisation, but rather a grouping of research topics that are found in the research literature, as these topics are not mutually exclusive. Thus, a particular study may focus on a specific topic and still be related to one or more of the other topics. This is the case for the research project reported in this thesis. Although it focuses on the design and development of a new programming tool and its evaluation, it takes into consideration findings and theoretical principles from various studies and theories. More precisely, these studies and theories focused on the difficulties experienced by novices and on the cognitive processes involved in teaching and learning programming.

1.3 Programming Environments, Novices’ Difficulties and Psychology of Programming

The design and development of the programming tool used and evaluated in this study, was influenced by (a) the features of existing educational software, (b) the problems novices are facing and (c) theories that explore human learning. Furthermore, its evaluation was based on quantitative techniques of knowledge measurement, proposed in the research literature. Research on these topics aims to improve novices’ performance in an introductory programming course. Therefore, introducing elements of these topics in the design of the developed programming tool should be an advantage because it would better support its users in learning to program. The following sections outline previous work that constituted the basis for this study.

1.3.1 Programming Languages and Integrated Development Environments

The creation of programs involves the expression of an algorithmic solution in a series of well-defined steps, using programming concepts (Murnane, 1993). The main reason for developing an educational programming language (PL) and an integrated development environment (IDE) for teaching is to assist students in writing programs, namely to convert the algorithmic steps of a solution to PL's commands. The skill of creating programs is considered important in the context of teaching and learning programming. Thus, it is desirable for educational PLs and IDEs to facilitate its users, especially novices, in the development and application of this skill. This can be achieved by simplifying the process of expressing an algorithm via a PL's programming concepts and by supporting its users in this task with relevant information. A second goal is to promote a deeper understanding of how computers execute programs (Goldweber, Bergin, Lister & McNally, 2006). A third goal is to motivate novices to be engaged in learning to program. In order to accomplish these three goals, PLs and IDEs tend to incorporate a commonly accepted set of features. These features include (a) a simple and readable syntax, (b) a small and orthogonal group of instructions and (c) the prevention of errors (Georgatos, 2002; Kelleher & Pausch, 2005; Mannila & de Raadt, 2006; McIver & Conway, 1996).

A plethora of educational PLs and IDEs has been developed to assist the introductory teaching of programming. The following grouping proposed by Gross and Powers (2005b) summarises trends in educational programming software in five categories, based on its main design rationale.

- Microworlds: learning environments based on manipulating objects (for example turtles or robots) with simple commands.
- Visual programming environments (iconic and textual): IDEs integrating visual characteristics for code creation. Code can be in graphical or textual form.
- Flow model environments: tools that support program creation with the use of connections between programming objects, for example by creating flowcharts.
- Object workbench environments: software designed for object-oriented (OO) programming with visual support and manipulation.
- Algorithm realisation environment (kinaesthetic, multimedia, animation, graphics): programming tools offering multiple types of presentation (visual, aural) of the execution of algorithms.

The following three techniques are proposed in the literature for the evaluation of educational PLs and IDEs. First, using the anecdotal technique, an instructor presents his/her personal view on the efficiency of the software by assessing its use in a classroom. Second, with the use of the analytical technique, a set of criteria is determined and the evaluation of the software is based on whether the software meets these criteria. Third, the empirical technique is based on the analysis of observational quantitative and/or qualitative data. These data are collected using various measures, such as students' scores in homework and exams, usage statistics provided by the software and surveys (Gross & Powers, 2005a; Mannila & de Raadt, 2006; Powers et al., 2006).

1.3.2 Misconceptions in Novices

A different approach in supporting novices to learn programming successfully is by uncovering their frequent misconceptions in the programming domain. The study of the problems, which novices are confronted with in their introductory programming course, has been an active research area for at least 40 years (Youngs, 1974). The findings of these studies have identified the following factors as especially problematic for novices. A first finding is that the syntax and semantics of PLs are a major source of errors for non-experienced programmers (Gomes & Mendes, 2007). Errors, also known as bugs, occur in syntax when the individual grammatical elements of a programming statement do not follow a specific order imposed by a particular PL. Syntax errors are usually detected by the compiler or the interpreter of PLs. Executing programs with semantic errors could produce unintended or undesired output. These kinds of error can only be detected during programs' execution and occur when a part or a whole programming statement is assumed to perform a certain function, which in reality it does not. Although syntax errors are not a formidable hindrance to creating programs, semantic errors seem to be particularly troublesome for novices (Allwood, 1986). A second finding is that, even in the case where syntax and semantics of statements are known, novices lack the ability to produce a programming solution (Lahtinen, Ala-Mutka & Järvinen, 2005). In this case, students fail to use the acquired programming knowledge in order to form and express a solution with the constructs of a PL, even though they are able to solve the problem using non-programming concepts and by means of paper and pencil. This is also reported as a lack of programming strategy (Robins et al., 2003). A third finding is that novice programmers possess a superficial knowledge of programming constructs already taught (Winslow, 1996). However, Spohrer and Soloway (1989) argued that bugs are produced, to a larger extent, by the deficiencies in program planning than by the misconceptions of programming constructs. A fourth finding is that beginners rely on their experience from conversations with other humans to help them with their interaction with computers during programming. This experience is responsible for two problems. First, students try to understand programming statements and their operation by interpreting the words that form these statements in the context of their everyday use (Bonar & Soloway, 1983, 1989). This results in misinterpreting the actual operation of programming statements. Second, novices expect from a computer to respond to programming statements in an anthropocentric way. Namely, they assume that the computer is able to view all the commands of a program at once and comprehend what the user intends to accomplish with the specific program. Moreover, the computer is expected to implicitly infer any programming statements or data that the user neglected to provide. This is what Pea (1984) called a superbug. This problem is further exacerbated by a fifth finding, a fact that eludes naive programmers: a command is executed within the computational environment that has been created by the execution of previous commands. This could be attributed to the lack of familiarisation with the PL's notional machine (Robins et al., 2003). The notional machine of a specific PL is a simple abstract model that provides a framework, within which the execution of a program, written in this PL, can be traced and explained (du Boulay, O'Shea & Monk, 1999). This helpful tool can assist novice and experienced programmers in comprehending the way that a program runs and anticipating its output. Thus, insufficient knowledge of the notional machine could be a source of confusion, especially for novices. Finally, a sixth finding is that another problematic aspect of programming for novices is debugging, namely the detection and correction of programming errors (Allwood, 1986). Except for having poor error-detection skills, novices have to confront the cryptic error-messages communicated by the compilers or interpreters of PLs. Commercial PLs are often selected for teaching programming in introductory courses, due to their wide professional acceptance. However, this type of PLs is designed for experienced programmers and they offer a very technical wording of errors, which may be incomprehensible especially to novice programmers. Thus, students often find themselves not only being unable to correct their bugs but also deprived from any assistance by the PL in doing so.

1.3.3 Cognitive Psychology and Programming

Knowledge from the discipline of cognitive psychology has been used for improving the teaching and learning programming as well as dealing with novices' deficits. These aims can be achieved by employing learning theories and revealing the cognitive mechanisms engaged in programming. Two dominant learning theories, namely Constructivism and Cognitive Load Theory (CLT), have provided a theoretical framework for facilitating programming instruction.

Constructivism, introduced by Jean Piaget in 1967, argues that new knowledge is formed based on previous experiences and knowledge already acquired. New experiences and information combined with previous knowledge in a dynamic operation add to (process of assimilation) or modify (process of accommodation) existing knowledge structures. The basic building block of knowledge was defined by Piaget as a schema (Sjøberg, 2007). A schema is an internal hierarchical cognitive structure that represents a set of attributes of a concept from the real world. Two types of information are contained in a schema, declarative and procedural knowledge. Declarative knowledge internalises the theoretical nature of a concept, namely what the concept defines. In the case that this concept has a practical application, procedural knowledge describes the way this concept can be used. Schemata allow the processing of complex knowledge as a single unit of information (Chalmers, 2003; Werhane, Hartman, Moberg, Englehardt & Pritchard, 2009). Ben-Ari (2001) was one of the first to investigate the implications of Constructivism in CS education.

CLT, introduced by John Sweller (1988), describes the learning process based on three types of memory - sensory, working and long-term - and three types of cognitive load - extraneous, germane and intrinsic. Knowledge is elaborated by working memory and stored in the form of schemata in long-term memory. CLT supports learning by reducing the extraneous cognitive load – that is load imposed on working memory by inappropriately designed teaching material (see Chapter 2.2.2.1) – and by increasing germane cognitive load – that is load on working memory related to learning (see Chapter 2.2.2.1). A successful management of extraneous and germane cognitive load is achieved through instructional design (Caspersen & Bennedsen, 2007; Chipperfield, 2004). The benefits of applying CLT and instructional design in teaching programming have been explored to a large extent by Shaffer, Doube and Tuovinen (2003), van Merriënboer (1990a, 1990b, 1992), and White and Sivitanides (2003) .

A cognitive structure that is hypothesised to support learning of complex systems is mental models. They were introduced by Kenneth Craik in 1943 by the term of ‘small-scale models’. Philip Johnson-Laird revisited the term in 1983 and argued that human reasoning and understanding depends on mental models (Westbrook, 2006). A mental model is considered as an extension of schema, with additional information for the use and adaptation of the knowledge existing in the schema.

Mental models are used by humans as subjective organisational structures that simplify the learning of complex systems. These structures facilitate the understanding of a particular system's operation, the conceptualisation of system parts that are not directly visible or accessible to its user, the description of the underlying processes that take place within this system and the prediction of possible states of the system (Darabi, Nelson & Seel, 2009). Mental models have been used to measure knowledge acquisition of different systems' users. For example, Cooke and Rowe (1997) used mental models to measure knowledge acquisition in avionics troubleshooting, NASA mission control tasks and personal relationships. Similar studies have been conducted in the context of CS and programming. In particular, mental models have been used to study the relation between them, self-efficacy and programming skills (Ramalingam, LaBelle & Wiedenbeck, 2004), to evaluate the understanding of recursive procedures by students (Kurland & Pea, 1985), to assess programming and debugging skills of students (Cañas, Bajo & Gonzalvo, 1994), to investigate the evolution and differences of procedural and OO experts (Corritore & Wiedenbeck, 1999), to predict success of pupils in introductory programming course (Caspersen & Bennedsen, 2007), and to measure students' knowledge of programming concepts (Ma, Ferguson, Roper, Ross & Wood, 2009).

Despite the wealth of international research studies and findings, only a handful of researchers have published findings from CS educational research in Greece. Evidently, only a small number of Greek educational software environments has been developed for teaching introductory programming. The Greek PLs and IDEs documented in the international literature are *AnimPascal (Satratzemi, Dagdilelis & Evagelidis, 2001), LECGO (Kordaki, 2009, 2010), ObjectKarel (Satratzemi,* *Xinogalos & Dagdilelis, 2003; Xinogalos, Satratzemi & Dagdilelis, 2006), DAVE (Vrachnos & Jimoyiannis, 2008)*, *WIPE (Efopoulos, Dagdilelis, Evangelidis & Satratzemi, 2005; Efopoulos, Evangelidis & Dagdilelis, 2005)* and *X-Compiler (Dagdilelis, Evangelidis, Satratzemi, Efopoulos & Zagouras, 2003; Evangelidis, Dagdilelis, Satratzemi & Efopoulos, 2001)*.

Based on these findings, the motivating force behind this research project was to provide a new educational programming tool to facilitate secondary- and tertiary-education students in their effort to learn programming, with a particular interest in Greek novice programmers. The term 'Greek novice programmers' does not refer in any way to the ethnicity of the students, but rather to novice programmers that study in Greek educational institutes. Hence, this term is always used in this thesis with the latter meaning. The following section discusses the approach adopted to effectuate this objective.

1.4 Research Approach

The motive behind my engagement with this project is to improve the teaching and learning of programming from a psychological and pedagogical point of view. Thus, the three primary aims of this research project are (a) the identification of important design principles for educational programming software, (b) the design and implementation of a new programming tool based on these principles and (c) the empirical evaluation of the new programming tool in educational institutes and the assessment of its impact on novice programmers. The first aim was accomplished by identifying misconceptions in novices, using the theoretical framework defined by constructivism and CLT and by reviewing the international research literature. A number of interviews with CS teachers in Greek educational institutes took place in order to further inform decisions and conclusions regarding the problems that novices face and the desired characteristics of educational software for teaching introductory programming. As a result, the following set of seven design principles was identified: (a) match of users' natural language with the linguistic attributes of the PL and IDE, (b) syntax and semantics of instructions supported by the PL, (c) visualisation, (d) abstraction of commands provided by the PL, (e) small set of instructions, (f) provision of error messages and (g) a high level of interaction with the IDE. The following main research question is addressed in this project.

How important is the set of the seven principles as a pedagogical consideration for educational programming software engineering?

In order to answer this question, the effect of compliance of programming tools with these design principles on mental–model quality and programming skills of novices as well as the effect of the quality of novices' mental models on their programming skills was studied. More specifically, the following hypotheses were tested:

H1: novice students who use a programming environment with a high level of compliance with the set of the seven design principles develop a higher level of programming skills than novice students who use programming environments that partially comply with this set.

H2: novice students who use a programming environment with a high level of compliance with the set of the seven design principles construct richer mental models in the domain of programming than novice students who use programming environments that partially comply with this set.

H3: the effect of programming environment on programming skills in novice students is mediated by the quality of their mental models.

The testing of these hypotheses could take place through the accomplishment of the second and third aim of this project. Therefore, I designed and developed a new educational programming tool, Koios, which embodied the combined set of the seven principles and highly complies with it. In order to investigate the effect of compliance with the set of the design principles, Koios was evaluated in Greek high-schools. The evaluation of Koios and data collection was based on a quasi-experimental design. In this design, two other popular educational PLs and IDEs, Glossomatheia (Nikolaidhs, 2008) and MicroworldsPro (LCSI, 2008), with a lower level of compliance with the set of principles than Koios, were used and compared with Koios. At the time this study took place, Glossomatheia was a popular choice for teaching introductory and advanced programming in Greek high-schools, while MicroworldsPro was the official recommendation for introductory programming in high-schools by the Greek Ministry of Education, Life-long Learning and Religious Affairs, as of 2008.

More specifically and according to the quasi-experimental design, the level of compliance with the set of principles was the independent variable. The independent variable had three levels, varying in their degree of compliance, the Koios prototype (high level of compliance), Glossomatheia (partial compliance) and MicroworldsPro (partial compliance). The dependent variable of the experiment was the acquired level of programming skills of pupils. The quality of students' mental models was the mediator. In particular, it was investigated whether the level of compliance significantly affected students' programming skills and the quality of their mental models. Additionally, whether the effect on students' programming skills was mediated by their mental models was investigated as well. Each level of the independent variable/programming tool corresponded with one quasi-experimental group. I taught the introductory programming course for each quasi-experimental group for 12 weeks of lessons, according to the official CS curriculum of the Greek Ministry of Education (Pedagogical Institute, 2004). The following 13 programming constructs were taught over this period: *integer, arithmetic expression, string, output-statement, input-statement, variable, assignment-statement, iteration-statement, procedure, procedure call, input parameter, logical condition* and *conditional statement*. Each quasi-experimental group used the corresponding programming tool for lessons and exercises during the study. In order to reduce variability between groups as much as possible, the same theoretical notes, examples, homework, teaching plan, method and material as well as procedural- and declarative-knowledge tests were used for all quasi-experimental groups.

In order to assess the quality of participants' mental models (declarative knowledge), three tests took place. The first one was administered before the beginning of the experimental manipulation, the second one after five weeks of lessons and the third one at the end of the study. The pairwise comparison technique (Cooke, 1994) was used to measure the mental models of students. The Pathfinder scaling algorithm (Goldsmith & Johnson, 1990) was used to create a Pathfinder network (PFNET), a non-hierarchical network representation of a mental model for each participant. The PFNET of each participant was compared with a baseline network. The closeness of the two networks was a measure for the quality of participants' mental model.

The level of programming skills (procedural knowledge) that students acquired was measured via tests specifically designed for this study. The items of these tests required from participants to predict the output of programs, complete missing programming statements and modify existing programs. Students' procedural knowledge was measured three times. The first measurement took place after three weeks of lessons, the second one after five weeks of lessons and the third one at the end of the study. Students' score in each procedural-knowledge test represented students' level of procedural knowledge at the time of the measurement.

Based on this quasi-experimental design, a first round of collecting data and evaluating the Koios prototype took place within the school year 2009-2010. In order to provide further evidence to test the three hypotheses, a second round of evaluation and data collection took place within the school year 2010-2011. The second round of data collection followed the quasi-experimental design and research plan that were used in the first round, with the following exceptions. In the second round, an improved version of Koios was used. The new version was developed based on observations from the first data collection and comments made by the supervisory team. The procedural-knowledge tests of the second round were refined versions of the ones used in the first round. The refinements were intended to make the new versions of the tests more sensitive to differences in students' procedural knowledge and were based on statistical analysis of the procedural-knowledge scores of the first round. Finally, a practical test was added after the final declarative- and procedural-knowledge measurement. During the practical test, students were asked to create three programs using only the programming environment that their group had been assigned to.

The validity of the three research hypotheses was tested through the statistical analysis of the collected data of both rounds. For this purpose, ANOVA and hierarchical multiple regression analysis were employed. Moreover, the set of the following covariates was taken into consideration during analysis: *gender*, *participant's GPA without CS, participant's GPA from previous year, homework* *frequency, duration of lesson, (group) size, day of week, hour of day* and *gaps between lessons*.

This research project’s contribution to knowledge is three-fold: (a) it theoretically identified seven design principles for designing educational programming tools for novice programmers, (b) Koios, a new programming tool was designed and developed based on a set of research-based scientific principles and (c) the identified set of design principles was empirically validated and Koios was empirically evaluated through quasi-experimental research. In the following section a brief outline of the main findings of this research is presented.

1.5 Research Outcomes

This section outlines three major conclusions of this study. All findings are thoroughly presented in Chapter Five and discussed in Chapter Six.

A first conclusion, based on the results, is that none of the three hypotheses were supported. This means that, at the end of the quasi-experiments, the mental–model quality and programming skills of Koios users were not significantly different from those of Glossomatheia and MicroworldsPro users. Moreover, a mediation effect (MacKinnon, 2008) of the quality of students’ mental models on their programming skills could not be established. This suggests that programming tool, procedural and declarative knowledge are independent components in the process of learning programming.

However, a second conclusion highlights the importance of programming tool, procedural and declarative knowledge in learning to program. From the results of the practical test, it was concluded that a programming tool with a high compliance with this set of principles can facilitate the *application* of procedural programming knowledge by novices during program creation. Finally, the third conclusion was that both types of knowledge in combination with a programming tool that complies highly with the design principles set contribute to a successful performance in an introductory programming course.

1.6 Structure of Thesis

In this chapter an introduction to the research reported in this thesis was presented. Moreover, a brief description of the research approach I followed to complete this project was discussed. Finally, this study's contribution to knowledge, as well as its major findings, were reported.

Chapter Two reviews the international literature on research areas with respect to the context of this study. The set of the seven design principles are also discussed in this chapter. The gaps in knowledge that this thesis attempts to fill are outlined in this chapter as well.

The third chapter documents the design and development of the Koios prototype – the educational tool used in this research – and its compliance with the seven design principles. Both the design choices made during the development and the features of Koios are discussed in this chapter.

Chapter Four reports the method used in this project. More particularly, the *Design*, *Participants*, *Material* and *equipment*, *Procedure* and *Data analysis* of this experiment are presented. Data collection is also discussed in this chapter.

The fifth chapter presents the results from data analysis. Results for the procedural- and declarative-knowledge data of the first round are presented first. The results of the second round follow, using the same presentation order. Finally, the results of the practical test are presented. These results were used to test hypotheses *H1*, *H2* and *H3*.

Finally, Chapter Six discusses the findings of this study based on the results presented in the previous chapter in relation to the research literature. Furthermore, this chapter discusses limitations, sources of potential bias, conclusions and contribution to knowledge of the research that was undertaken, and outlines suggestions for future work.

Chapter Two

Literature Review

Contents

2.1 Overview

This chapter discusses the literature that informs the research that was conducted in this project. The first section presents two pedagogical theories, namely Constructivism and Cognitive Load Theory. The two theories provide the theoretical background for this study. The second section reviews the difficulties that novice programmers face during their introductory course. Based on the first two sections, a set of seven design principles is identified and discussed in the third section. The fourth section reviews international and Greek education software for novice programmers and focuses on its relation to the identified set of the seven principles. The fifth section is concerned with psychological methods for measuring programming knowledge. In particular, declarative and procedural programming knowledge, mental models and the Pathfinder scaling algorithm are discussed. This chapter concludes with a summary of important findings that are obtained from the literature review and a statement of how this project aims to contribute to knowledge.

2.2 Learning Theories

Three primary aims of this project are (a) the identification of design principles for educational programming software, (b) the design and implementation of a new programming tool as an embodiment of these principles and (c) the evaluation of the new tool in educational institutes and the assessment of its impact on novice programmers. Learning theories can define a theoretical framework for enabling the accomplishment of the first two aims and interpreting the findings of this study. From various theories that propose a model of human learning, two dominant theories, Constructivism and Cognitive Load Theory, were selected to inform the theoretical background of this research for the following reasons. Constructivism considers learning as an active process of knowledge construction. Thus, within an educational context and proper guidance, a learner can construct himself/herself his/her knowledge of a domain. In the domain of programming, the educational context and the proper guidance can, in part, be provided by specially designed educational software for programming instruction. Moreover, the design principles for educational programming software can be identified via principles of constructivism. Furthermore, LOGO (1980), one of the first and most influential educational programming languages, was inspired by constructivism. Cognitive Load Theory examines how people process information and proposes a set of guidelines for managing this information for successful learning. Therefore, integrating these guidelines in the design and implementation of the context and interface of educational programming tools could result to the improvement of the learning effectiveness of these tools. Moreover, both theories are quite popular in Computer Science Education (CSE) literature.

Of course, other learning theories could be considered. However, choosing a small set of theories could avoid possible conflicts between them and reduce unnecessary complexity of the theoretical framework for this research.

Therefore, this section discusses the theory of Constructivism and Cognitive Load Theory. First, the theory of Constructivism is presented, followed by Cognitive Load Theory. At the end of this section, the relation of the two theories to this project is explained.

2.2.1 Constructivism and Computer Science Education

2.2.1.1 Theory of constructivism.

Constructivism is an epistemological theory, a theory of knowledge and how can it be acquired by human beings, and was introduced by Jean Piaget (1896 -1980). According to Piaget, knowledge is not a part of an objective world and therefore the quest for an absolute truth is not feasible. Constructivism describes the acquisition of knowledge as an active and repetitive process of combining new experiences with existing ones. This means that each of us constructs new knowledge based on our personal experiences. Therefore, theoretically, when individuals with different experiences acquire knowledge of a concept, this knowledge is differently constructed by each individual (Ben-Ari, 2002; Sjøberg, 2007). However, in practice, the knowledge that is acquired by each individual could introduce some misconceptions due to previous experiences, but it is unlikely to vary substantially between individuals because it regards the same specific concept.

The basic building block of knowledge used in this process is a cognitive structure called schema. The term schema was introduced by Piaget, but it was earlier described in the experiments of Frederic Bartlett (1932). According to Bartlett, people perceive and understand the world through a network of unconscious mental structures. Later, the term was revisited from an educational viewpoint by Anderson (1977), which led to schema theory.

According to Piaget, a schema contains information not only about a concept, but also about the vocabulary, the actions and the experiences related to the specific concept. Piaget's theory describes two major processes that facilitate knowledge acquisition, namely assimilation and accommodation. *Assimilation* occurs when a person is confronted with new information and attempts to explain it using preexisting knowledge, concepts or ideas. The result of assimilation is the addition of new information into existing schemata. *Accommodation* is a process complementary to assimilation and takes place when a person modifies already existing schemata to conform to new information or experiences. Therefore, these two processes promote the construction and modification of one's knowledge according to one's experiences and perception of the world.

However, Piaget's work focused more on cognitive development of children rather than learning per se. With the term development a series of long-term changes in a person's knowledge, skills and beliefs is described. According to his theory, the processes involved in cognitive development are adaptation and equilibrium. *Adaptation* is considered to be the modification of behaviour to meet a set of circumstances and it takes place via the processes of assimilation and accommodation. *Equilibrium* is defined as the ability to bring balance between a child's accumulated knowledge and the external world. Particularly, this ability is responsible for allowing children to correct any incongruities between reality and their perception of reality and effectively continue with their development. Perhaps the most influential idea of Piaget's theory is that cognitive development occurs in stages. These are developmental stages that each child has to go through as it grows older. Each stage is considered as a series of thinking patterns and forms the basis on which the next stage will be constructed. The rate at which a child passes through these stages may be different due to personal characteristics, but each child must pass through all the stages with a specific order. These developmental stages as well as their predefined order are (a) the sensorimotor stage, (b) the pre-operational stage, (c) the concrete operational stage and (d) formal operational stage (Chalmers, 2003; Ma, 2007; Seifert & Sutton, 2009; Werhane et al., 2009).

This body of work was only a means to Piaget's end, which was to study the nature and acquisition of knowledge and how humans learn to think logically. Unsurprisingly, Piaget himself was not very interested in providing a method for effective teaching and learning. However, during the1970s academics and teachers began to pay attention to the ideas of Jean Piaget about personal development and the developmental stages. This was the beginning of constructivism as a learning theory. The aim of this field is to study the implications of applying Piaget's theories into science education. Constructivist theories became and still are very popular, because they focus on the active role of the person who acquires knowledge or, in an educational context, the active role of the 'learner'. The scientific branch of constructivist theory that studies the state and attitude of students during the learning process is called individual or cognitive constructivism. Sjøberg (2007) presents the following ideas as a commonly accepted theoretical basis of cognitive constructivism: (a) learners actively construct their own knowledge, (b) students bring their explanations about many phenomena to the learning process, (c) learners have their own preconceptions about the world; many of these are common between learners and usually supported by culture and society, (d) a number of these preconceptions are inconsistent with scientific facts and often hard to change, (e) knowledge in the human mind is represented in cognitive structures of concepts that can be modelled and partly described, (f) effective instruction needs to acknowledge and engage learners' preconceptions and (g) knowledge may be a personal construction through an active process, but this process is influenced by many factors 'outside' the learner, such as teachers, co-learners and instructional material. The study of other factors that influence knowledge construction, 'outside' the learner, is the origin of different 'types' of constructivism, such as social (often with reference to Lev Vygotsky), simple, radical, contextual, sociotransformative and sociocultural constructivism.

Criticism against constructivism is mostly concerned with its epistemology and its philosophy, but it appears to hardly affect its status and application as a learning theory (Ben-Ari, 2001). Because only the educational implications of constructivism, not its epistemological theory, are relevant to this work, further discussion of its critique seems to be out of the scope of this thesis

2.2.1.2 Constructivism in computer science education.

A number of research studies in the field of CSE have implicitly referred to constructivist ideas. However, one of the first explicit study of the implications of applying cognitive constructivism in CSE was conducted by Ben-Ari (1998, 2001). He studied the difficulties students faced while they were using a what-you-see-is-what-you-get (WYSIWYG) word processor. His major findings were that (a) CS novices lack a cognitive structure, to which they can refer, in order to construct effective knowledge from their interaction with a computer – he called this cognitive structure an effective model – and (b) the computer constitutes an accessible ontological reality. The latter finding means that students, who hold a model with misconceptions about a task performed by a computer, are faced with an immediate exposure of these misconceptions during the actual performance of the task by a computer. Furthermore, he concluded that novices cannot create a correct model of computers based on their intuition and experiences from the physical world, as they can do in other scientific disciplines. He also suggested that constructivist principles should be taken into consideration in the design of software and programming languages for educational purposes.

Since then, the study of applying constructivist ideas in CSE has become popular and a number of research studies have focused on this topic. The tutoring system InSTEP (Odekirk-Hash & Zachary, 2001) was designed to provide a constructivist learning experience for engineering students in an introductory programming class. Their findings showed that the users who had feedback from InSTEP required less help from teaching assistants in solving problems than the users who had no feedback from InSTEP. However, both groups required about the same time to complete the task and in a subsequent test, they did not perform significantly differently. Influenced by Ben-Ari's conclusion regarding students lacking an effective model of computers, Powers (2004) decided to teach computer architecture as the introductory computing course, but he did not report any results about its effectiveness. Gonzalez (2004) adopted a constructivist approach in creating teaching material and activities for introductory programming classes with different learning styles. His findings suggested that constructivist activities facilitated students' understanding of the programming material. However, students with programming experience did not find the activities challenging. Lui, Kwan, Poon and Cheung (2004) developed a set of guidelines influenced by constructivism in order to help weak students succeed in an introductory programming course using C. Their results showed that students improved their performance and confidence in programming, they were able to perform better in homework and exams, and the fail rate of the course was reduced. A pedagogical strategy based on a constructivist approach was presented and implemented by Hadjerrouit (2005) for teaching object-oriented (OO) programming. He reported that students improved their problem solving skills and understanding of software engineering concepts. Based on constructivism, Vagianou (2006) presented a conceptual framework, which is called program working storage, for facilitating the understanding of complex programming concepts and challenging novices' preconceptions. Her findings were that, at the end of the course, the users of this framework were able to provide theoretical explanations of programming concepts, apply these concepts practically in exercises and create small programming projects. A set of measureable learning milestones within a curriculum context for improving instruction of programming was proposed by Mead et al. (2006). Constructivism was one of the theoretical underpinnings of their work. Beynon (2009), based on Ben-Ari's work, presented three case studies on how empirical models can be used from a constructivist viewpoint to teach the bubblesort algorithm, solve Sudoku puzzles and recognize groups from their abstract multiplication tables. Milner (2010) referred to constructivism to explain why the concept of 'static' in Java is often difficult to grasp. A visual programming environment was developed by Lee (2011) in order to help teachers programmatically produce their teaching material from a constructivist point of view. Finally, Teague and Lister (2014) administered a reversibility programming task in order to determine how well could novice programmers produce code that would cancel out the result of a program's execution. *Reversibility* is an ability supposedly developed by learners at the *concrete operational* stage, the third one of the developmental stages proposed by Piaget. They interpreted their findings using think aloud data based on neo-Piagetian stages and discussed the application of neo-Piagetian theory on novice programmers.

Research on the implications of applying constructivism in CSE seems to focus on providing pedagogical guidelines on how to effectively teach CS-related subjects. However, as Sjøberg noted, "a set of principles for learning do not directly translate into a set of recommendations for good teaching. One cannot logically deduce a scientifically based pedagogy from a theory of learning" (Sjøberg, 2007, p. 9). Unsurprisingly, the findings of Odekirk-Hash and Zachary (2001) suggest that learning can be facilitated by incorporating constructivist ideas in the design of software, which is used for introductory programming. This topic, however, seems to have drawn very little attention in research. Because of the small number of studies in this topic, further investigation is required for the potential benefits of applying constructivism in the design of educational software for introductory programming.

2.2.2 Cognitive Load Theory and Computer Science Education

2.2.2.1 Basic concepts of cognitive load theory.

Cognitive constructivism proposes a framework in which, one interacts with new experiences and information as well as with already acquired knowledge in order to expand one's knowledge. However, the hypothesised learning processes that take place in the human mind during knowledge acquisition are described by Cognitive Load Theory (CLT). CLT was introduced by John Sweller (1988) and is a theory that puts forward a model of how new information is processed by the human brain and a method for achieving effective learning.

Human memory can be divided in three parts: (a) sensory memory, (b) working memory and (c) long-term memory (Raaijmakers, 1993). Sensory memory receives all sensory stimuli from the outside world and forwards them into working memory. A stimulus can be retained in sensory memory for only a small fragment of time and if it is not forwarded to working memory is considered lost.

Working – or short-term – memory is hypothesised to be the place where all conscious processing of information occurs. Its main characteristic is its hypothesised limited capacity. According to Miller (1956) working memory can hold seven plus or minus two items at a time, although these figures can be tripled through proper training (Caspersen & Bennedsen, 2007). However, when items must be processed rather than just be held in working memory, extra limitations are imposed. This happens, because the processing of items requires working-memory capacity. This allows the processing of only four items plus or minus one simultaneously (Sweller, van Merriënboer & Paas, 1998). Recently, Paas and Sweller (2012) amended the application of working-memory limitations. They argued that these limitations are crucial, only when novel biologically secondary information is processed (originating from social or cultural environments, for example reading or solving problems).

All the processed information is stored in long-term memory in the form of schemata. The term schema is used in CLT with respect to schema theory, the work of Anderson (1977), who expanded Piaget's concept of schema. According to Sweller, van Merriënboer and Paas, "a schema categorizes elements of information according to the manner in which they will be used" (Sweller et al., 1998, p. 255) , but it can also consist of problem-solving rules (Sweller et al., 1998, p. 257). The capacity of long-term memory is believed to be virtually limitless and the management of information that is held there takes place unconsciously. When relevant schemata are required for recalling or processing, they can be retrieved from long-term memory and placed in working memory.

Schemata not only serve as cognitive structures for storing knowledge, but also as organisational structures that enable the processing of a large amount of information. During instruction or learning, schemata continue to change and incorporate more information. New information, in combination with the learner's prior knowledge, reorganises and rearranges the learner's schemata so it can be stored in them and thus be 'learned'. This is a conscious and active process and its practice over a long period of learning results to more complex schemata. Through this process, low-level schemata are combined to create high-level schemata that contain larger quantities of more complex information. The construction of even more complex schemata makes feasible the processing of large amounts of information, because each schema is processed in working memory as one item. Working memory is able to process a schema as a single item, even if it is too complex or it contains much information. Therefore, the more sophisticated a schema is, the more information in working memory can be processed, without imposing extra load on working memory's capacity. Thus, a first way to overcome the limitations of working memory is by *complex schema construction* (Ayres & van Gog, 2009; Caspersen & Bennedsen, 2007; Sweller, 1988; Sweller et al., 1998).

A second way to deal with working memory's limitations is by *schema automation*. Schema automation describes the unconscious and effortless recall and processing of schemata in working memory. In order to execute a task or solve a problem that is not yet mastered, the conscious processing of relevant schemata must take place in working memory. As one's abilities improve in performing the specific task or problem solving, less and less conscious effort is required for its performance or solving similar problems. When one becomes an expert in the particular task or problem-solving area, then one performs the task or solves the problem automatically, namely without any conscious effort or use of working memory. Thus, when an expert performs a task or solves a problem from the domain of his/her expertise, he/she can perform it effortlessly and correctly with a reduction in the use of working-memory resources. Because of this reduction, working-memory resources are available for effectively addressing more complex problems or tasks (Ayres & van Gog, 2009; Sweller, 1988; Sweller et al., 1998).

Obviously, a high level of schema construction and schema automation are identifying qualities of an expert in any specific domain. Thus, the aim of instruction should be enabling novices develop such a high level of schema construction and automation. The main concern of CLT is to provide the appropriate framework in order to accomplish that aim. Given the presumable limitations of working memory (Raaijmakers, 1993), the learning content and the instructional environment, CLT distinguishes three types of cognitive load: (a) intrinsic, (b) extraneous and (c) germane cognitive load.

Intrinsic cognitive load is the load imposed on working memory by the nature of the subject taught. This type of cognitive load is considered to be manageable, but not amenable to change. The main source of intrinsic cognitive load is the element interactivity of the subject. The level of element interactivity is determined by the number of the elements of a task or subject that can be successfully learned without needing to refer to their relation with other elements. In order to process new information correctly, a number of items – in most cases schemata – that interact with the new information must be present in working memory. Especially in complex domains, the number of the elements that a learner must hold simultaneously in working memory is high. This allows little or no space at all for the processing of new information in working memory, which in turn results to poor schema construction. However, if the subject has low element interactivity, there are no serious obstacles in learning, because only a few elements need simultaneous processing. This way, the necessary space in working memory for effective manipulation of new information is provided. The level of element interactivity differs between people according to a person's expertise. For example, solving linear equations can be a subject with low element interactivity for a math professor, but a subject with high element interactivity for a secondary-school student, who is just beginning to learn how to solve them. This can be explained by the schemata about linear equations that these two persons have. Obviously, the math professor's schemata are far more complex and sophisticated than the schemata of the secondary-school student. This difference in schema complexity, allows the math professor to treat a linear equation as a single element of information. However, the student, who is just a novice in this kind of equations, has available only low-level schemata. In order for the student to solve the equation, all the relevant low-level schemata need to be processed simultaneously in working memory. Thus, the same problem presents different degrees of element interactivity for different individuals (Chandler & Sweller, 1991; Sweller, 1994; Sweller et al., 1998).

The cognitive load produced by the instructional material is called *extraneous cognitive load*. This type of load is caused mostly by poorly designed material and it is additive to intrinsic cognitive load. The poor design of material hinders schema construction, instead of fostering it. When new information is presented, a certain amount of mental effort is required for its comprehension by learners. If the presentation of new material is properly designed, then the extraneous cognitive load is low. Nonetheless, if the new information is not well presented, learners must put in extra effort to extract and collect the necessary information from the poorly designed presentation. This additional effort increases extraneous cognitive load and occupies space in working memory, which otherwise could be used for processing the new information. However, this observation is valid only when the intrinsic cognitive load is high. In the case of low intrinsic cognitive load, extraneous cognitive load seems not to affect the learning process. This occurs because a low intrinsic cognitive load leaves enough space in working memory for the management of extraneous cognitive load. Recently, Sweller (2010) attributed extraneous cognitive load to element interactivity of instructional material and methods. This formulation does not change the origin and properties of extraneous load. Obviously, instructional material and methods can be altered in a way that their unnecessarily high element interactivity can be lowered and, in turn, extraneous load can be reduced. However, this formulation seems to propose element interactivity as a common basis for explaining both intrinsic and extraneous cognitive load. According to CLT, in order to improve learning, extraneous cognitive load must be reduced. This reduction is achieved through instructional design. Instructional design offers educational guidelines that enable the presentation of new material in a way that minimise the extraneous cognitive load (Chandler & Sweller, 1991; Sweller, 1994; Sweller et al., 1998).

A third kind of cognitive load, which was added in CLT later, is *germane cognitive load*. A high cognitive load was observed, when learners were presented with instructional material with high variability. However, this load was related to neither intrinsic nor extraneous cognitive load. This type of load was considered beneficial because it facilitated schema construction and transfer of learning. Hence, germane cognitive load was introduced to explain the load on working memory that is useful for schema construction and automation. Recently, Sweller (2010) redefined the term, suggesting that germane cognitive load should be related to working-memory resources that are allocated to process intrinsic load, rather than with a certain type of cognitive load. Since the introduction of the concept of germane cognitive load, instructional design has focused not only on managing intrinsic load and reducing extraneous cognitive load, but also on increasing germane cognitive load. However, because germane cognitive load is additive to the other two types of load, it is a crucial condition for instructional design to avoid cognitive overload. Cognitive overload can occur when the working-memory limits are exceeded by the sum of the various types of cognitive load. This way, the resources of working memory are allocated for processes that foster learning and schema construction, while processes that prevent these functions are controlled by minimising their harmful effects (Paas & van Merriënboer, 1994; Sweller et al., 1998; van Merriënboer & Sweller, 2005).

2.2.2.2 Principles of instructional design based on cognitive load theory.

Over the decades of research in the application of CLT in various disciplines, researchers have demonstrated seven techniques that reduce extraneous cognitive load. These techniques are: (a) the *goal-free effect*, (b) the *worked-example effect*, (c) the *completion problem effect*, (d) the *split-attention effect*, (e) the *modality effect*, (f) the *redundancy effect* and (g) the *expertise reversal effect* (Chong, 2005; Sweller et al., 1998; van Merriënboer & Sweller, 2005; van Mierlo, Jarodzka, Kirschner & Kirschner, 2011).

The first technique is called the *goal-free effect*. During problem-solving instruction, a common practice is that an initial and a final (goal) state of a problem are given and learners have to produce a solution, moving from the initial to the final state of the problem. Many students, lacking a schema for solving the particular problem (problem-solving strategy), do not work forward to solve the problem, but they resort to means-end analysis. The application of this method involves moving (backwards) from the final to the initial state of the problem by passing through the minimum possible number of different states between the final and initial state and applying available problem-solving operators (Sweller, 1988). This strategy poses a significant extraneous load on working memory. This happens because learners have to bear in mind the initial state, the final state, the possible states of the problem and the available problem-solving operators as well as valid ways to apply the operators and move between states of the problem. It also offers little support for schema construction. The *goal-free effect* suggests that a problem should be presented without a final state, so that learners would engage the problem creatively, without having a final state of a problem as a goal. In this case, a learner needs to remember only the current state of the problem and the operators that can be applied to it. This combination is considered to help schema construction. However, this technique can be applied only when the problem space has a few alternatives available.

The second technique is the *worked-example effect*. By studying worked examples learners seem to be able to focus on the steps that are required to solve a problem and to abstract a generalised strategy for solving similar problems. A worked example is a problem that is presented together with its solution. The solution has the form of the detailed steps that an expert would take to solve it. The study of these examples is supposed to be more beneficial than just solving the actual problems. However, worked examples can be beneficial only if they are studied fully and carefully by learners. Furthermore, an overuse of worked examples without any novel problem-solving tasks may be not effective, because learners may be content to follow a stereotyped method to solve problems, instead of coming up with new and innovative solutions (Paas & van Merriënboer, 1994; van Merriënboer, Schuurman, de Croock & Paas, 2002).

The third technique, the *completion problem effect*, is complementary to the worked-example effect. In this case, a worked example is presented, but not in its entirety. Some steps of the solution are missing and learners are asked to complete them. Completion problems are an intermediate stage between worked examples, which offer a full solution, and conventional problems, which offer no solution. This technique engages learners in the solution of a problem more actively than merely studying a worked example, but with the necessary guidance, which is not provided in a conventional problem. In this case, similarly to the worked-examples effect, extraneous cognitive load is kept low and schema construction is facilitated. Furthermore, learners focus on and actively participate in the creation of the solution. Nevertheless, the construction of a completion problem requires delicate consideration, because careful decisions must be made on which parts of the problem should be presented and which parts should be omitted (van Merriënboer, 1990b, 1992; van Merriënboer, Schuurman, et al., 2002).

The *split-attention effect* is a fourth technique demonstrated by CLT researchers. In many disciplines, such as math and physics, the presentation of a problem consists of a textual part and a graphical part (pictures, diagrams). In most cases, the text contains data or information that refer to the associated diagram or picture. The correct understanding of the problem requires that the data or information are analysed with reference to the associated diagram or picture. To accomplish that, one must retain in working memory both the graphical and the textual part and the references between them. However, this causes an increase in the extraneous cognitive load and only a small portion of working memory can be allocated to the solution of the problem. An integrated presentation of the graphical and textual parts can reduce the extraneous cognitive load. In this case, the learner needs not to split his/her attention between two or more sources of information and mentally combine them. Instead, he or she may focus on the integrated source and collect the required information.

A fifth technique is the *modality effect*. Baddeley (1992) proposed a model of working memory, that consisted of a system, the central executive, that classifies the incoming information in working memory, shifts between tasks and retrieval strategies, manages attention and co-ordinates two slave systems: (a) the phonological loop and (b) the visuo-spatial sketchpad. The phonological loop is responsible for processing auditory information, while the visuo-spatial sketchpad processes visual and spatial information. Later, Baddeley (2000) added a third slave system in the model, the episodic buffer, which is responsible for binding visual, spatial and auditory information chronologically. This model suggests that the processing of visual and auditory information uses different resources of working memory and hence, this processing can take place in parallel. Therefore, audio and visual data can be processed simultaneously, and thus expanding the capacity of working memory. Based on this principle, instructional material can be designed in such way that combines audio and visual information. For example, a problem can be presented with explanatory spoken text. Obviously, this occurs only when the visual and audio information are combined in a way that support each other. This means that neither the visual nor the audio information should be complex enough to require extra cognitive resources for processing it. In the case that a badly designed graphical component or a long auditory text is used, there may be an increase in the extraneous cognitive load, instead of a decrease (Leahy & Sweller, 2011).

A sixth technique is the *redundancy effect*. This effect occurs, when a number of different self-contained sources present the same information. A self-contained source of information provides information that can be understood on its own, without any reference to other material. When two or more self-contained sources of information present the same material in different forms, cognitive resources are required for the integration of the different forms and their processing. Apparently, the specific material could be understood by occupying less working-memory space, if it was presented only once. Thus, the redundant presentation of the same material causes an increase of extraneous cognitive load. The redundancy effect can be dealt with by removing multiple self-contained sources of information. This is so, because "integrated information is very hard to ignore" (Sweller et al., 1998, p. 284).
The seventh and final technique is the *expertise reversal effect* (van Merriënboer & Sweller, 2005). The six effects that were discussed before, can improve the instruction of novices. However, as novices gain more experience, the instructional techniques that were helpful for novices, seem to have no effect or even to hamper learning. This observation denotes a relationship between cognitive load effects and level of expertise. Therefore, a critical factor for designing an instructional method is the consideration of the level of expertise and prior knowledge of learners. For example, in the case of the completion problem effect, no expertise reversal effect is detected, as long as the presented information in the completion problems is gradually reduced (fading worked examples) as learners gain expertise.
The discussion of managing intrinsic cognitive load and increasing germane cognitive load is less extensive than of the reduction of extraneous cognitive load. As mentioned previously, the use of completion programs has also been suggested for the increase of germane cognitive load. A second method to increase germane

cognitive load is to present examples with high variability. The presentation of multiple examples seems to enable learners to identify similar features between problems that are relevant to their solutions.

A method proposed to manage intrinsic cognitive load is to follow a simple-to-complex or a low-to-high fidelity strategy. The main aim of these strategies is to produce a learning environment or a presenting sequence of the material in a way that the element interactivity is reduced. It is important to note that the total number of the elements of a task or subject and their interrelations is not reduced, because this would result to poor learning outcomes. The information of the learning material is distributed between lessons in a manner that enables a learner to process each lesson's information without having to retain a significant number of elements and interrelations in working memory (Paas, Renkl & Sweller, 2003; Paas & van Merriënboer, 1994; van Merriënboer & Sweller, 2005; van Mierlo et al., 2011).

2.2.2.3 Criticism of cognitive load theory.

Despite its wide acceptance (Ayres & van Gog, 2009; Paas, Van Gog & Sweller, 2010; Verhoeven, Schnotz & Paas, 2009), CLT has been recently subjected to some criticism. Moreno (2006) questioned the effectiveness of worked examples, because a number of studies yielded results that contradicted CLT' s predictions and reported some methodological issues in CLT research. Schnotz and Kürschner (2007) revisited the three types of cognitive load, the requirements for learning and instructional design principles. They suggested that intrinsic load can be fixed for specific learning conditions, but instructional design should adjust it and occasionally increase it according to the desired learning outcome and learners' expertise. They also identified other sources of extraneous load in addition to element interactivity, such as interactivity and retention of relevant information, interactivity of irrelevant information and unnecessary allocation of time and resources in easy tasks. Germane cognitive load was associated only with cognitive processes that are involved in learning activities which are more cognitively demanding than the performance of simple tasks. Furthermore, they argued that germane load can be constrained, not only by working-memory limitations, but also by intrinsic load and motivation. They also proposed that learning does not necessarily use working-memory resources and that schema construction and automation can take place without the involvement of germane load. In addition to unnecessary high element interactivity, unnecessary mental effort and low intrinsic load were considered as factors for hindering learning. They also recommended the modification of intrinsic load according to a learner's level of expertise and the adjustment of germane load according to the intrinsic load of the learning task. De Jong (2010) raised a number of issues regarding the conceptual accuracy, research methods and external validity of CLT. He argued that the three types of cognitive load cannot be clearly distinguished from each other because (a) they are used to describe different entities (for instance, intrinsic load stems from "objects", while germane load stems from "processes") and (b) the type of load that is produced by a cognitive process can be interpreted according to a learner's expertise and learning outcome (for example, the load of a process can be germane for a novice, but extraneous for an expert). Moreover, based on these arguments, he questioned the additive nature of the three loads. De Jong also scrutinised the measurement of cognitive load in relevant studies. His major criticism was the lack of a standard measure that can directly and reliably measure cognitive load and cognitive overload. Ideally, such a measure should also be able to identify each of the three types of cognitive load and not allow any speculative interpretations of the measurement results. Furthermore, he discussed methodological issues of research, such as participants who are included in the study, but not interested in learning the domain, duration and different learning conditions between CLT studies and warned about the potential implications of cognitive overload during an empirical study. The problematic nature of these issues was noted by Moreno (2010) as well, who also questioned the ability of CLT to predict the effects of instructional design. Finally, Kalyuga (2011) argued that the concept of germane cognitive load, as defined in the basic CLT, is not required for describing mental activities involved in effective learning and understanding; these activities could be described by intrinsic load. However, Kalyuga (2011) seemed to acknowledge the redefinition of germane cognitive load as proposed by Sweller (2010) (see Section 2.2.2.1).

2.2.2.4 Cognitive load theory in computer science education.

The implications of applying CLT as well as of its effects in CSE have been investigated by a number of studies. One of the major contributors to CLT and to the study of its application in computer science and programming is Jeroen van Merriënboer. He investigated instructional approaches and tactics for introductory programming in high school (van Merriënboer, 1990a; van Merriënboer & Krammer, 1987). He compared the program completion with the program generation strategy and found the former to be more effective than the latter with regard to program construction (van Merriënboer, 1990b, 1992). He and Paas (1990) studied schema acquisition and automation and how they can be facilitated in the context of elementary programming. Moreover, van Merriënboer and Kirschner (2007) introduced the 4C/ID model, a method for designing instructional programmes for complex skills acquisition (van Merriënboer, Clark & de Croock, 2002; van Merriënboer, Kirschner & Kester, 2003). Mayer and Moreno (2002, 2003) investigated how CLT can aid multimedia learning and the design of such tools, and proposed a cognitive theory of multimedia learning. Their work produced a set of guidelines for multimedia design; for instance, an explanation is more effectively presented by images and words than by words alone, and words and pictures should be presented simultaneously. The theoretical basis of their work was the following research fields: (a) dual coding theory – visual and auditory information are processed separately, (b) CLT and (c) constructivism. Garner (2002a, 2002b) used the completion problem effect to produce CORT, an interface tool that allowed students to complete missing lines of code in programs. A statistical analysis of exam marks, average time spent on problems and average estimation of help required by students revealed that the exam marks between users and non-users of CORT did not differ significantly. However, users of CORT required significantly less help and time to complete the programs than non-users of CORT (Garner, 2009). Tuovinen (2000) proposed a way of applying CLT effects in computer education, with an emphasis on the significance of learners' existing computer knowledge. Schafer et al. (2003) proposed an application of CLT in CSE, based on the demonstrated effects of CLT. Abdul-Rahman and du Boulay (2014) investigated the effect of different worked-examples strategies on the quality of programming-knowledge schemata regarding loops acquired by learners using cognitive load measurements and taking into account learning style. No significant differences between groups were overall detected and the subjectivity of cognitive load measurements was acknowledged as a possible explanation for the inconsistency of their results.

Caspersen and Bennedsen (2007) created an instructional design based on CLT and other theories for teaching OO programming. The instructional design was used for four years, but it was not formally evaluated. CLT and constructivism were two of the theoretical foundations for the cognitive-based curricular approach proposed by Mead et al. (2006) for programming. Gray, Clair, James and Mead (2007) developed fading worked examples for specific programming concepts. An integration of CLT and HCI principles was presented by Hollender, Hofmann, Deneke and Schmitz (2010) in the context of educational software. Their review showed that there is common ground between CLT and HCI, namely the reduction of unnecessary load in users' mind. They also proposed two (unnamed) models, which integrate basic ideas of the two theories, for research purposes. The first model explores the extraneous cognitive load produced by the mere use of the software, while the second integrates the three types of cognitive load into the concept of usability. Furthermore, they noted that usability principles alone are not sufficient for successful learning.

It seems that CLT, despite its criticism, has inspired a number of researchers to study its application in teaching various subjects that belong in CSE including programming. Nevertheless, the way that CLT principles could influence the design of educational tools for novice programmers is a topic that has been rather neglected by researchers, with the exception the work of Garner (2002a, 2002b, 2009). Although I agree with some points of critique, like the subjectiveness of cognitive load measures or the unclear borders between the various types of cognitive load, this project is concerned with neither introducing a teaching strategy for programming based on CLT nor measuring cognitive load. Therefore, CLT concepts that are still controversial can be overlooked. This study is, however, concerned with identifying a set of design principles for educational software that aims to support learning of programming and, designing and implementing a programming tool as a manifestation of this set. In this context, CLT can be useful in informing these design principles as well as providing guidelines for the design of the educational tool. Therefore, for this thesis, CLT provides a theoretical background for reducing harmful cognitive load rather than a pedagogical framework for the instruction of programming.

2.2.3 Relation of Constructivism and Cognitive Load Theory to This Project

This section reviewed two theories of cognitive psychology that are quite popular in CSE. The majority of the studies focus on their application in providing a theoretical framework for teaching and learning CS subjects, including programming. These two theories have provided a combined conceptual basis for a very small number of studies that were discussed in the previous section (Mayer & Moreno, 2002, 2003; Mead et al., 2006). However, this is not the case for the design of an educational tool that facilitates learners during their introductory programming course.

This project considers constructivism and CLT as its theoretical underpinnings, not only for identifying design principles that are supposed to improve the teaching and learning of programming, but also for designing an educational programming tool, which manifests these design principles. It is important to note that the literature review revealed that the two theories have apparently never been combined in the design and development of educational software for programming. This combination seems to provide a promising and intriguing theoretical foundation as well as a challenging framework for the design and development of the educational programming tool.

2.3 Learning Objectives of Teaching Programming

In order to correctly identify the obstacles that novices face during their introductory programming course, it is important to define the educational objectives of computer programming instruction. Therefore, possible objectives of 'learning to program' are discussed and the stance of this thesis on this matter follow.

Despite the numerous studies in computer science education research, the fundamental question of 'what should the learning objectives of programming instruction be ?' has not yet received a widely accepted answer (Blackwell, 2002b). The answers that have been proposed until now include various topics, which range from mathematical theory (Dijkstra, 1989; Pair, 1990) to learning taxonomies.

Initially, programming was related to mathematical theorems. For example, Hoare (1969), who developed the Quicksort sorting algorithm, suggested that programmers could establish the properties of their programs through the application of mathematical axioms and theorems. This approach, however, is nowadays scarcely adopted, probably, due its prerequisite for a rigorous mathematical background.

A more traditional answer has to do with learning the syntax, semantics and features of the PL used in the introductory programming course (McGill & Volet, 1997). Knowing the syntax and semantics of a PL is an important aspect of programming. This kind of knowledge is required for someone to be able to create programs in a particular PL. Nevertheless, from an educational viewpoint, the mere knowledge of these surface attributes does not guarantee that someone has developed any deep understanding of the programming domain and real programming skills. Consequently, this type of knowledge acquisition could be better described as programming language instruction rather than programming instruction.

Furthermore, Pea and Kurland (1987) supported that idea that learning to program has little to do with learning the syntax and features of a PL.

One of the widely adopted answers to the previous fundamental question relates programming-knowledge acquisition with the development of problem-solving skills (Black, 2006; Caspersen & Bennedsen, 2007; Dagdilelis, Satratzemi & Evangelidis, 2004; Deek & Espinosa, 2005; Koppelman, 2008; Porter & Calder, 2003; Sheard, Simon, Hamilton & Lönnberg, 2009; Tu & Johnson, 1990; Wiedenbeck, 1985). A typical programming course includes, to a larger or smaller degree, solving various (scientific or every-day) problems through the creation of suitable programs. In this context, a novice programmer should not only know a set of programming constructs, but he/she should also be able to use these constructs in order to solve problems programmatically. More specifically, in computer-programming, problem-solving involves (a) identifying the necessary steps to deal successfully with a particular problem, (b) convert these steps into an algorithm and (c) correctly translate the algorithm into a program, using the commands and formal syntax provided by a PL. Linn (1985) argued that this level of competence is reached through a series of cognitive accomplishments. This series has three stages: (a) learning the language features, (b) learning how to design programs for solving problems and (c) developing problem-solving skills, which are transferable in other programming languages or similar domains, for example math or physics. However, it is suggested that “problem solving is necessary, but not sufficient for programming” (Robins et al., 2003, p. 160).

A similar suggestion with focus on Linn’s second stage of cognitive accomplishments was proposed by Elliot Soloway (1986). He has greatly contributed in this field and together with James Spohrer presented in the book ‘Studying the Novice

Programmer' (1989), one of the early and thorough collections of studies in novices' difficulties. Soloway suggested that learning to program means learning to create mechanisms and explanations. According to him, a mechanism is a series of actions that produce a desired result and an explanation is a justification of this mechanism's design. In the programming domain, a mechanism is equivalent to a program and an explanation to the knowledge of how and why a program must be created in a certain way. He emphasised that goals and plans in program creation are crucial components and should be explicitly presented and taught. He also supported the idea that this method will enable novices to know how and why a specific piece of code should be produced.

Du Boulay (1989) introduced the term of the notional machine, which is a mental representation of the computing mechanism defined by the constructs of a specific PL. The main purpose of a notional machine is to facilitate the understanding of the interaction between a program in a specific PL, the computer and the user. This mechanism allows programmers to know which constructs of the specific PL they should use, understand how a program runs, predict its output, modify it, debug it and gain awareness of hidden states of the program-computer-user interaction. Therefore, the understanding a PL's notional machine is of high importance and could be considered as a main learning objective of programming instruction, although it was never explicitly formulated as such.

Finally, another method for defining the learning outcomes of programming instruction is the use of learning taxonomies. A learning taxonomy is a set of learning stages ordered in a specific way; it is used for developing the learning objectives of a course, the appropriate material and assessment methods. Learning taxonomies are generic systems that distinguish the learning objectives in the following three domains: (a) cognitive, (b) affective and (c) psychomotor and can be applied in different disciplines (Fuller et al., 2007). Accordingly, taxonomies have been used in programming instruction for the design of courses, the design of educational materials, and the assessment and analysis of students' attitude towards exercises. The only taxonomies that have been applied in programming instruction are Bloom's taxonomy (Fitzgerald, Simon & Thomas, 2005; Fuller et al., 2007), a revised version of Bloom's taxonomy (Fuller et al., 2007) and the Structure of the Observed Learning Outcome (SOLO) taxonomy (Fuller et al., 2007; Lister, Simon, Thompson, Whalley & Prasad, 2006). Bloom's taxonomy consists of six levels of performance: (a) knowledge, (b) comprehension, (c) application, (d) analysis, (e) synthesis and (f) evaluation. Each level is based on the skills that were developed at levels preceding it. The revised version of Bloom's taxonomy replaced the nouns that are describing the levels of performance with verbs and has the following six categories: (a) remember, (b) understand, (c) apply, (d) analyse, (e) evaluate and (f) create. The SOLO taxonomy has the following five levels: (a) prestructural, (b) unistructural, (c) multistructural, (d) relational and (e) extended abstract. These levels are used to evaluate a learner's understanding of what is taught. However, the levels of the three taxonomies appear to be unable to properly describe the skills that are developed in applied disciplines, like the teaching and learning of programming (Fuller et al., 2007).

The diversity of the definitions about the learning objectives of teaching and learning programming reveals that there is not much unanimity on the subject. However, two underlying aspects are common amongst the most popular definitions. The first one is that programming instruction aims to promote the understanding of programming concepts. The second one is that programming instruction aims to enable a programmer to communicate his/her ideas and directions to a computer using these programming concepts in the form of programs (Pair, 1990). Cognitive psychology defines these two aspects as types of knowledge. More specifically, the first one is considered as declarative knowledge and the second one as procedural knowledge (Davies, 1993; de Jong & Ferguson-Hessler, 1996). Therefore, this thesis considers the fostering of declarative and procedural knowledge in the domain of introductory programming as the learning objective of programming instruction (Palumbo, 1990). This description is also supported by constructivism. According to constructivism, knowledge acquisition involves the creation and modification of schemata. Schemata are knowledge structures that contain declarative and procedural knowledge (de Jong & Ferguson-Hessler, 1996). Hence, learning is viewed as a process of acquiring declarative and procedural knowledge. Therefore, learning to program can be described as the acquisition of declarative and procedural programming knowledge. Because of the importance that these types of knowledge have in this project, they are further discussed in Section 2.7.1.

2.4 Difficulties Experienced by and Misconceptions of Novice Programmers

In order to improve the teaching and learning of programming, it is critical to identify what is hampering it. Two crucial factors that hamper programming instruction are the misconceptions that novices have and the difficulties that they face in their introductory course. Thus, this section is concerned with the misconceptions and difficulties of novices that are reported in international literature.

The term ‘misconception’ is preferably used in this section, instead of the term ‘mistake’, because from a constructivist viewpoint, what is considered a mistake is just a corollary of a learner’s misconception(s). Therefore, it is more useful to locate and identify what learners seem to misunderstand than the results of these misunderstandings, namely their mistakes.

A novice programmer is considered to be a learner, who is engaged in programming instruction and has no previous programming experience. As the novice develops his/hers programming skills, he/she progresses through the following stages of programming efficacy: advanced beginner programmer, competent programmer, proficient programmer and expert programmer (Robins et al., 2003).

It has been reported that the majority of novices acquires only a low level of programming knowledge and skills during their first programming course (Garner, Haden & Robins, 2005; Kurland, Pea, Clement & Mawby, 1989; Linn & Dalbey, 1989; Winslow, 1996). This can be attributed, to a large extent, to the factors that make programming difficult for novices. Thus, the identification of these factors can not only provide a means for supporting novices through their first experience with programming, but also can be a guide for the improvement of the teaching methods, PLs and tools that are employed in programming instruction. A presentation of the most common and important difficulties and misconceptions of novices, which was greatly aided by the review of Robins et al. (2003), follows.

2.4.1 Difficulties with Syntax of Programming Languages.

A commonly cited difficulty that novice programmers face is the syntax of a PL (Gomes & Mendes, 2007; Guibert, Girard & Guittet, 2004; Simon et al., 2007; Teague & Roe, 2008). The syntax of a PL is a set of rules on how to correctly write code in this PL and defines the correct spelling and order of commands as well as the available punctuation marks (Hristova, Misra, Rutter & Mercuri, 2003). If the commands of a program are not written according to the PL’s syntax then the program cannot be executed and the computer informs the user of the syntactic errors. Once these errors are corrected then the program can be executed. The syntax of most PLs is designed to enable experienced programmers to write code fast and efficiently. This design principle regarding the PL’s syntax often leads to a set of unnatural rules. These rules are unnatural in the sense that they are different from the ones that people use in human languages. Consequently, this unfamiliar syntax is difficult and time-consuming for a novice to grasp. Furthermore, having to remember the PL’s unnatural syntax during the demanding task of program creation makes this task even more demanding by increasing intrinsic cognitive load.

Some examples of how a PL’s syntax can become a source of difficulties for novices follow. Many PLs use a semicolon to denote the end of programming statements. However, with certain statements, for example an IF statement, the use of a semicolon will result to syntactic error (Ross, 2000). In other cases, for example in a WHILE statement, it is not wrong to use a semicolon, but it substantially changes the function of the statement (du Boulay, 1989). Because only a syntactically correct program can be executed, Grandell, Peltomäki and Salakoski (2005) suggested that novice students may put considerable effort in producing a syntactically correct program at the expense of producing an algorithmically correct program. McIver and Conway (1996) mentioned the concepts of *syntactic synonym*, *syntactic homonym* and *elision*. The first concept describes a syntactic feature of PLs, which allows that a particular programming statement can be used in programs following two or more different syntactic expressions. The second concept refers to the case where different statements follow the same syntax. Elision, the third concept, is the syntactical feature that allows omitting a specific part of a statement. Obviously, the three concepts can be quite problematic for novice programmers, in other words a novice can mix up different syntaxes when entering a statement or different statements with the same syntax and be baffled when a part of syntax can be omitted.

Many studies tend to ignore syntactical errors because they account only for a small percent of novices' errors (Allwood, 1986) or because they can be easily detected by the compiler/interpreter of the PL (McIver, 2000). However, syntax is a source of problems for novices that must not and cannot be overlooked. Evidently, studies still report the "persistence, frequency, and uniform distribution of problems relating to basic syntactic details" (Garner et al., 2005, p. 178) and that they "have a substantial impact on the experience of learning to program" (McIver, 2000, p. 7).

2.4.2 Misconceptions Regarding the Semantics of Programming Languages.

The semantics of PLs can be a source of misconceptions for novices (Corritore & Wiedenbeck, 1999; Youngs, 1974). The semantics of a PL describes how to derive meaning from this PL's commands. As in the case of PLs' syntax, semantics was designed to offer efficiency in code development, neglecting its naturalness. A semantic error is committed when programmers suppose that a command performs a certain action, which, in reality, it does not perform. Semantic errors occur because of novices' misconceptions about programming concepts. When a student fails to learn that a programming construct has a specific functionality in a program or he/she fails to correctly understand the specific functionality, then the student holds a misconception of the specific programming concept. These misconceptions can be generated by: (a) insufficient knowledge of programming concepts and (b) knowledge associated with natural language and human interaction (Guibert et al., 2004).

The first source of novices' misconceptions is misunderstanding or lacking to understand how a programming concept can be used in a program. Programming concepts are the building blocks of programs and their understanding is very important in programming. Nevertheless, these concepts are generally difficult for novices to grasp. One reason for this is that they are designed based on the operations that computers can perform. Novices, however, have no everyday experience with similar constructs from other domains that would enable them to spontaneously use concepts in the programming domain (Rogalski & Samurçay, 1990). This can make programming concepts seem unnatural and foreign to humans. The fact that, to a greater or lesser extent, the learning and understanding of every programming construct can be problematic for novices and be a potential source of semantic errors is supported by numerous studies. For each of the following programming concepts there is at least one study, which reports that the particular programming concept in a certain PL can be difficult for novices: variables (du Boulay, 1989; Rogalski & Samurçay, 1990; Samurçay, 1989), input/output statements (Dagdilelis et al., 2004; Ebrahimi, 1994; Putnam, Sleeman, Baxter & Kuspa, 1989), assignment statement (Ebrahimi, 1994; Kaczmarczyk, Petrick, East & Herman, 2010; Putnam et al., 1989), iteration statements (Putnam et al., 1989; Rogalski & Samurçay, 1990), conditional statements (Ebrahimi, 1994; Putnam et al., 1989; Rogalski & Samurçay, 1990; Ross, 2000), arrays (du Boulay, 1989), recursion (George, 2000; Kahney, 1989), functions/procedures (Milne & Rowe, 2002), parameters (Grandell, Peltomäki, Back, Salakoski & 2006) and classes (Kinnunen & Malmi, 2008).

The second source of misconception about programming constructs is knowledge students bring from their experience with other humans and natural language

(Spohrer & Soloway, 1989). When novices attempt to interpret the meaning of commands, they often find themselves at an impasse due to their limited programming knowledge. Novices, having nowhere else to draw conclusions from, turn to natural language semantics to interpret the meaning of commands (Bonar & Soloway, 1983; Kurland & Pea, 1989). For example, Ala-Mutka (2004) reported that the 'while' iteration-statement was interpreted by novices as a statement that is executed continuously, instead of a statement that tests its condition in every iteration. Similarly, Pane, Myers and Ratanamahatana (2001) reported that the word 'then', often used in conditional statements as *consequently*, was associated by the participants of their study with the sequential form of programs. Furthermore, Bonar and Soloway (1989) reported that novices have problems when they attempt to transfer a step-by-step solution, which is expressed in natural language, into a program.

Equally problematic can be the experience of human interaction that novices bring in their interaction with computers. Novice programmers are not aware of the low-level processes computers use to execute commands and process data (Ben-Ari, 2001). This lack of information can lead novices to base their interaction with computers on their previous interaction experiences, which are mostly built on human interaction. Thus, novices often expect from computers to react as humans would do. Pea (1984) called this attitude, namely expecting computers to interact as if they had a secret mind of their own, the 'superbug'. He classified the "conceptual misunderstandings (or 'bugs')" (Pea, 1984, p. 3) into three categories according to their source: parallelism bugs, intentionality bugs and egocentrism bugs. Parallelism bugs are generated by the assumption that computers can process or be 'aware' of two or more commands at the same time. Intentionality bugs are produced when programmers expect from computers to guess or speculate what they intend to accomplish based on the programming statements and data they provide in a program. Egocentrism bugs are created when programmers expect that computers can go beyond what they write and understand what they mean. Novices tend to forget that computers can process only one command at time and that each command operates in the programming environment that has been created by the execution of previous commands. Instead, they assume that computers can have a full view of a program, make sense of this program, infer the programmer's intention and even correct or fill in any wrong or missing details of the program, according to the intended result.

However, Spohrer and Soloway (1989) argued that misconceptions about programming concepts are not as extensive as believed. They suggested that this belief is so popular, because teachers relate bugs to the programming constructs that should be used to correct them. Therefore, teachers could suppose that novices have problems with correcting these bugs, because of novices' misconceptions about the semantics of these constructs. Spohrer and Soloway (1986, as cited in Simon et al., 2007) also found that only few of the bugs in their participants' programs could be attributed to participants' misconceptions, while the majority of the bugs was associated with the logic of programs.

Overall, semantics errors can be very problematic and frustrating for novice programmers, even if they are not as common as believed. Programmers use the available programming concepts of PLs to transform a series of actions into a program. In the case that novices lack a clear understanding of one or more concepts or they interpret their functionality wrongly, the correct transformation of these actions into a program becomes extremely difficult or impossible.

Furthermore, due to their nature, semantic errors are more difficult to detect than syntactic errors; moreover, compilers and interpreters, with the exception of static semantic errors, cannot inform programmers about them.

2.4.3 Difficulties with Problem-Solving, Programming Strategies and Programming Plans.

The high level of connection between problem-solving and programming was discussed in Section 2.3. In a typical programming course students are asked to solve problems programmatically. Winslow (1996) described how problem-solving is applied in programming in four stages: (a) students must understand the problem, (b) produce a solution for the problem and transfer it to the programming domain (translate it into a computer-equivalent form), (c) create a program based on the computer-equivalent form using a PL and (d) execute and (if necessary) correct the program. The second stage has proven to be a very difficult task. Lister et al. (2004) reported that many students are not able to abstract a problem from its description. An explanation for this phenomenon could be that students lack in general problem-solving skills (Gomes & Mendes, 2007; McCracken et al., 2001). However, it is reported that even when students know the solution they are unable to transfer it into the programming domain, in other words express it in the form of a computer program (Winslow, 1996). Furthermore, Green and Petre (1996) introduced the term closeness of mapping to describe the mental distance between the problem domain and the programming domain. They argued that the greater this distance is, the more difficult the solution to the problem becomes.

A very significant difficulty is that the majority of novices, even when they have overcome problems with syntax and semantics, are bewildered about how to use the programming concepts they know in programs. This difficulty is often cited in the international literature (Lahtinen et al., 2005; McCracken et al., 2001; Porter & Calder, 2003; Simon et al., 2007). In order to describe the processes involved in the generation of structured solutions to problems in the form of pseudocode as well as their transformation to programs, the terms programming strategies and programming plans were introduced. *Programming strategies* describe knowledge of how to use programming constructs to produce solutions for problems (de Raadt, 2008), while *programming plans* are stereotypical predefined solutions in the form of code that perform a specific action, for example averaging the elements of an array (Soloway & Ehrlich, 1984). Programming strategies seem to describe what cognitive psychology defines as procedural programming knowledge. Accordingly, syntax and semantics of PLs are components of declarative programming knowledge.

Various difficulties for novices have been identified regarding programming strategies and plans. Soloway (1986) demonstrated that when experts engage in program creation, they recognise a number of programming goals and employ their programming strategies and plans to achieve these goals. They have formed a large set of programming strategies and plans over time through practice and problem-solving, and are able to retrieve and modify the required information from this set. Novices lack this accumulated body of strategies and plans, and therefore must devise their own plans instead of retrieving them (Robins et al., 2003). This activity can be extremely challenging and demanding. Spohrer and Soloway (1989) proposed nine categories of problems that could cause difficulties in program creation for novices: (a) summarisation problem, (b) optimisation problem, (c) previous-experience problem, (d) specialisation problem, (e) natural-language problem, (f) interpretation problem, (g) boundary problem, (h) unexpected-cases

problem and (i) cognitive-load problem. Spohrer, Soloway and Pope (1989) found that novices inappropriately combined different plans, thus producing a ‘merged plan’, in order to implement two plans with the same code. Because, the plans should not have been combined in the first place, this combination was only an extra source of problems. De Raadt, Watson and Toleman (2006) proposed that the explicit instruction of goals and plans would be beneficial for novices. Crucially, Gilmore (1990) suggested that programming plans are insufficient without the knowledge of how to apply them, namely programming strategies.
Spohrer and Soloway (1989) argued that a majority of bugs in novices programs is related to the plan-composition problems that novices have rather than to misconceptions about programming constructs. However, Ebrahimi (1994) demonstrated that plan-composition problems and misconceptions about programming constructs are highly correlated. He calculated the correlation between plan-composition problems and programming-concept problems across four PLs (Pascal, C, Fortran and LISP) and the values of *r* ranged from 0.91 to 0.97. Not surprisingly, he stated that “that knowledge of Plan Composition and Language Constructs is indispensable to the understanding of basic programming” (Ebrahimi, 1994, p. 477). Furthermore, Kranch (2012) suggested that novices should first familiarise themselves adequately with the basic programming concepts in order to tackle successfully more complex issues, such as programming strategies and plans.

2.4.4 Difficulties with Error Messages

Novices are reported to have problems with the error messages produced by compilers and interpreters (Marceau, Fisler & Krishnamurthi, 2011; Nienaltowski, Pedroni & Meyer, 2008; Simon et al., 2007). Once again, the design choices that were adopted for making PLs fast and efficient seem to be a source of problems for novices.

The wording of errors messages can be very cryptic for novices and, practically, novices find it hard to understand them (Allwood, 1986). This, to a very large extent, can be attributed to their poor design and the way compilers parse programs (Traver, 2010). Often these messages do not point exactly to the command that should be corrected. Furthermore, they scarcely provide useful information, from novices' viewpoint, on how the particular error should be fixed. This can result into taking unnecessary or, even worse, incorrect actions in order to respond to error messages. Other reasons that reinforce the difficulties of novices with error messages include possible mismatch of linguistic attributes between novices' natural language and the PL and/or IDE, a lack of experience and the limitations of human memory and attention (Traver, 2010).

2.4.5 Difficulties with Tracing and Debugging Programs

Two tasks that are required for programming and are difficult for novices to perform are tracing and debugging their programs, respectively. The tracing of a program has to do with reading, understanding and mentally following a program and its execution, while the debugging task has to do with detecting and correcting program errors.

Lister et al. (2004) reported that students have more problems reading a program than writing one. They also considered reading and tracing skills as perquisites for problem-solving. The distinction between the abilities to read a program and write one was emphasised by Winslow (1996) as well. Vainio and Sajaniemi (2007) identified four problematic aspects of program-tracing (a) single-value tracing, (b) inability to distinguish between functions and structures, (c) inability to use external representations and (d) inability to raise abstraction level in order to manage cognitive load. Strategies that novices use to read and understand programs were studied by Fitzgerald et al. (2005), who concluded that these strategies are applied poorly. They noted that this could indicate that novices lack efficient knowledge to read and trace code. Despite the problems novices have with reading programs, Manilla (2007) noted that they seem to determine the difficulty of a program based on the effort required to produce it rather than to read it.

Allwood (1986) reported that novices have difficulties with error detection; they devote a large proportion of their time to this and are reluctant to take actions for correcting their errors. An explanation for this phenomenon could be the limited programming knowledge that novices have. Because of their limited knowledge novices do not examine many hypotheses for the origin of program errors and thus, often select an incorrect hypothesis, which aggravates this problem further (Gilmore, 1991). Perkins, Hancock, Hobbs, Martin and Simmons (1986) categorised novices according to their attitude towards program errors in three groups: (a) stoppers, (b) movers and (c) tinkerers. Stoppers were considered students that stumbled on a problem and could not continue without help. Movers were considered students who attempted to overcome their errors by successfully modifying their programs. Students, who modified their program continuously and randomly, often producing more errors than corrections, were considered as tinkerers. Generally, detecting and correcting program errors seem to be very difficult tasks for novices. However, identifying programming errors is considered to be more difficult than correcting them (Robins et al., 2003).

2.4.6 Other Difficulties

Du Boulay (1989) used the term ‘notional machine’ to refer to a model that describes how the system makes “sense of the program” (du Boulay, 1989, p. 287). The notional machine of a PL is defined by the programming commands of the PL and thus each PL has a different notional machine. The notional machine provides a basis for predicting and anticipating the behaviour of running programs written in the specific PL. Du Boulay argued that there are difficulties with understanding the notional machine and how the textual representations of programs relate to it. The importance of the notional machine was stressed out by Berry and Kölling (2014), as they designed a notional machine for the BlueJ programming environment and software that visualises diagrams of this notional machine. However, the effectiveness of this tool has not yet been formally tested.

Another difficulty that novices have is that they find it hard to elaborate their weaknesses and problems with respect to programming (Allwood, 1986; Mannila, 2007). This phenomenon not only renders some of the difficulties that novices have obscure, but also prevents the in-depth analysis of the already known ones.

2.4.7 Programming Behaviour and the Nature of Novices' Programming Knowledge

Based on the difficulties discussed in the previous sections, some general remarks can be made about the novice programmers’ behaviour and nature of their knowledge. In their review, Robins et al. (2003) discuss the exploratory nature and opportunistic design that novices employ during programming. Novices employ ‘depth-first’ and ‘bottom-up’ strategies when they write programs (Rist, 1989) and develop code by trial-and-error (Allan & Kolesar, 1997). Furthermore, when they are in an impasse, they resort to previously-written code and modify it without a specific purpose until the compiler produces no errors, but without being able to justify their course of actions (Gaspar & Langevin, 2007).

Many students seem to comprehend programming structures in isolation and have fragile and fragmented knowledge of programming constructs and regular programming tasks (Lister et al., 2004; Mannila, 2007). Fragile knowledge is considered to be insufficient knowledge or inability to use knowledge appropriately (Ko & Myers, 2005). Novices comprehend programs two times slower than experts and form only half of the chunk levels that experts do (Kranch, 2012). Novice programmers usually understand code based on the surface characteristics of programs, such as syntax, create programs ‘line by line’ instead of a more structured approach and are unable to efficiently use already acquired knowledge (Clear et al., 2008; Kurland et al., 1989; Lahtinen et al., 2005; Mayer, 1989). They also fail to recognise basic patterns in programs and comprehend programs as a whole (Fix, Wiedenbeck & Scholtz, 1993). Programming representations of novices vary greatly and their type of programming knowledge is more context-specific (Allwood, 1986). A final remark concerns how correctness of programs is perceived by students. Novices consider programs to be correct, even if they cannot be executed, as long as they contain a ‘core’ of correct statements (Kolikant & Mussai, 2008).

2.4.8 Difficulties Experienced by and Misconceptions of Novice Programmers in Greece.

My profession at the time that this research was conducted was CS teacher in Greek educational institutes. This included the teaching of introductory programming courses. This fact motivated me to study the current state of international and Greek introductory programming education and to contribute to its improvement. Therefore, the teaching and learning of programming in Greece is of special interest for this project. However, only one review, regarding the difficulties students face during programming instruction in Greek secondary schools, was found in the international literature.

Dagdilelis et al. (2004) presented the major difficulties of novice programmers in Greek secondary schools. They reported that students have problems with input statements, variables and data structures, have misconceptions about programming constructs, engage in an anthropomorphic communication with computers and have difficulties creating programs. Not surprisingly, their findings are similar to the ones mentioned in the international literature.

I conducted a small-scale interview study with teachers of computer science, who teach programming in Greek secondary and tertiary institutes. One of the reasons for doing this was to further provide evidence about the actual problems of novices in Greek educational institutes. Therefore, a part of this small-scale study was concerned with the difficulties that novices face during programming instruction. The interviews revealed that students cannot combine elements of their programming knowledge effectively to create programs, hold misconceptions about programming constructs and have problems with programming commands. My findings are in agreement with those reported by Dagdilelis et al. (2004) as well as with those reported in the international literature. These findings as well as an executive summary of this small-scale study are reported in Appendix A.3.

2.5 Seven Design Principles

The discussion of constructivism and CLT, as well as the difficulties of novice programmers during their introductory programming course, provides the framework for identifying design principles for educational PLs and IDEs. Seven principles were identified as important for the design of educational software for introductory programming. These are: (a) match of users' natural language with the linguistic attributes of the PL and IDE, (b) syntax and semantics of instructions supported by the PL, (c) visualisation, (d) abstraction of commands provided by the PL, (e) small set of instructions, (f) provision of error messages and (g) a high level of interaction with the IDE. These principles are presented and discussed in the following subsections.

2.5.1 Match of Users' Natural Language with the Linguistics Attributes of the Programming Language and the Integrated Development Environment

The vast majority of PLs and IDEs discussed in international literature are in the English language. This is no surprise, because English is the natural language for a very large proportion of the people that design and use these PLs and IDEs. Furthermore, English is spoken by over 335 million people around the world (Paul, Simons & Fennig, 2013). However, people who want or need to use this software and are non-native English speakers need to have at least an elementary understanding of English (Kodama, Sato & Miyazaki, 2000). For information technology (IT) professionals, who are non-native English speakers, this constitutes no obstacle, because a significant number of them possess this elementary understanding. Nevertheless, things are very different for novice programmers who are non-native English speakers, have a scarce understanding of English and have to work with these tools. Furthermore, programming with these tools becomes virtually impossible for novices who do not speak English at all. As the discipline of programming is introduced in secondary schools or even elementary schools, students are more likely to lack an elementary understanding of English. This could also be the case for older people who would like (or have) to engage in programming, but their knowledge of English is insufficient.

A central aspect of programming involves the mental mapping of a plan to a program using the programming constructs of a PL. However, if the linguistic attributes of this PL are foreign to its users, then this mapping must be performed in two stages. The first stage is the mapping of the plan to an equivalent form, in users' natural language, of computer programs. The second one is the mapping of this equivalent form in users' natural language to a program consisting of programming constructs with foreign linguistic attributes. Perhaps, the paramount necessity to avoid the two mappings could be presented in a clearer way with the following example.

A popular analogy for programming instruction is the learning of a foreign language (Black, 2006; du Boulay, 1989). Obviously, the learning of a foreign language involves translation of words. When the linguistic attributes of PLs are foreign to novices, they must translate them to their natural language. Thus, programming with the foreign linguistics attributes of PLs results to using a translation of a translation. If we take the analogy further, this is equivalent of learning a foreign language through another foreign language, for example an English speaker learns German by translating the English words first in Spanish, and in turn, the Spanish words in German.

The futility of this undertaking is obvious. What may be less obvious is that between these mappings critical information for knowledge acquisition might be lost or misunderstood. These mappings are not necessarily straightforward and in many cases they are characterised by a degree of ambiguity. Programming constructs are already problematic for novices, even when they are in novices' natural language (see Section 2.4.2). In the case of novices who are non-native speakers of the language incorporated by the PL and IDE, this phenomenon could be further exacerbated by the additional and ambiguous mappings between foreign and mother language. Thus, more misconceptions may be created, which potentially may be more difficult to correct. Furthermore, people who do not understand the exact meaning of programming concepts used to solve problems may develop poor problem-solving skills.

The following analysis, based on CLT, provides further support for this principle. Evidently, the translations that a novice who is not fluent with the linguistic attributes of the PL and IDE need to make require cognitive resources in working memory. This requirement reduces even more the limited capacity of working memory. Programming, however, is known to be cognitively demanding (Mow, 2008). Thus, the additional reduction of cognitive resources makes learning programming more difficult. Moreover, ineffective schemata may be formed, due to an increase in extraneous cognitive load produced by managing the translation and comprehension of programming concepts at the same time. Traver (2010) noted that the mismatch between novice's mother tongue and the linguistic attributes of PL and IDEs could be one of the reasons that novices have trouble with error messages.

It is also reported that IDEs should match the linguistic attributes of the users' native language and the mental mappings novices have to do during programming should be minimised (Pane & Myers, 2000; Pane et al., 2001). Barnes, Fincher and Thompson (1997) suggested that it is easier for novices to solve problems in their native language before transferring the solution to programs. As previously discussed, if the PL and IDE do not match students' linguistics attributes an additional mapping is required. Furthermore, Pillay and Jugoo (2005) reported that programming ability is positively influenced when the language of programming instruction is the same with the natural language of students. Moreover, readability, which is the level of effortlessness that is required to read something, of programs is a crucial factor for program comprehension (Georgatos, 2002; Murnane, 1993). Apparently, programs in users' native language would be easier read than programs in a foreign language.

Therefore, in order to effectively facilitate the teaching and learning of programming, the linguistic characteristics of the PL and IDE should match as close as possible those of its users. Ideally, these characteristics should be the same.

2.5.2 Syntax and Semantics of Instructions Supported by the Programming Language

Instructions of PLs convey their meanings to programmers through their semantics. Additionally, with the exception of visual programming languages, the programs written in a particular PL are subjected to the rules of the PL's syntax. Thus, the syntax and semantics of the instructions supported by a textual PL are the features that render the programs in this PL readable and meaningful. Without these features, programs in textual PLs would be incomprehensible, not unlike a text in an unknown language. Therefore, syntax and semantics are two major features that characterise each PL and consequently, its design.

The difficulties caused by the syntax and semantics of PLs were discussed in Sections 2.4.1 and 2.4.2, respectively. Based on these difficulties as well as on suggestions mentioned in international literature, the following design guidelines concerning syntax and semantics of programming statements can be proposed.

Syntax is only a means to communicating with computers. However, syntax is a source of many problems for novices, because it can feel very unnatural. An approach to alleviate this difficulty is the design and implementation of natural-language PLs (Myers, Pane & Ko, 2004; Riker, 2010), like Pegasus (Knöll & Mezini 2006). However, this approach suffers from its own weaknesses, for instance ambiguity of naturalness' definition, the openness of natural language and a lack of consistency (Green, 1990a; Pane et al., 2001; Yin, 2010). Furthermore, a certain level of formality is unavoidable or even preferred for natural-language PLs (Bruckman & Edwards, 1999). In addition, readability, simplicity and consistency achieved (for instance, by avoiding syntactic synonyms) seem to be more important features of syntax than naturalness (Deek & Espinosa, 2005; Gupta, 2004; McIver & Conway, 1996; Pane & Myers, 1996). Particularly, the readability of programs provided by syntax is more important than the provided writeablity, in other words the ease of writing programs (Georgatos, 2002). Gaspar, Langevin and Boyer went even further so as to propose the "hypothesis that 'syntax doesn't really matter' to novice programmers' education" (Gaspar, Langevin & Boyer, 2008, p. 210). They also referred to attempts that used visual tools as alternative replacements of syntax. Additionally, Kahler (2002) stated that syntax can be problematic for students even when they understand how to complete a programming task.

Semantics can be another source of difficulties for novices and especially a source of misconceptions and a complicated problem to cope with. The semantics of PLs should facilitate the transition from human thinking to programming concepts (Kaasbøll, 1999). However, this is quite difficult because the words used in programming commands do not support a direct mapping between the use of these words in natural language and the use of these words in commands (Kurland & Pea, 1985). Moreover, semantics should be as unambiguous as possible and prevent possible conflicts with previous non-programming knowledge of novices. Ideally, semantics should promote programming concepts' understanding rather than hindering it (Pane & Myers, 1996). Moreover, different syntax should be used to distinguish different semantics; for example, parentheses should be used either to include the elements of arrays or to include the parameters of functions, but not in both cases (McIver & Conway, 1996). Semantics should also be simple to understand (du Boulay et al., 1999) and consistent with non-programming knowledge (Gupta, 2004; Pane & Myers, 1996).

These syntactic and semantic features should be taken into consideration when designing educational PLs. Consideration should also be given to the degree of implementation of each feature as well as the balancing between features. For example, excessively natural syntax can significantly improve readability, but it can also produce syntactic synonyms or reinforce semantic misconceptions, for example anthropomorphic characteristics of computers (Pane et al., 2001). Overall, syntactical and semantic features that promote simplicity and understanding for novices should be favoured over powerful or efficient characteristics that are preferred for experienced programmers.

2.5.3 Visualisation

Visualisation is widely believed to be very helpful for novice programmers. However, there is much scepticism as to whether this is true (Amershi, Carenini, Conati, Mackworth & Poole, 2008; Lahtinen, 2008; Mulholland, 1997; Myers, Ko & Burnett, 2006).

Visualisation in programming can be divided in two large categories. The first one is *program visualisation* or *programming visualisation*, whilst the second is *visual programming*. Programming visualisation involves the use of graphics to facilitate program understanding, after programs have been created via a textual PL. In contrast, visual programming uses graphical means to produce programs (Hu, 2004; Myers, 1990; Navarro-Prieto & Cañas, 2001; Price, Baecker & Small, 1993). Visual programming can be further categorised in graphical interactions systems and visual-language systems. The distinction between the two categories is that in the first category the system records and 'learns' from users' actions, while in the second, the spatial layout of visual symbols specifies programs. These visual symbols can be (a) control/data flow diagrams, (b) icons, (c) tables/forms or (d) others (Kiper, Howard & Ames, 1997).

Researchers have discussed the potential advantages of incorporating visualisations for programming instruction in classrooms. A number of reasons that make graphics so appealing to use are mentioned by Petre, Blackwell and Green (1998). For example, graphics provide information which can be easily understood because it is presented in a dense form and at a higher level of abstraction. Petre et al. (1998) argued that visualisations can make concrete concepts more abstract and thus, facilitate their grasp. They also suggested that visualisations could be advantageous for novices by presenting the sequence of program execution. Du Boulay et al.

(1999) underlined the importance of being able to view specific components and functions of the notional machine. This can be accomplished by dynamically describing the states of the notional machine through pictures or text. In particular, this approach can be rather useful in examining the activities of the notional machine during program execution by means of graphical or textual traces (du Boulay et al., 1999). Brusilovsky (1994) suggested that the semantics of programming concepts should be visually disclosed during execution. Furthermore, he argued that "visibility provides a feedback for exploratory learning and problem solving" (Brusilovsky, 1994, p. 108). This is quite interesting for its pedagogical implications from a constructivist viewpoint, because constructivism underlies the active construction of knowledge by learners and particularly explanatory learning under proper guidance. Hence, feedback from programming visualisations can help novices correctly construct and explore programming knowledge. Specifically for systems that support visual programming, Green and Petre (1996) argued that the creation of programs in visual languages could be considered easy due to the relaxed syntax restrictions, the provided level of abstraction and the freedom of choosing the order, in which to proceed with program creation.

In addition, the benefits of visualisation can be supported by CLT. According to CLT techniques and especially the *modality effect* (discussed in Section 2.2.2.2), the presentation of information in a graphic form can reduce cognitive load. Hence, if some programming components are displayed graphically, novices can retain more cognitive resources for problem-solving and program creation.

Many studies support the claims that visualisations can be beneficial for novices by reporting positive results of incorporating visualisations in programming education. Kasurinen, Purmonen and Nikula (2008) used visualisation in introductory programming courses to increase students' motivation. They reported a 10% improvement in exercise grades and passing rates and, based on these results and literature, they suggested that there is a need for visualisation in introductory courses. Ebel and Ben-Ari (2006) reported an improvement in students' behaviour in class when using program visualisation. George (2000) reported that diagrammatic traces helped novices acquire better mental models of recursion than students who did not use diagrammatic traces. Ramadhan, Deek and Shihab (2001) found that visualisations make program diagnosis systems more efficient in supporting its users. They also reported that the dynamic visualisation of program behaviour could help novices acquire a clear mental model of programming and improve their problem-solving skills. Sajaniemi and Kuittinen (2003) used program visualisation and specifically program animation, to distinguish the roles of variables in programs. They reported that participants improved their program comprehension and program-writing skills. Moons and De Backer (2009) used visualisations in a programming tool for an object-oriented course. Their aim was to facilitate novice programmers in understanding programming concepts, debugging their programs and increase their motivation. However, they did not report any results for the effectiveness of their tool. Lieber, Brandt and Miller (2014) used an IDE extension that provided always-on, real time visualisation of programs' execution to help users deal with their misconceptions about their ideas of what a piece of code does and what it actual does. They tested this IDE with graduate students, who had programming experience and professionals and reported positive results in both experiments. Vihavainen, Airaksinen and Watson (2014), in their meta-analysis, reviewed the pre-intervention and post-intervention pass rates of 60 studies that employed 13 different categories of teaching approaches for improving introductory programming. The results revealed that interventions using visual programming tools was among the top five based on absolute and realised improvement of pass rates.

Nevertheless, there are also empirical studies on the effectiveness of visualisations in programming that did not verify the expected effects. Myers (1990) highlighted a number of problems with visualisations, for instance they are neither suitable for large programs or data sets nor portable. Ihantola, Karavirta, Korhonen and Nikander (2005) reported that developing multi-language visualisations – versions of the same visualisation in different languages – is often neglected. Hundhausen (2002) draws attention to the fact that visualisations in class, under particular circumstances, can also be a source of distraction. A similar remark was made by Petre (1995), who argued that novices might not get the expected help out of visualisations, because they lack the knowledge to explore visualisations effectively. Green and Petre (1996) noticed that making changes in visually created programs requires a considerable amount of effort and time. More important, Hundhausen, Douglas and Stasko (2002) concluded that visualisations are rendered effective by the way they are used and not by their context. Furthermore, they noted that visualisation with activities within a constructivism framework were more successful than with activities within other theoretical frameworks. Naps et al. (2002) reached a similar conclusion about the use of visualisations, namely that visualisations can have a positive effect on novices only when novices adopt an active role towards them.

A relevant issue is how information should be represented in a visualisation. Larkin and Simon (1987) discussed how diagrammatic representations can be more beneficial in problem-solving than textual representations. They mentioned that diagrams can present more information, make visible implicit information and support useful indexing of information. However, they remarked that diagrams can be used effectively by those who have sufficient knowledge to do so. Edwards (2005) noted that diagrams can convey compact information, but fail to represent details, especially when they are large. Blackwell, Whitley, Good and Petre (2001) discussed the applications of diagrams and particularly flowcharts in visual programming. They concluded that diagrams can be helpful for novices in certain programming tasks, like tracing control flow, but not for the entire programming activity. Similar arguments for the beneficial use of flowcharts in program creation were made by Scott (2010). Navarro-Prieto and Cañas mentioned that pictorial representations and especially icons can "be helpful because they facilitate the access to semantic information" (Navarro-Prieto & Cañas, 2001, p. 810). Flowcharts and icons have been used as visual PL notation for procedural and object-oriented courses (Calloni, Bagert & Haiduk, 1997; Scott, 2010)

Furthermore, there is also an ongoing debate on whether a graphical or a textual representation of programs is more beneficial for novices (Green, 1990b). Both approaches have been found to have positive effects and the superiority of one representation over the other has not yet been determined. Calloni and Bagert (1994; 1995) reported that the users of an iconic programming environment performed better in exam scores and syntax understanding than users of textual languages, PASCAL and C++, respectively. Similarly, Cilliers, Calitz and Greyling (2005) reported that using a programming tool with icons in an introductory programming course improved the performance and passing rates of novices. They also suggested that a textual representation of programs alone is not beneficial for novices with respect to their program understanding and writing skills. Nevertheless, textual forms seem to provide better representation for large programs, while graphics are better for smaller programs (Blackwell et al., 2001; Edwards, 2005; Pane & Myers, 1996).

In conclusion, visualisation in programming can be beneficial for students as long as it engages them in an active learning role. More particularly, program visualisation could be used for presenting explicitly program execution and supporting the understanding of semantics, while visual programming could be effective for creating small programs with the use of diagrams and icons.

2.5.4 Abstraction of Commands Provided by the Programming Language

Abstraction is a mechanism that allows humans to deal with a problem or situation by hiding irrelevant details. For example, one does not need to know the exact way that a car pedal works in order to drive. It suffices to know that stepping on the accelerator or the brake pedal makes the car move or stop, respectively. At this level, the exact workings of these pedals provide no useful information and thus it can be ignored. However, for a car mechanic this kind of information is crucial and necessary. Hence, the context within which an abstraction is used, determines the level of the abstraction. Furthermore, abstraction fosters a reduction in extraneous cognitive load by hiding unnecessary information (McIver, 2000; Shaffer et al., 2003) The ability to abstract is fundamental in many science disciplines, including programming (Blackwell, 2002a; Gomes & Mendes, 2007; Hazzan, 2003). In fact, the nature of programming includes a constant use of abstractions in various levels, because there is no direct mapping between problem domain and programming domain (Green & Petre, 1996). Hence, abstraction is a valuable skill for programmers as well as an important feature of programming languages. Programmers use abstractions to solve problems and create programs (Návrat, 1996), while PLs use abstraction to provide programming concepts with a certain level of detail, which facilitates programming (Pane & Myers, 1996). The abstract concepts of programming become more concrete via their implementation in a PL in the form of commands. However, the details of commands' implementation are hidden from users. Thus, the implementation choices of these programming concepts in a PL determine the level of abstraction (or concreteness) of PL's commands (Návrat, 1996).

Abstraction is examined in this thesis as a feature of PLs. Achieving the appropriate level of abstraction for a PL is difficult (Green & Petre, 1996). This is especially true in the case of PLs for novice programmers. The level of abstraction of commands provided by PLs for novices should enable them to perform various operations without being difficult to use (Gupta, 2004). McIver and Conway (1996) proposed that the abstraction level of commands should be as close as possible to the abstraction level of the problem domain and should be neither too high nor too low. In particular, the abstraction level of commands should be high enough to hide unnecessary technical details, for example how an output command is implemented, and to allow users to focus more on the functionality of commands, for instance what the programming effect of an output command is. This approach not only reduces potential cognitive load produced by irrelevant details, but also hides details that could be distracting. However, a high level of abstraction could obscure critical information that is required for the comprehension of commands and possibly cause misconceptions. Furthermore, abstractions at the appropriate level could make a program more understandable (Green & Petre, 1996).

2.5.5 Small Set of Instructions

The majority of PLs for novice programmers consists of a small set of programming instructions (Dagdilelis et al., 2004). An educational PL is not a software tool for producing professional programs. Therefore, the full capabilities of a professional PL are not required. Thus, an educational PL could be either a mini-language or a sub-language. A mini-language supports only the programming constructs that are fundamental in introductory programming, while a sub-language is a subset of a professional PL and supports only the commands that implement basic programming constructs (Brusilovsky, 1994). Given that the basic programming constructs, like input, output, iteration and conditional statements, need to be supported, the emerging set of programming commands would be rather small.

The commands that would be included in this set should also meet a high-level of orthogonality. That is, the dependence between the commands of the set should be as little as possible. Thus, there should be only one way for performing a specific action (Georgatos, 2002; Wiedenbeck, 1985).

The implication of this design principle can have some positive effects. A small set of programming concepts would be easier to learn (Gupta, 2004). The syntax of these concepts could be simpler, because there would be only a few syntactic rules to remember (McIver & Conway, 1996). This way, syntactic synonyms and homonyms could be avoided. Furthermore, semantics could become clearer to novices, because it would be easier to distinguish between programming commands of a small set. Moreover, it is apparent that a small number of instructions can contribute to the minimisation of intrinsic cognitive load during programming (Brusilovsky, 1994).

2.5.6 Provision of Error Messages

As discussed in Section 2.4.4, error messages are a major source of difficulties for novice programmers. Not only novices are confused by the complexity of error messages, but they can also extract little information on how to fix them. Error messages can be useful in learning programming, only when programmers can understand and use them (McIver & Conway, 1996). Various recommendations have been proposed towards the aim of making error massages more comprehensible and supportive for novice programmers. Provision of 'good error messages' is one of the usability heuristics proposed by Nielsen (1993). Furthermore, du Boulay et al. suggested that error messages "form an important window into the machine" (du Boulay et al., 1999, p. 267). The term machine in the previous phrase refers to the notional machine of the PL.

Allwood (1986) suggested that error messages should be more detailed and informative instead of consisting of a couple of words. However, Nienaltowski et al. (2008) reported that the length of error messages is not as important as is their placement and their form. Similarly, the importance of enhanced error messages for novice programmers was investigated by Denny, Luxton-Reilly and Carpenter (2014). They developed CodeWrite, a web-based tool that provided enhanced error messages and examples on how to correct them. However, no significant differences were found in the debugging performance of programmers between the control group and the group that used CodeWrite. Possible explanation for their findings were that errors were simple and could be corrected without the use of CodeWrite and that the additional information was not used by students – they did not pay much attention or could not be used in their code. Marceau et al. (2011), after examining students' responses to error messages, recommended that the presentation of error messages should use simple non-technical language and colours to highlight code. Colour should be used to highlight location in the code of commands reported by error messages. Similar suggestions were recommended by McIver and Conway (1996) and Traver (2010). Traver (2010) also proposed that error messages should be specific, context-insensitive, non-hostile towards users and avoid anthropomorphic characteristics. Du Boulay and Matthew (1984) proposed that special checkers should be used for programs created by novices. These checkers should provide understandable and informative error messages before compilation. Only after a program has successfully passed all checkers should it be submitted for compilation. Thus, the error-checking function of compilers should be set apart from its function of producing intermediate code.

2.5.7 A High Level of Interaction with the Integrated Development Environment

The term interaction describes a process of exchanging “information, responses and feedback between the learner and the computer” (Jih & Reeves, 1992, p. 40) and it is hypothesised to increase learners’ performance and motivation. According to constructivism, the active involvement of students in knowledge construction is very important. Students can learn from their own discovery, but this does not necessarily lead to effective learning. Effective learning aims at constructing knowledge structures that represent reality sufficiently, accurately and consistently and this can be achieved through proper guidance, usually, provided by teachers (Ben-Ari, 2001). For example, visualisations seem to have positive pedagogical effects for novice programmers, only when novices are actively engaged in them (Hundhausen et al., 2002). Du Boulay et al. (1999) suggested that interaction with the system of a PL can decrease the distance between its notional machine and its users. Therefore, engaging novices in an active communication with the IDE, by providing a high level of interaction, could be beneficial in programming, especially during the creation of programs. It is important, though, that this active communication promotes effective learning.

A high level of interaction between the IDE and its users could be achieved through the provision of proper guidance throughout the creation of programs and thus foster effective learning. This could also result in error minimisation or prevention. Providing guidance could also reduce extraneous cognitive load. Guidance can bypass actions like recalling or decision-making that can increase cognitive load. Therefore, providing guidance can help novices to carry on with cognitive-demanding activities using fewer working-memory resources, for example dealing with novel math problems. Furthermore, a high level of interaction could make IDEs more appealing to novices and increase motivation. Interaction in IDEs could be used for presenting the notional machine, the notation, the input, the output, the editor and the debugger (Romero, du Boulay, Robertson, Good & Howland, 2009).

2.5.8 Further Issues Regarding the Seven Design Principles

The majority of the design principles seem to focus on and address the difficulties of novice programmers. However, these principles can have pedagogical effects as well. Examining the seven design principles from the viewpoint of constructivism and CLT, the following assertions for their pedagogical impact can be made. The clear presentation of just the necessary programming concepts and the hiding of other irrelevant information can reduce extraneous cognitive load and increase germane cognitive load. In addition, this can also be achieved by *adopting the* *linguistic attributes of users' native language for the PL and IDE* and using *less unnatural syntax* and *more understandable semantics*. Furthermore, providing an appropriate *level of abstraction of programming commands*, *a small set of instructions* and *easily comprehensible error messages* can also contribute to the management of intrinsic and the reduction of extraneous cognitive load. *Visualisation* and *a high level of interaction with the IDE* can help in the assimilation and accommodation of knowledge by engaging users in an active role throughout the process of learning to program. Users can employ these features to meet their personal learning needs in order to individually and actively construct their knowledge structures of programming concepts.

With the exception of the work of Pane and Myers (1996), McIver and Conway (1996) and Kelleher and Pausch (2005), there seems to be little research on determining principles for educational programming software that could support novices. This is evident especially when one considers the vast literature that exists on the difficulties that novices have during introductory computer programming. Although the work presented in this thesis was inspired and motivated by the work of the latter six authors, it is different and complementary to the existing body of work with respect the following two factors. First, the design principles presented here were identified not only through novices' difficulties but also from pedagogical theories, which provided the theoretical basis for well-documented educational research. Second, the set of design principles, except from having been identified based on theoretical arguments, was empirically studied as well. This was accomplished by designing, implementing and evaluating an enhanced educational programming environment, which served as a manifestation of applying the combined set of design principles (see Chapters 3 and 4).

It may be possible to identify and add more design principles to this set. However, retaining a small number of design principles, without compromising the quality and impact of this set, can be advantageous. First, a small combined set of principles can be relatively easy to implement. Second, as the number of principles increases, there has to be some trade-off between their application (Pane & Myers, 1996). Therefore, by selecting and keeping only a small number of the potentially most effective design principles, a maximum use of their effects can be achieved without the undesirable results of conflicts between them.

2.6 Educational Programming Languages and Integrated Development Environments for Novice Programmers

As discussed in the previous section, a part of this project is designing and implementing a new educational programming environment, based on the set of seven design principles. The great number of existing educational PLs and IDEs is revealed by a prominent review study by Kelleher and Pausch (2005) as well as similar studies by other authors (Deek & McHugh, 1998; Pears et al., 2007). Naturally, this project is based on and influenced by previous PLs and IDEs for novices. Therefore, this section discusses the most-cited and prominent PLs and IDEs that were designed and implemented especially for introductory programming. The PLs and IDEs are reviewed through the prism of the seven design principles. This means that their review focuses on how they comply with the seven design principles. Because the state of affairs in programming education and pedagogical programming software in Greece is of particular interest for this study, the following distinction is made. First, international (non-Greek) educational PLs and IDEs are reported, while Greek PLs and IDEs for novices follow.

2.6.1 International Educational Programming Languages and Integrated Development Environments

This subsection presents non-Greek PLs and IDEs for novice programmers. In particular, the degree and/or the type of compliance of these PLs and IDEs with the seven design principles is discussed. The selection of the programming tools to be presented was based on the following criteria. First, the programming tool should be mentioned in the international literature. Second, the target group of the software is novice programmers. Third, the tool supports, as a minimum, the basic programming concepts, regardless of the computational paradigm, for example procedural or object-oriented. Fourth, the programming software is not specialised for a specific educational aim, for instance the teaching of sorting algorithms. Fifth, these PLs and IDEs were developed after 1998 (10 years before the commencing of this study). Predating software is possibly obsolete at present due to advances in technology, and in pedagogical and psychological research.

In order to display this review more efficiently, tabular layout was chosen. The first column of Table 2.1 shows the name of the programming tool, each of the following seven columns reports compliance with each of the seven design principle and the final column presents the method(s) used for its empirical evaluation. The aim of this review is to investigate and present the level of compliance of the selected educational tools with the set of principles as well as to report the methods used to evaluate these tools. However, it was decided that the results of the reported evaluations would not be presented in Table 2.1, because (a) presenting evaluation results does not inform nor contribute to the aim of this review and (b) results from different evaluation methods cannot be used to make meaningful comparisons between tools. References for the bibliographic sources of each programming tool and its evaluation method(s) are shown in the first and last columns of the table, respectively. The degree and/or type of compliance of a particular PL and IDE with each design principle may or may not be reported in the source(s) that presents the programming tool. The latter case is reported in the table as N/R. However, the degree or type of compliance may not be explicitly reported, but it may be inferred from the description of the programming tool. In this case, the degree or type of compliance is reported within brackets. For example, the linguistic attributes of a programming tool may not be explicitly reported. However, a figure displaying the tool may be provided and hence, it can be used to infer the tool's linguistic attributes. If a design principle is not applicable to a programming tool, then this is reported as N/A.

The level of interaction between users and each IDE is reported using the following three categories: low, medium and high. The level of interaction provided by each IDE corresponds with one of the three categories regarding the interaction between the IDE and its users during the creation and execution of programs. This involves using dialogue boxes, providing information and preventing errors during program creation, controlling the execution of programs and providing feedback regarding programs' execution. The most important non-Greek educational PLs and IDEs as well as their relation to the seven design principles are presented chronologically in Table 2.1

Table 2.1
Incorporation of the Seven Design Principles by non-Greek Educational PLs and IDEs

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
FLINT (Ziegler & Crews, 1999)	[English]	N/A	Visual programming, visualisation of program execution	N/R	N/R	Syntax errors are not possible	[High]	N/R
GRAIL (McIver, 2000; McIver & Conway, 1999)	[English]	Natural-language-like syntax and semantics	N/R	[High]	Yes	N/R	N/R	Numbers of syntax and logical errors (McIver, 2000)
InSTEP (Odekirk-Hash & Zachary, 2001)	[English]	Syntax and semantics of C language	N/R	[Abstraction level of C language]	[No]	Suggestion s of possible solutions	[Medium]	Students' understanding of programs (Odekirk-Hash & Zachary, 2001)
HANDS (Pane & Myers, 2006; Pane, Myers & Miller, 2002)	[English]	Verbose natural-language-like syntax and semantics	Visual representation of difficult textual elements	[High]	[No]	[N/R]	[High]	Students' programming performance on tasks (Pane et al., 2002)

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
DrJava (Allen, Cartwright & Reis, 2003; Allen, Cartwright & Stoler, 2002; Hsia, Simpson, Smith & Cartwright, 2005; Reis & Cartwright, 2004)	[English]	Syntax and semantics of Java language	N/R	[Abstraction level of Java language]	Provides three levels regarding the size of the sets of commands	Locations of errors in the code are highlighted	[High]	N/R
Jerro (Sanders & Dorn, 2003)	[English]	Syntax and semantics of both Java and C languages	Microworld, visualisation of program execution	[High]	[Yes]	N/R	[Medium]	Students' satisfaction regarding the tool (questionnaire) (Sanders & Dorn, 2003)

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
Jeliot (Ben-Ari et al., 2011; Ben-Bassat Levy, Ben-Ari & Uronen, 2003)	[English]	Syntax and semantics of Java language	Program animation	[Abstraction level of Java language]	[No]	N/R	[High]	Paper-and-pencil programming assignments (Ben-Bassat Levy et al., 2003), prediction of programs' execution, interviews with students to measure tool's effectiveness, video record of lesson, eye movement tracking, program comprehension analysis, and teachers' views on animation and experiences (Ben-Ari et al., 2011)

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
BlueJ (Kölling, Quig, Patterson & Rosenberg, 2003)	[English]	Syntax and semantics of Java language	Visualisation of programming elements	[Abstraction level of Java language]	[No]	Explanatory error messages, lines with errors highlighted, suggestions of possible solutions	[High]	Students' views on tool's usefulness, tool's best/worst aspect and how helpful was the tool in understanding OO programming and passing the course (questionnaire) (van Haaster & Hagan, 2004)
JPie (Goldman, 2004a, 2004b)	[English]	Syntax and semantics of Java language	Visual programming	[Abstraction level of Java language]	No	Immediate but low-detailed error messages	[High]	N/R
jGRASP (Hendrix, James H. Cross & Larry, 2004)	[English]	Syntax and semantics of Java language	Visualisation of programming elements	[Abstraction level of Java language]	[No]	N/R	High	N/R

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
VIP (Virtanen, Lahtinen & Järvinen, 2005)	[English]	Syntax and semantics of C++ language	Visualisation of program execution	[Abstraction level of C++ language]	[Yes]	N/R	[High]	Students' views on tool's usefulness (questionnaire) and usage of tool (Virtanen et al., 2005)
RAPTOR (Carlisle, Wilson, Humphries & Hadfield, 2005)	[English]	Syntax and semantics of both Pascal and C languages	Visual programming, visualisation of program execution	[Abstraction level of Pascal and C languages	[No]	Syntax errors are not possible	[High]	Properties of students' programs and students' views on tool's features (questionnaire) (Carlisle et al., 2005)
ProGuide (Areias & Mendes, 2007)	Portuguese	N/A	Visual programming	N/R	Yes	Natural-language error messages, visual element highlighted	[High]	N/R
Karel J Robot (Bergin, Stehlik, Roberts & Pattis, 2005)	[English]	Syntax and semantics of Java language	Microworld, visualisation of program execution	N/R	[Yes]	N/R	N/R	N/R

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
B# (Greyling, Cilliers & Calitz, 2006)	[English]	Syntax and semantics of Pascal language	Visual programming, visualisation of program execution	[Abstraction level of Pascal language]	[Yes]	Syntax errors are not possible	[High]	N/R
ALVIS Live! (Hundhausen & Brown, 2007; Hundhausen, Farley & Brown, 2009)	[English]	Syntax and semantics of SALSA pseudo code language	Visual programming, visualisation of programming elements, visualisation of program execution	[Abstraction level of SALSA language]	[Yes]	Dynamic feedback on commands' syntactic correctness	[High]	Students' views on tool's usability (questionnaire), video analysis of program creation process, participant observation, artefact collection, properties of student's programs and time spent on task by students (Hundhausen & Brown, 2007; Hundhausen & Brown, 2008; Hundhausen et al., 2009)

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
ELP (Truong, 2007)	[English]	Syntax and semantics of Java, C and C# languages	N/R	[Abstraction of Java, C and C# languages]	[No]	Friendly error messages, links to lines with errors	[Medium]	Students' and teachers' views on tool features (questionnaire) (Truong, 2007)
Alice (Bishop-Clark, Courte, Evans & Howard, 2007)	[English]	[Natural-language-like syntax and semantics]	Visual programming, 3D microworld	[High]	[No]	Syntax errors are not possible	[High]	Students' confidence in programming, enjoyment of programming and understanding of programming concepts (questionnaire), students' reflective essays (Bishop-Clark et al., 2007)
PEN (Nishida et al., 2008)	Japanese	Constructs based on Japanese words	Visualisation of program execution	N/R	N/R	N/R	[Medium]	Students' expectations for the programming lessons (questionnaire), students' impressions (Nishida et al., 2008)

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
Turtlet (Kasurinen et al., 2008)	[English]	Syntax and semantics of Python language	Microworld, visualisation of program execution	[Abstraction level of Python language]	[No]	N/R	[Medium]	Students' views on tool's usefulness (questionnaire) (Kasurinen et al., 2008)
Programimate (Scott, 2010; Scott, Watkins & McPhee, 2008)	[English]	Syntax and semantics of Java, Visual Basic.NET and Visual Basic 6 languages	Visual programming, visualisation of program execution	[Abstraction level of Java, Visual Basic.Net and Visual Basic 6 languages]	[Yes]	Meaningful error messages	[High]	Students' views on tool's features, usability and efficacy (questionnaire) (Scott, 2010)
ViLLE (Rajala, Laakso, Kaila & Salakoski, 2008)	[English]	Syntax and semantics of Java, pseudo code and C++ languages	Visualisation of program execution	[Abstraction level of Java, pseudo code and C++ languages]	[No]	N/R	[High]	Properties of students' programs (Rajala et al., 2008)
CORT (Garner, 2009)	[English]	Syntax and semantics of Visual Basic	N/R	[Abstraction level of Visual Basic language]	[No]	N/R	[Low]	Time spent on programs and average amount of help asked by students, program understanding and program creation (Garner, 2009)

Table 2.1 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
Scratch (Resnick et al., 2009)	[English]	[Natural-language-like syntax and semantics]	Visual programming, visualisation of program execution	[High]	[No]	Errors are not possible	High	Students' experiences with the tool (questionnaire) (Malan & Leitner, 2007)
Jype (Helminen & Malmi, 2010)	[English]	[Syntax and semantics of Python language]	Visualisation of program execution	[Abstraction level of Python language]	[No]	N/R	High	N/R
Greenfoot (Henriksen & Kölling, 2004; Kölling, 2010)	[English]	Syntax and semantics of Java language	Microworld, visualisation of program execution	[Abstraction level of Java language]	[No]	No particular attention was paid to providing good error messages	[High]	No formal studies were carried out
FLIP (Good, 2011)	[English]	C-like language	Visual programming, visualisation of program execution	[Abstraction level of C language]	[No]	N/R	[High]	N/R
CodeSkulptor (Tang, Rixner & Warren, 2014)	[English]	[Syntax and semantics of Python language]	Visualisation of program execution	[Abstraction level of Python language]	Yes	N/R	[High]	N/R

Note. PLs = Programming Languages; IDEs = Integrated Development Environments; N/A = not applicable; N/R = not reported; [ ] = the type/degree of compliance is not reported, but inferred.

Table 2.1 contains the results for 27 international programming tools for educational programming. Only two of these tools are not in the English language, namely ProGuide and PEN, which are in the Portuguese and the Japanese language, respectively. The majority of the tools adopted the syntax and semantics of popular professional PLs, like Java and C/C++. Only, three tools, Alice, Scratch and PEN, implemented a more natural-like approach for the syntax and semantics of their PLs. With the exception of InStep, CORT and ELP, incorporation of visualisation is a design choice followed by all programming tools. This fact could underlie the instinctive appeal to and importance of visualisation for improving programming instruction. Its most popular form is the visualisation of program execution, followed by visual programming. The PLs of these tools support overall a high level of abstraction, but is not reported for PEN. However, the majority of the reviewed work do not explicitly mention the level of abstraction provided, probably because it is difficult to define. Provision of error messages is a design principle that is not considered or reported. However, half of the reviewed programming tools support some form of help for novices regarding error messages. All programming tools, except for CORT, appear to provide a medium to high interaction level between users and their IDE.

Overall, most of the reviewed tools do not seem to support all seven design principles and appear to have been designed without considering (at least, a high level of) compliance with the combined set of principles. From the methods that were used for the evaluation of these tools, two seem to be the most popular. The first one is using questionnaires to collect users' opinions and views regarding programming tools' features and usefulness. The second one is the analysis of programs' properties, for example correctness or output of programs created by students. The latter method can provide a measure of students' understanding of applying programming knowledge. The review of Greek educational software follows.

2.6.2 Greek Educational Programming Languages and Integrated Development Environments

The research reported in this thesis is especially concerned with the present state of computer programming education in Greece. Therefore, the available tools that support and facilitate the teaching and learning of introductory programming are very important for this study. Thus, educational software that is developed for Greek novice programmers is discussed in this section.

The criteria for selecting and the layout of presenting the educational software follow that of Section 2.6.1 and Table 2.1, respectively. One additional criterion was introduced for this review: whether the evaluation of the educational PL and/or IDE took place only in Greek educational institutes. The aim of this additional criterion is to include educational software that was developed with regard to novice programmers in Greece.

Hence, Table 2.2 shows, Greek PLs and IDEs that were designed for Greek novice programmers and how they relate to the seven principles. The software is listed in chronological order.

Table 2.2
Incorporation of the Seven Design Principles by Greek Educational PLs and IDEs

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
X-Compiler (Dagdilelis et al., 2003; Evangelidis et al., 2001)	[English PL - Greek IDE]	Greek Pascal-like language	Visualisation of program execution	[Abstraction level of Pascal language]	Yes	Errors message in natural language, suggestions of possible solutions	[Medium]	N/R
AnimPascal (Satratzemi et al., 2001)	[English]	Syntax and semantics of Pascal language	Visualisation of program execution	[Abstraction level of Pascal language]	N/R	Line with error highlighted	[Medium]	Properties of student's programs (Satratzemi et al., 2001)

Table 2.2 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
objectKarel (Satratzemi et al., 2003; Xinogalos et al., 2006)	[English]	Syntax and semantics of Karel++ language	Microworld, visualisation of program execution	[High]	Yes	Line with error reported, errors message in natural language	High	Properties of students' programs (Satratzemi et al., 2003); Students' difficulties, properties of students' programs and students' views on tool's usability and educational effectiveness (questionnaire) (Xinogalos et al., 2006)

Table 2.2 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
WIPE (Efopoulos, Dagdilelis, et al., 2005; Efopoulos, Evangelidis, et al., 2005)	[English]	Syntax and semantics of Pascal language	Visualisation of program execution	[Abstraction level of Pascal language]	Yes	Error messages in natural language, suggestions for possible solutions	[Medium]	Classroom observations, properties of students' programs, comparative evaluation with Borland Pascal, interviews with teachers and students' views on tool's features (questionnaire) (Efopoulos, Dagdilelis, et al., 2005; Efopoulos, Evangelidis, et al., 2005)
DAVE (Vrachnos & Jimoyiannis, 2008)	[Greek]	[Greek Pascal-like language]	Algorithm visualisation	[High]	[Yes]	N/R	[Medium]	N/R

Table 2.2 (continued)

Name	Design Principle							Evaluation method
	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE	
LECGO (Kordaki, 2009, 2010)	[English PL - Greek IDE]	Syntax and semantics of C language	Graphical Output: Geometrical shapes	[Abstraction level of C language]	N/R	N/R	[High]	Properties of students' programs, students' difficulties and questions, and graphic functions used (Kordaki, 2010)

Note. PLs = Programming Languages; IDEs = Integrated Development Environments; N/A = not applicable; N/R = not reported; [ ] = the type/degree of compliance is not reported, but inferred.

The results presented in Table 2.2 reveal that only six programming tools have been developed for introductory programming in Greece and are reported in the international literature. Furthermore, the majority of these PLs and IDEs present a low level of compliance with the set of design principles. Users' natural language, syntax and semantics, level of abstraction and small set of commands appear not to have been considered in the design of those PLs. This is the case for the linguistic attributes of and the level of interaction with IDEs as well. Only three out of six tools support error messages in natural language. However, visualisation features seem to have been incorporated in all PLs and IDEs.

As in the case of international educational software, most of the tools were evaluated using questionnaires that measured tools' usability and properties of students' programs, such as correctness and number of missing or misplaced commands. Furthermore, with the exception of WIPE (Efopoulos, Dagdilelis, et al., 2005; Efopoulos, Evangelidis, et al., 2005) and LECGO (Kordaki, 2010), the effectiveness of each tool on the programming performance of students was not evaluated in comparison with the effectiveness of other tools (McIver, 2002; Vogts, Calitz & Greyling, 2008).

2.6.3 Overview of Surveying Educational Programming Languages and Integrated Development Environments

The overall conclusions of reviewing international and Greek PLs and IDEs for novices are the following. A considerable portion of the PLs and IDEs were designed in the English language. Furthermore, a great number of the analysed educational programming tools incorporated the unfriendly syntax and semantics of professional PLs. Visualisation was integrated in various forms by the majority of

PLs and IDEs. The level of abstraction of PLs was almost never explicitly reported and probably not considered, while the design principle of providing understandable and helpful error messages was often neglected. Most of IDEs support a medium to high level of interaction with their users.

Finally, the evaluation methods that were employed were based on students' views on tools' usability and properties of students' programs. However, the level and quality of programming knowledge that students actually acquired by using the specific tool cannot be fully assessed by either method (see Section 2.7 for a discussion of how to measure students' programming knowledge). Moreover, most of programming tools were not evaluated by comparing their impact on students with the impact of other programming tools (Vogts et al., 2008). This comparison, however, is essential, because comparing a new programming tool with a 'traditional' one can provide evidence for its effectiveness and establish its superiority.

In order to overcome the weaknesses of the reported evaluation methods, a more in-depth method, which involves knowledge measurement, can be employed for the evaluation of programming tools. Therefore, the following section discusses declarative and procedural programming knowledge, how it can be elicited and measured, and its relation to this thesis.

2.7 Knowledge and Mental Models

In Section 2.3, the learning objectives of programming instruction were reviewed. Two underlying ideas were identified as common amongst the different learning objectives. First, learning programming involves the acquisition of knowledge regarding programming concepts. Second, learning programming involves the development of knowledge on how to use these concepts in program creation.

Cognitive psychology defines these types of knowledge as declarative and procedural knowledge, respectively. As mentioned in Section 2.3, this thesis considers learning to program as the acquisition of declarative knowledge and the development of procedural knowledge in the programming domain (Palumbo, 1990; Shaffer et al., 2003). Hence, the first part of this section discusses declarative and procedural knowledge in programming. The second part discusses mental models as a method for measuring declarative knowledge.

2.7.1 Declarative and Procedural Knowledge in the Programming Domain

Cognitive psychology defines declarative or conceptual knowledge as the knowledge about constructs, facts, principles and their interrelations (de Jong & Ferguson-Hessler, 1996). Declarative knowledge is hypothesised to be stored in the form of schemata (Ormerod, 1990) or propositional networks (McGill & Volet, 1997). This type of knowledge is consciously accessible and processed, and can be communicated through speech (Robillard, 1999). In the programming domain, declarative knowledge refers to knowledge of programming concepts, syntax and semantics of PLs (Palumbo, 1990).

Procedural knowledge is defined as knowing how to effectively use, if it is applicable, the concepts of declarative knowledge and especially in problem-solving situations (de Jong & Ferguson-Hessler, 1996). Procedural knowledge "is usually associated with very efficient automatic behaviours" (Harvey & Anderson, 1996, p. 71) and is often represented by production rules (Ormerod, 1990). A production rule has the form of an if-then statement, namely IF <condition> THEN <action>. In the case that a condition is validated to be true, then the related action is triggered. Actions can be either physical activities or mental processes. When a production rule is mastered within a domain, it can be applied to other but similar domains (Harvey & Anderson, 1996). Procedural knowledge is hard to describe verbally and the methods of applying it usually cannot be forgotten (Robillard, 1999) . The development of procedural programming knowledge seems to be more cognitively challenging than of declarative programming knowledge (Palumbo, 1990). Moreover, expertise seems to be more associated with procedural knowledge than declarative knowledge (Rose, Rose & McKay, 2007). Specifically in programming, procedural knowledge enables one to use programming concepts, syntax and semantics (declarative programming knowledge) to create programs. Hundhausen et al. (2002) also noted that visualisation in programming instruction seems to affect more procedural than declarative programming knowledge.

A challenging question that is still unanswered is which type of knowledge is acquired first. Anderson (1982) postulated that programming knowledge is first acquired as declarative knowledge. This knowledge is initially used through general procedural rules that are applicable in the programming domain. This way declarative knowledge is transformed into production rules, which are the basis for procedural knowledge acquisition. This transformation is called proceduralisation. Through practice, the application of procedural knowledge becomes more spontaneous and less cognitively demanding. Obtaining experience in the domain leads to improvements in the quality of declarative and procedural knowledge, and their use becomes faster and more effective. However, McGill and Volet (1997) reported that there is evidence that a certain level of procedural knowledge can be acquired just by mere observation and can be applied without extensive possession of declarative knowledge. Furthermore, based on other sources, they noted that fundamental procedural knowledge can even enable the acquisition of complex declarative knowledge. This finding seems to contradict hypothesis H3 proposed in this thesis and according to which the effect of programming tool on procedural knowledge (programming performance) is mediated by declarative knowledge (mental-model quality). However, Anderson's (1982) postulation – that declarative knowledge is acquired first – and the mental-model hypothesis (Kellog & Breen, 1990) – the richer one's mental-model quality becomes in a domain, the better one performs in the particular domain – support the idea that mental-model quality in the programming domain can affect programming performance and more specifically, as the former improves so does the latter. Therefore, the testing of hypothesis H3 is especially interesting from a research viewpoint.

Not surprisingly, declarative and procedural programming knowledge have been used in various topics of CSE studies, with the following findings. Procedural knowledge in the programming domain can be applicable to several PLs as long as they use the same programming paradigm, for instance procedural programming (de Raadt, 2008). This is the case for declarative knowledge regarding semantics and programming constructs as well (Harvey & Anderson, 1996). However, declarative knowledge of syntax seems to be related to a particular PL (Shneiderman & Mayer, 1979). Wiedenbeck (1985) noted that novice programmers tend to rely more on declarative knowledge than procedural knowledge. As they progress to become experts, they rely less and less on declarative knowledge and procedural knowledge becomes more significant. A similar finding is presented by Harvey and Anderson (1996), who reported that once declarative knowledge has become "proceduralized, there is no further need for declarative instruction" (Harvey & Anderson, 1996, p. 93). Van Merriënboer (1990a) proposed some methods for developing declarative and procedural instruction in programming. These methods include the teaching of concrete models, program plans and worked-out examples. McGill and Volet (1997) studied how declarative and procedural programming knowledge are related to syntactic and conceptual knowledge. Syntactic and conceptual knowledge are used in the educational computing literature, and refer to facts and rules of PLs and to the comprehension of programming concepts and principles, respectively. They concluded that the set of declarative and procedural knowledge describes a different type of knowledge from the set of syntactic and conceptual knowledge, and the two sets of knowledge do not overlap. Furthermore, they attested that the sole distinction between declarative and procedural knowledge is adequate for CSE research. Declarative and procedural knowledge, although they can be distinguished, are hardly isolated from each other in practice. The nature of programming knowledge that novice programmers have was presented in Section 2.4.7.

There is also a third type of knowledge with respect to knowing the conditions under which declarative and procedural knowledge should be used. This type of knowledge is called conditional knowledge and is assumed to be general, not related to specific context, acquired after much experience and hierarchically above declarative and procedural knowledge. Conditional knowledge is used for tackling problem-solving situations, directing actions when information is missing, and provide diagnosis and alternatives when errors occur (McGill & Volet, 1997). However, this thesis is not concerned with this type of knowledge; due to its metacognitive characteristics, it offers no useful assessment of novices’ performance within this project’s context.

The scarce use of in-depth measures, such as declarative and procedural programming knowledge, for evaluating PLs and IDEs for introductory programming instruction is evident not only from this section, but also from Sections 2.6.1 and 2.6.2. As proposed by Ben-Ari, “techniques, which elicit the internal structures of the student, are far more helpful than research that measures performance alone” (Ben-Ari, 2001, p. 67). Therefore, measuring declarative and procedural programming knowledge can provide a more objective and meaningful way to study the impact of educational PLs and IDEs. How these two types of knowledge can be measured is discussed in the following section.

2.7.2 Measurement of Declarative and Procedural Knowledge

In order to measure knowledge, it first must be elicited. In turn, eliciting knowledge requires a fundamental awareness of its form and structure. A proposal for representing the structure of already acquired knowledge is the theory of mental models (Carley & Palmquist, 1992).

2.7.2.1 Mental models

The notion of mental models was introduced by Craik (1943). He described mental models as small-scale representations for internalising objects of the external world. These structures are used for explaining, reasoning and predicting our interaction with these objects. The term was revisited in 1983 by Philip Johnson-Laird as well as by Dedre Gentner and Albert Stevens (George, 2000). Johnson-Laird (1983) argued that mental models are a mechanism that humans employ in order to perform cognitive activities, such as understanding, reasoning and decision-making. According to him, mental models are analogical and static representations of situations, which reflect the situations’ structure. He noted that the closer the structure of a situation and the structure of its mental model are, the more useful the mental model becomes. The papers collected by Gentner and Stevens (1983) viewed mental models as a means of comprehending, explaining and interacting with complex physical systems and technological contraptions (Greca & Moreira, 2000). Norman (1983) postulated that mental models can convey one's belief about the system they describe, are influenced by the observable states of the system, and provide understanding and prediction of a system's behaviour. However, mental models are described as incomplete, unstable, implicit, without rigid boundaries, unscientific and subjective to time-delayed modifications (Ma, 2007). Mental models are a way of coping with complex situations or systems, which include overwhelming details and their processing involves managing significant intrinsic cognitive load. Thus, mental models may be formed as described above as an attempt to balance between the high demands in cognitive resources, and the successful explanation and prediction of interaction with the situation or system. However, due to their dynamic nature they can be modified through instruction (Bayman & Mayer, 1984) or interaction with systems (Norman, 1983). Furthermore, Bayman and Mayer (1984) noted that individual users build different mental models in spite of going through similar hands-on instruction. This can be explained by another function of mental models, which is to provide a framework for incorporating new knowledge (Darabi et al., 2009; Gentner & Stevens, 1983; Jih & Reeves, 1992; Ma, 2007). This function is also consistent with the constructivist theory that postulates that new knowledge is acquired and organised based on previous knowledge. Mental models also help individuals to successfully deal with novel and problem-solving situations and mentally accessing invisible states of systems (Bayman & Mayer, 1984).

The work of Johnson-Laird, and Gentner and Stevens had a strong impact on the field. However, the notion of mental models is until today vague and this lack of a clear definition is a source of criticism (Johnson-Laird, 1989). More recent definitions have described mental models as ‘‘a schema plus cognitive processes for manipulating and modifying the knowledge stored in a schema” (Merrill, 2002, p. 278), ‘‘a rich mental representation of how a task domain is organized” (van Merriënboer & Kirschner, 2007, p. 285), “ad hoc cognitive constructions of relevant elements within the domain to address a present problem” (Darabi et al., 2009, p. 724) and “a mental model is generally knowledge about the task or the system that facilitates understanding, explanation and prediction” (Cooke & Rowe, 1997, p. 185).

Despite a lack of agreement on their definition, mental models have been used for measuring knowledge acquisition in a variety of situations. These include avionics troubleshooting, NASA mission control tasks, personal relationships (Cooke & Rowe, 1997), the use of electronic calculators (Young, 1981) and chemical-engineering instruction (Darabi et al., 2009).

Similarly, mental-model research has been applied in studies of computer programming. Robins et al. (2003) underlined the fact that novice programmers must hold a number of different mental models. These include the mental model of the problem domain, the mental models of programming constructs, the mental model of the intended program, the mental model of the program that is actually written, the mental model of programs’ execution and the notional machine. Corritore and Wiedenbeck (1999) used mental models to investigate the differences between procedural and object-oriented experts. Bayman and Mayer (1983) associated learning to program with building correct mental models of programming concepts. Ma et al. (2008; 2009) reported that novices hold incorrect mental models of basic programming constructs, even after the completion of an introductory programming course. The authors used visualisation and cognitive conflict to improve the incorrect mental models with important, but possibly short-term effects. Scott, Watkins and McPhee (2008) used flowchart visualisation in a programming IDE to aid students establish accurate mental models of program execution. They presented preliminary results that considered the IDE beneficial, but without any further details about the effect of flowchart visualisation on students' mental model of execution. George (2000) studied the use of diagrammatic traces in recursion. Without the use of diagrammatic traces students tended to think of recursion as iteration, while explicit use of diagrammatic traces helped in building a correct mental model of recursion. Ramalingam et al. (2004) reported that mental models of students positively influenced self-efficacy and, in turn, both positively influenced programming performance. Cañas et al. (1994) investigated the effect of visually tracing the execution of programs on students' mental models. They reported that students who used the visualisation technique possessed mental models of programming concepts based on the semantic characteristics of these concepts, while students who did not use the visualisation techniques organised the concepts based on their syntactic characteristics. However, no significant differences were observed between the two groups in their programming performance. The authors noted "a possible disassociation between the acquisition and understanding of conceptual-declarative knowledge and the use of this knowledge to perform a procedural task" (Cañas et al., 1994, p. 809). They postulated that this phenomenon occurred because, at the very first stages of learning, it is possible that the formation of mental models is influenced by learning conditions, for example the use of the visualisation technique. However, after this stage, mental models do not further change until a significant improvement has been made in procedural knowledge. Moreover, their results supported the idea that, at the first stages of learning, the quality of mental models does not seem to significantly affect performance. This explanation can be further supported by the observation that mental models' modification requires time and by McGill and Volet' s (1997) remark that procedural knowledge can be correctly used before a proper development of declarative knowledge. The apparent inconsistency between this finding and (mediation) hypothesis H3 was discussed in Section 2.7.1.

Evidently, mental models can be a useful tool in computer-programming research. However, it has sporadically been used as an evaluative measure of PLs and IDEs. Thus, it is not only helpful, but also interesting, from a research viewpoint, to include mental models in this study. Because of lacking a formal definition, it is important to state what mental models mean in the context of this study. Hence, in this project, mental models are viewed as dynamic knowledge structures, which hold information that includes declarative knowledge (Banks & Millward, 2007). Therefore, from this point forward, mental models are considered as representations of declarative knowledge.

In order to elicit mental models, various techniques have been proposed. These include concept-elicitation methods, data-collection methods and structural analysis. The selection of a technique is made according to its suitability for eliciting a particular type of knowledge (Cooke, 1994; Marhan, Micle, Popa & Preda, 2012). Carley and Palmquist (1992) proposed that "mental models can be represented as networks of concepts" and "the meaning of a concept for an individual is embedded in its relations to other concepts in the individual's mental model" (Carley &

Palmquist, 1992, p. 602). Furthermore, expert performance in a scientific domain seems to be associated more with the structure and interrelations of the acquired concepts (mental model) of the specific domain rather than the simple acquisition of domain-specific concepts (Rose et al., 2007). This is also true in the case of programming (Lau, 2009). Therefore, the structural analysis of mental models seems to be the most appropriate technique for eliciting mental models, at least for the purposes of this study. According to Cooke (1994), structural analysis can be performed using multidimensional scaling, clustering routines, network scaling, free association tasks, question-answering protocols, graph construction tasks and concept maps. Amongst these methods, the network scaling technique and specifically the Pathfinder Scaling Algorithm seems to produce the most desired outcomes for this study (for more details see Section 2.7.2.2).

2.7.2.2 The measurement of declarative knowledge through mental models – Pathfinder Scaling Algorithm

The network scaling method produces a graphical representation of concepts' relation in the form of a network. The strength of a relation between two concepts is depicted by the spatial distance between them. In order to analyse the structure between a set of concepts, the Pathfinder Scaling Algorithm (PSA) (Schvaneveldt, 1990) takes as input a rating for each pair (pair rating) of concepts. Therefore, for a set of *n* concepts, the number of pair ratings is *n* x (*n* - 1) / 2. Each rating denotes how closely two particular concepts are related. PSA analyses the ratings for all possible pairs of concepts and produces a non-hierarchical network of concepts. The distance between the concepts represents the relation between the concepts. The closer two concepts are, the stronger their relations is. These networks are called PFNETs and can serve as a good representation of mental models (Goldsmith, Johnson & Acton, 1991). An example of a PFNET can be seen in Figure 2.1.

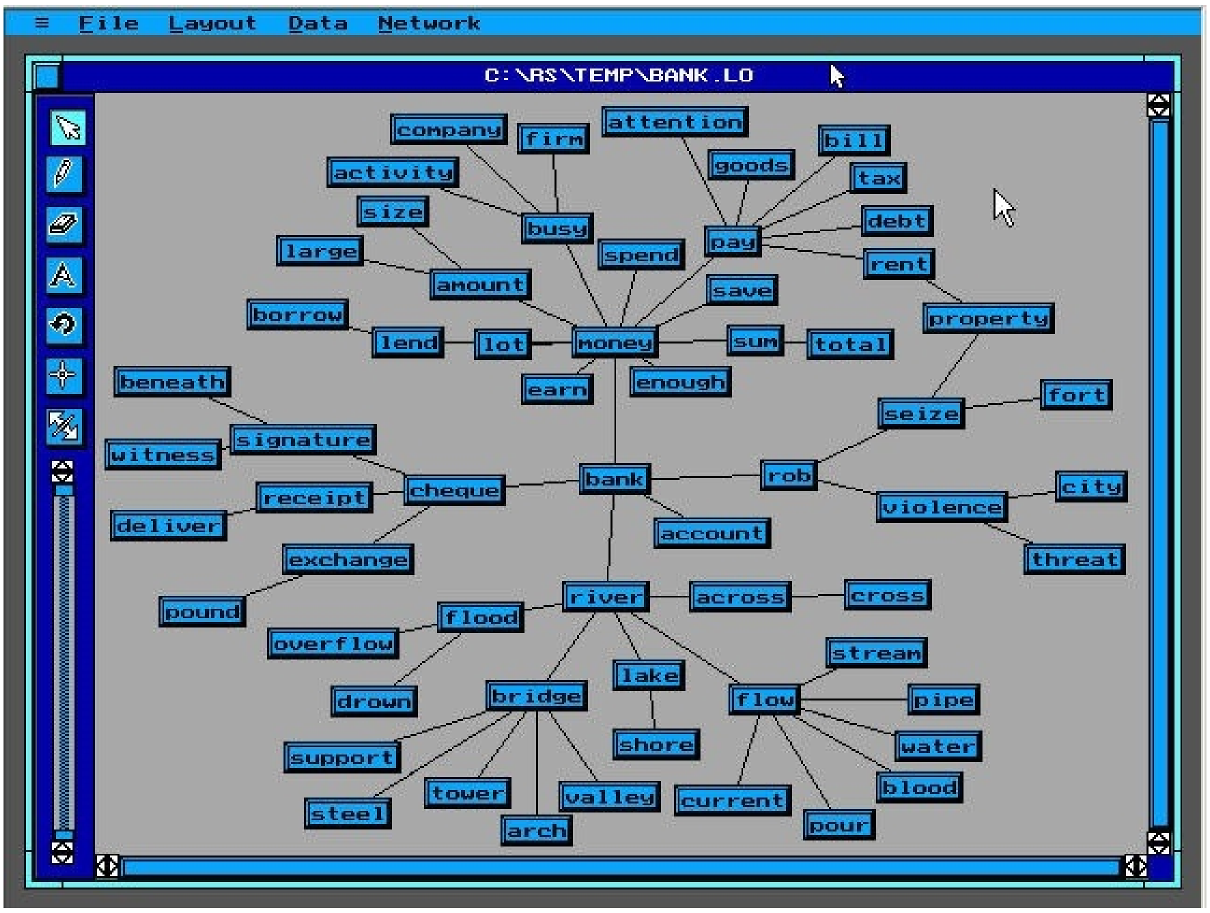


Figure 2.1. Example of a PFNET (retrieved from the Interlink website: http://interlinkinc.net/KNOT.html).

The use of PSA in structural analysis of mental models is preferred over other available techniques for the following reasons. First and foremost, PSA can accept an expert's PFNET and use it as a baseline. Hence, the closeness between participants' PFNET and an expert's PFNET can be estimated and an objective assessment of knowledge structure can be obtained. The closeness of two PFNETs is denoted by the similarity index and ranges from 0 (minimum) to 1(maximum) (Goldsmith & Johnson, 1990). Second, the similarity between an expert's and novices' PFNETs can be used to test the mental-model hypothesis. That is, the higher quality of an individual’s mental model of a domain, the better the individual’s task performance in the particular domain (Kellog & Breen, 1990). Third, PSA seems to represent concepts interrelations more meaningfully and be more predictive than multidimensional scaling (Cooke, 1994). Fourth, PSA can represent non-hierarchical networks, which clustering algorithms cannot (Cooke, 1994). Fifth, PSA provides more effective and faster data collection and data analysis procedures from participants than questioning-answering protocols and graph construction tasks. Finally, PSA can estimate the consistency of ratings of pairs for a set of concepts and determine how carefully the rating task was done or could be performed by raters. This measure is denoted by the coherence index. Values of coherence above 0.20 are considered indicative of a carefully performed rating task.

The assessment of structural knowledge and mental models with the Pathfinder algorithm has been used in various studies. Goldsmith and Johnson (1990) discussed and empirically studied the applicability of structural analysis in classroom learning. They used raw proximities, multidimensional scaling distances, Pathfinder graph-theoretic distances and closeness of PFNETs to compare students’ and instructor’s representations of statistics. They concluded that between the four measures, Similarity of PFNETs was the best predictor of course performance. Furthermore, they noted that as students advanced during their statistics course, their knowledge representations changed and became similar to instructor’s representation. Gillan, Breedin and Cooke (1992) used PSA to compare declarative-knowledge representations between a group of HCI-human factors experts and a group of HCI-software development experts. A third group of HCI non-experts was used as a control group. Cooke and Rowe (1997) used the network similarity index that was provided by PSA to measure mental models in the domains of mission control tasks, avionics troubleshooting and personal relationships. Rose et al. (2007) studied the development of knowledge structures of novice accountants during instruction and use of a decision aid. They compared novices' with experts' networks, obtained by PSA, in order to monitor the development of knowledge structures. They noted that PSA successfully measured changes in knowledge structures as these developed. Serrano, Quirin, Botia and Cordón (2009) used PFNETs to simplify the representation of multi-agent systems. They reported that PFNETs representations facilitated the understanding and debugging of these systems. Chen (2011) incorporated PFNETs in a personalised diagnosis and remedial learning system. The system compared the PFNETs of an expert and a novice user. Based on the differences between the two PFNETs novice's misconceptions were identified and reported.

Despite the advantages that PSA offers, this method has been used only in a small number of computer-programming studies. Cooke and Schvaneveldt (1988) studied how four different groups of programmers cognitively organised 16 programming concepts. The four groups were defined as naive, novice, intermediate and advanced and differed in their programming experience, which ranged from zero to one, from one to two, from two to three and three or more years of experience, respectively. The authors used PSA to produce the PFNETs of participants as well as an average PFNET per group. Their findings were that as experience decreased (a) more concepts were incorrectly defined and (b) within-group consensus on the semantic relation between concepts decreased as well. Trumpower and Goldsmith (2004) studied the effect of structural aids and specifically interactive overviews on structural knowledge of computer-programming concepts. They used the definitions of 12 programming concepts to create three interaction overviews. The first was structured alphabetically, the second randomly and the third according the knowledge structure of experts. Each structured interaction overview was presented to a group of students. The knowledge acquired by each group was evaluated by eliciting students' knowledge structures using PSA and performing a procedural-knowledge task, namely predicting the output of a sorting program. Their results revealed that the group which was presented with the experts' knowledge structure produced better knowledge structures and performed better in the task than the other two groups. This result was attributed to the visible structure of the interaction overview. These findings support the idea that the structure of experts' knowledge can be used to help novices in achieving expertise in a domain. Furthermore, it may be interesting and useful from a pedagogical viewpoint that this structure together with the underlying information can be conveyed visually from experts to novices. Lau and Yuen (2010) studied the effect of gender and learning style on mental models of sorting-algorithm instruction. Additionally, Lau and Yuen (2011) studied the effect of gender, learning style, mental models, prior composite academic ability and medium of instruction on programming performance. In both cases they employed PSA to evaluate the mental models of participants. In the first (2010) study they reported that the mental models of females and concrete learners were more similar to one expert's mental model than the mental models of men and abstract learners, respectively. In the second (2011) study, their findings supported the idea that programming performance can be influenced by the quality of mental models – this is predicted by mental-model hypothesis (Kellog & Breen, 1990) –, learning style, prior academic ability and medium of instruction, but not by gender. However, they reported that gender can affect the quality of mental models. In particular, females' mental models were more similar to the expert's mental model than males' mental models in the second study as well. They speculated that this possibly occurred because the expert was a female and suggested further investigation. Furthermore, they provided evidence for the idea that the more similar a student's and an expert's mental model get, the better the student's performance becomes. Exceptionally, Kahler (2002) used PSA and backward thinking to measure procedural programming knowledge. She reported a significant positive correlation between similarity index and project score, but only in one of the three measurements. However, Kahler's approach of employing PSA as a measure of procedural knowledge has not been repeated until now, at least to my knowledge. This can be attributed to lack of strong evidence that PSA can indeed represent procedural knowledge appropriately.

2.7.2.3 Measurement of procedural knowledge

Measuring procedural programming knowledge is more straightforward than declarative programming knowledge. This is true at least at the level that this project is concerned with.

Procedural knowledge, the ability to select and follow a series of proper actions to achieve a specific goal can be measured in a deterministic decision environment by posing a problem and asking for the rules or steps to successfully solve this problem (Rose et al., 2007). The aim of developing programming skills is to learn how to correctly construct a program that addresses a particular problem. This involves using all programming concepts one knows or a subset to create a feasible and valid programming solution to the particular program. Thus, the programming skills of students are considered as their procedural knowledge of the domain of programming.

In order to assess students’ procedural programming knowledge specially designed items can be used. In order to respond to these items students need to use their procedural knowledge. These items can include prediction of programs’ output, filling in missing commands, modifying existing programs and creating new ones. In order to measure procedural programming knowledge, Lau and Yuen (2011) developed tasks of predicting the output of a program and filling in missing commands of programs. Similarly, prediction tasks were administered by Trumpower and Goldsmith (2004) as procedural-test items. Kurland et al. (1989) used scores of specifically-constructed and typical class-tests to measure programming performance.

2.8 Conclusions

After the introduction, the first section of this chapter presented two popular pedagogical theories that have influenced the teaching and learning of programming, namely constructivism and CLT. The second section presented the various views on the learning outcomes of introductory programming instruction. The third section reviewed the difficulties that novice programmers face during their introductory programming course. Based on these sections, a set of seven design principles for educational programming software was identified and discussed in the fourth section. These design principles are (a) match of users’ natural language with the linguistic attributes of the PL and IDE, (b) syntax and semantics of instructions supported by the PL, (c) visualisation, (d) abstraction of commands provided by the PL, (e) a small set of instructions, (f) provision of error messages and (g) a high level of interaction with the IDE. The fifth section examined international and Greek educational programming tools and their relation to the set of design principles. Finally, the sixth section was concerned with aspects of cognitive psychology about knowledge and its measurement in the programming domain. In particular, declarative and procedural programming knowledge, mental models in the domain of introductory programming and the Pathfinder scaling algorithm, an algorithm for assessing structural knowledge, were presented and discussed.

From the review of the literature, the following gaps in knowledge are identified. First, constructivism and CLT, have not been used as a combined theoretical background for the design of programming tools for novices. Second, the impact of a set of design principles for educational programming software has rarely been empirically studied. Third, declarative knowledge, in combination with procedural knowledge, may reveal a more in-depth knowledge of concepts. However, the impact of programming tools on declarative and procedural programming knowledge that novices acquire by their use is almost never examined. Fourth, the effectiveness of a new educational programming tool has scarcely been evaluated by comparing it to the effectiveness of educational software already used for programming instruction. Fifth, linguistic attributes of PLs and IDEs are a very important factor in the design of programming tools. However, only a small number of evaluations regarding introductory programming tools with Greek linguistic attributes for Greek novices are reported in the international literature and the available tools have various shortcomings in terms of compliance with the set of identified design principles. Sixth, the effect of the seven design principles on the mental models and programming skills of novices has not been studied in depth.

This project aims to contribute to knowledge by addressing these gaps through the pursuit of three aims. The first aim, which was accomplished and discussed in Section 2.5, was to identify a set of scientific principles based on constructivism and CLT, as well as on novices' difficulties, for the design of introductory programming tools. The second and third aims are:

- to design and implement a new Greek programming tool as an embodiment of this set of scientific principles for introductory programming courses in Greece;
- to evaluate the new programming tool and assess its impact on the programming performance of novices through quasi-experimental research and, via this evaluation, test hypotheses H1, H2 and H3.

Testing hypotheses H1, H2 and H3 involves the investigation of the effect of compliance with this set on the quality of novices' mental models and programming skills as well as to explore a potential mediation effect of the novices' mental-model quality on their programming skills. The following chapter describes the design and implementation of a new Greek programming tool. This new tool is called Koios and it was developed as a manifestation of the application of the seven design principles.

Chapter Three

The Koios Programming Environment

Contents

3.1 Overview

The third chapter of this thesis presents the new programming tool that was developed as a manifestation of the application of the combined set of seven design principles (see Section 2.5). This programming tool is called Koios and was named after a titan from Greek mythology, who is related to the concept of intelligence and the quest for knowledge. The second section of this chapter discusses the choices regarding the design of Koios, while the third section discusses the development of Koios. This is followed by a comparison of two versions of Koios that were developed. Due to the nature of the research project that is reported in this thesis, the design and implementation of Koios is primarily presented from an educational point of view.

3.2 Design of Koios

The following section describes the design process of Koios. The set of the seven design principles provided the groundwork for this process. Naturally, the design process was influenced by previous studies relevant to this subject as well. Thus, a number of design choices were inspired by or based on recommendations that were proposed in the work of Pane and Myers (1996), McIver (2000), Kelleher and Pausch (2005) and Mannila and de Raadt (2006). The design with respect to the set of the seven design principles is discussed first. Important design choices beyond this set of principles follow.

3.2.1 Compliance with the Seven Design Principles

As discussed in Chapter Two (see Sections 2.5.8 and 2.8), the set of the seven design principles formed the cornerstone of Koios' design. Achieving a high level of compliance between Koios and the set of the design principles was the fundamental guideline during the design process. The results of following this guideline follow.

3.2.1.1 Linguistic attributes of Koios (first design principle).

The prototype of Koios was designed for novice programmers who study in Greek educational institutes. Therefore, according to the first design principle (see Section 2.5.1) Koios was developed in the *Greek language*.

3.2.1.2 Koios' set of commands (fifth design principle).

Determining the concepts that should constitute the curriculum of an introductory programming course continues to be a topic for research. However, this topic is beyond the scope of this project. Therefore, the concepts that would be supported by Koios were determined by (a) the fifth design principle (see Section 2.5.5), (b) the target users of Koios and (c) CS curricula of educational institutes. Following the fifth design principle, Koios should incorporate a *small set of programming commands* (see Section 2.5.5). The target audience of Koios is novice programmers and thus all the basic programming concepts should be supported. Moreover, because the evaluation of Koios was going to take place in Greek educational institutes, the curricula of these institutes should also be considered. The minimum common set of concepts that emerged from a review of these curricula are: *constant*, *variable*, *array*, *arithmetic operators*, *relational operators*, *logical operators*, *input-statement*, *output-statement*, *assignment-statement*, *conditional statement, iteration-statement*, *function* and *procedure*. These concepts are also included in the majority of introductory modules in programming in secondary and tertiary education across the world. The adoption of this set of programming constructs by this study does not present any drawbacks, because the effect of principles under investigation is in principle not affected by the programming concepts included in a curriculum. Furthermore, this set of constructs facilitated the evaluation and data-collection process in Greek educational institutes, because it did not interfere with their CS programme of studies.

3.2.1.3 Program creation with Koios (third, sixth and seventh design principles).

In order to allow novices to focus on the programming concepts during program creation, it was decided that the cognitive load imposed by the syntax of commands should be minimised (Gaspar et al., 2008; Grandell et al., 2005; McIver, 2000; Myers, 1990; Scott, 2010). Thus, the users of Koios can *create* their *programs* only *in a graphical/visual way* (Petre et al., 1998), as specified by the third design principle – *visualisation* (see Section 2.5.3). Visual programming is a rather inefficient method for creating large programs (see Section 2.5.3). However, the programs that are typically created in an introductory course are usually small. Therefore, creating small programs by means of visual programming can be a powerful method for program creation, especially for programming tools for novices. A list of icons represents the available programming commands. The choice was made to represent commands in the form of images instead of text because images seem to facilitate the construction of mental models and schemas (Tudoreanu & Kraemer, 2008). A programming command can be added into a program by dragging-and-dropping or copying-and-pasting its icon into the program (Lee & Ko, 2011).

Dropping or pasting a command's icon in programs initiates a dialogue session between users and Koios via dialogue boxes. During this process, the user is guided through the creation of a command and all the necessary parameters regarding it are requested from users. Thus, the detection of syntax errors during the creation of each command can be facilitated and immediate feedback can be returned to users (Grandell et al., 2005) via error messages. Following the sixth design principle – *provision of error messages* (see Section 2.5.6), these *error messages* were designed to *provide comprehensible descriptions* of errors *and, whenever possible, directions for correcting* them. Hence, the functionality of an interpreter is simulated, which is considered pedagogically beneficial for novice programmers (Gaspar et al., 2008). Moreover, this process can sustain a *high level of interaction* between Koios and users, as defined by the seventh design principle (see Section 2.5.7). Most important, following this process, a command is added in the program only when its parameters are specified correctly, which guarantees that programs are always without syntax errors and can be executed at any time. This process could also reduce extraneous cognitive load (see Section 2.2.2.1), because users do not need to recall the syntactic rules and required parameters for correctly creating a command (Gaspar et al., 2008; Scott, 2010). Moreover, novices can be guided by Koios through the creation of each command, focusing on the programming concepts and their functionality.

3.2.1.4 Notations supported by Koios.

However, a basic parameter that was always considered throughout the design process was that Koios should be a programming tool for the initial stages of programming instruction and therefore, should provide features that would make the transition to PLs for professionals as smooth as possible. For this reason, programs can be viewed in a graphical or a textual way (Pane et al., 2001; Petre et al., 1998). Furthermore, this approach underlies the distinction between designing a program and writing code (van Merriënboer, 1990a). The textual view provides a ‘traditional’ textual code that follows a specific set of syntactic and semantic rules, and is created automatically by the graphical representation of the program. For more formal and technical details regarding the syntax and the grammar of the language that is supported by Koios see Appendices B.2 and B.3 .

3.2.1.5 Syntax and semantics of commands supported by Koios (second design principle).

The syntax and semantics of the programming commands were designed according to the second design principle, which suggests that natural syntax and clear semantics of commands should be supported by PLs (see Section 2.5.2). I attempted to produce a *syntax* that feels *natural* (at least, in the Greek language), while it conforms to a certain level of formality. Because the textual form of programs is created automatically and syntactic rules do not need to be remembered, the phrasing of commands in textual view can be verbose and be more descriptive than typical programming commands. This, in turn, can make the textual form of programs more readable and informative. Likewise, the semantics of the commands were designed to convey their programming meaning and to be as clear as possible. To accomplish this aim, the auto-generated code follows conventional rules of indentation and different combination of colours and font styles are used to distinguish different programming constructs (see Section 3.3.3 and Table 3.3). Thus, understanding each command's function in a program is facilitated and novices are familiarised with the formal structure that they will possibly encounter when moving on to a commercial PL.

3.2.1.6 Abstraction level of commands supported by Koios (fourth design principle).

Furthermore, considering the fourth design principle (see Section 2.5.4), special care was taken so that commands provide the *appropriate level of abstraction* for novices. This level should allow programmers to understand the basic function of each command and its properties, while hiding unnecessary technical details.

3.2.1.7 Program execution with Koios (third and seventh design principles).

Programs are executed in a new window, which shows these in textual form. The textual view of programs during execution was preferred over the graphical view because otherwise the textual representation of programs could be neglected. Users of Koios create their programs in a graphical way and, thus, they are required to pay much attention to this form of programming. If the textual form of programs is not presented during the process of executing a program, then it is possible that users could pay little attention to this form. However, at present, the dominant way of creating programs is by using text and a visual representation of large programs is inefficient. Thus, involvement with the textual form of programs is important for any novice programmer. The textual form of programs supported by Koios can be more readable and informative (see Section 3.2.1.5) than the textual form of programs supported by other programming tools. Therefore, presenting programs in textual form during execution draws users' attention to this form as well.

In order to make programs' execution more comprehensible and reduce working-memory load, each command is highlighted while it is being executed, and explanatory and commentary messages are shown regarding the function of the command (du Boulay et al., 1999; Petre et al., 1998). Two features that further increase the interaction between Koios and users, following the seventh design principle, which suggests a high level of interaction between IDEs and users (see Section 2.5.7) are the following. First, buttons that can start, pause and terminate as well as control the speed of programs' execution are available. Second, data input during execution is performed via dialogue windows.

3.2.2 Design Choices beyond the Seven Design Principles

The core of designing Koios was based on the seven design principles. However, in order to produce a state-of-the-art programming tool, additional design choices were required for this process. The most important of these choices that I had to make during the design of Koios follow.

3.2.2.1 Programming paradigm supported by Koios.

The first design choice was *the programming paradigm that Koios would support*. The two most dominant paradigms are the procedural and the object-oriented (OO) paradigm. Despite the ongoing debate on the appropriateness of each paradigm (Corritore & Wiedenbeck, 2001; Duke, Salzman, Burmeister, Poon & Murray, 2000; Milne & Rowe, 2002; Wiedenbeck & Ramalingam, 1999; Wiedenbeck, Ramalingam, Sarasamma & Corritore, 1999), the procedural paradigm was selected. This choice was made based on the set of programming constructs that are supported by Koios. This set can be supported by both paradigms, but it does not contain any OO concepts. Thus, the selection of the OO paradigm would require the introduction of unnecessary programming concepts, which, in turn, would increase extraneous/intrinsic cognitive load (see Section 2.2.2.1). Therefore, the procedural paradigm seemed more suitable. Apart from the increased complexity introduced by the OO paradigm (Rist, 1996), I find the procedural paradigm more appropriate for introductory programming courses, because it seems that the OO paradigm is not more ‘natural’, easier or powerful than the procedural paradigm as hypothesised (Détienne, 1997; Green, 1990a; Wiedenbeck et al., 1999), OO concepts are difficult for novices to understand (Milne & Rowe, 2002) and novices need less time to familiarise themselves with the procedural paradigm (McCracken et al., 2001). Moreover, the curriculum in which Koios was used and the other tools that were compared in this research use the procedural paradigm.

3.2.2.2 Koios' provision for comments in programs.

The second design choice was that *the prototype of Koios does not need to support comments*. The inclusion of comments in programs is recognised as a helpful source in reading and understanding code and a good programming habit. However, the findings presented by Barr, Holden, Phillips and Greening (1999) and Grandell et al. (2005) suggest that the inclusion of comments by novices may yield undesired results. A number of students in the study of Barr et al. (1999) inserted comments that were similar to the code that they had written and in other cases comments did not explain or were not relevant to the code. Grandell et al. (2005) reported that over 65% of the participants in their study did not include comments in their code and that the quality of the comments did not always match the quality of the program. They also reported that reasons for omitting comments could be that students consider comments unnecessary for small programs, that the inclusion of comments delays the programming process and that properly conveying their programming ideas via comments is hard. Based on these findings, it was decided that providing a feature for inserting comments is not necessary, at least in the first versions of Koios.

3.2.2.3 Values of expressions and conditions during execution.

A third design choice was to *omit the presentation of values of expressions and conditions during execution*. A mere presentation of values of expressions and conditions during execution may lead users to a passive stance because they are aware that information is always available. Because of this availability of information, users may be less attentive and not track all the changes in these values. This choice, however, is expected to increase users' cognitive load, because they will have to retain more information in their memory regarding these values. However, because of the active engagement with this information, this extra cognitive load could be germane (Hundhausen et al., 2002; Tudoreanu & Kraemer, 2008) and, thus, reveal more about programs' execution than a simple presentation of values. In order to assist users in this task, Koios notifies them when the values of variables change, via relevant explanations and comments during execution. The new values are not revealed, but a description regarding the change is available. Although this choice partially violates the requirement for visibility (du Boulay et al., 1999; Pane & Myers, 2000), it was followed in order to engage users in a more active tracing of programs' execution. Furthermore, omitting the presentation of changed values aims at serving the following purpose as well. To overcome the lack of this type of information students are intended to take up a useful program- tracing and debugging practice; that of monitoring the values of variables, expressions or conditions, in which they are interested, by using an output statement to show these values on screen during execution.

3.2.2.4 Minor design choices.

During the design process I faced a number of minor design issues. I dealt with these issues by producing arguments that were in favour of and arguments that were against a specific proposal regarding each design question. Based on these arguments a choice was made for each design issue. These choices were primarily made with respect to the educational benefit of novice programmers, and not without overlooking the likelihood that these choices could slow down the programming process for more advanced programmers. The nature of these issues allows for their brief presentation in tabular layout. Each row of the first column of Table 3.1 presents a design choice. In some cases, a sentence in parentheses follows and describes how this design choice was refined in Koios' programming environment. The second column presents arguments that were considered for accepting this proposal, while the third column presents arguments that were considered for rejecting it. These choices and arguments are presented in Table 3.1.

Table 3.1
Design Choices with Arguments in Favour and Against

Design choice	Arguments in favour	Arguments against
Identifiers are not case-sensitive and specifically all letters are capital.	There would be no confusion between identifiers with the same name in uppercase, lowercase or any possible combination.	Many commercial programming languages are case-sensitive.
The names of constants, variables, and arrays will be unique within the scope of each procedure and/or function.	Different items (constants, variables and arrays) of different procedures/functions can share the same name, highlighting the concept of scope.	Different items (constants, variables and arrays) with the same name that belong to different functions and/or procedures could be mistakenly considered as the same item.
The names of global constants, variables and arrays cannot be the same as the names of local constants, variables, arrays and parameters and vice versa.	Allowing the same names could cause confusion regarding which one of the objects that share the same name is used in each case. This is not allowed by the majority of programming languages.	This could be a good example of how local scope overrides (shadows) global scope. However, perhaps this is too complicated for novices.
Constants, variables and arrays can be declared everywhere in programs. (Koios automatically moves all declarations in the beginning of a function/procedure before compilation.)	It is not necessary to declare items in a particular part of a program, but one can declare an item whenever or wherever is needed.	One would have to search the whole program to locate items' declaration, instead of looking at a specific part of the program.
Constants, variables and arrays can be either initialised or uninitialized during their declaration. The default selection is to be initialised.	An uninitialised variable/array may cause errors during the execution of programs.	In this way a clear distinction can be made between constants that have to be initialised and variables that do not.

Table 3.1 (continued)

Design choice	Arguments in favour	Arguments against
Moving (cutting) constants from one location of programs to another location of the same program is allowed. (Koios allows moving items only if they are not used by other commands.)	If an item is created in an inappropriate location of programs, it can be very easily reallocated to the right one. Alterations of the same program can be done without deleting and recreating parts of a program.	Moving an item may affect other commands of programs, for example moving a variable that is used in a for-statement.
Copying items from one program or location of a program to another location or program is not permitted.	A copied item would probably need some modifications in order to properly work in its new location and this could cause more errors than just creating a new one. The process of creating a new item has equal or lower level of complexity than the process of modifying a copied item.	A copy of already created items cannot be reused and identical objects have to be created from scratch.
Arrays cannot be passed as parameters into functions or procedures.	In many modern and commercial languages arrays are passed as pointers into functions or procedures, but the concept of pointers could be considered too advanced for novices.	Some programs could be easier implemented if arrays could be passed as parameters.

Table 3.1 (continued)

Design choice	Arguments in favour	Arguments against
Only one-dimensional and two-dimensional arrays are supported.	Arrays with more than two dimensions are generally not used in introductory programming courses. The representation and manipulation of one-dimensional and two-dimensional arrays could be relatively easy and comprehensible for students. Once programming examples of one-dimensional and two-dimensional arrays are understood, the use of multi-dimensional arrays becomes more a problem-solving tool rather than an educational aim.	More advanced programming examples may require the use of multi-dimensional arrays.
Functions have a return type, while procedures do not.	The use and role of functions and procedures in programs can be easily clarified and distinguished.	Some commercial languages do not have a special implementation of procedures.
Different functions and procedures cannot have the same name.	In order for functions and procedures to share the same name, they should be able to be overloaded and the concept of polymorphism would have to be introduced, but this concept could be considered too advanced for novices.	More advanced programs could use overloaded functions.

Table 3.1 (continued)

Design choice	Arguments in favour	Arguments against
The name of functions or procedures cannot be the same as the name of their parameters.	Parameters can be distinguished from their functions or procedures.	
The symbols == and != will be used for equality an inequality operators, respectively.	A clear distinction between the equality operator (=) in mathematics and the assign (=) operator in many popular programming languages can be made. The transition to a commercial programming language will be smoother as users will have been familiarised with these symbols (==, !=) that are widely used in popular commercial programming languages.	The words *equal to* and *not equal to* could be used respectively instead of symbols.

Naturally, the design process was not and could not be limited to the choices that are reported here. Therefore, the most significant design choices were presented in this section. Further choices regarding the development and implementation of Koios are discussed in the following section.

3.3 Development of Koios

The Koios programming environment was developed in Java, using the NetBeans IDE – versions 6.5 and 6.7 – and Java Development Kit 6. I chose to develop Koios in Java because it is hardware-independent and platform-independent (Gosling & McGilton, 1996). This means that, theoretically, Java applications can run in any hardware configuration, for example on a desktop or a tablet PC, and on any operating system, such as MS Windows, Mac OS X and GNU/Linux, provided that the Java Virtual Machine is installed. Thus, Koios can run on any machine that has that Java Virtual Machine installed. This way, hardware-dependence, a ‘deadly sin’ of PLs for novices, according to McIver and Conway (1996), is avoided.

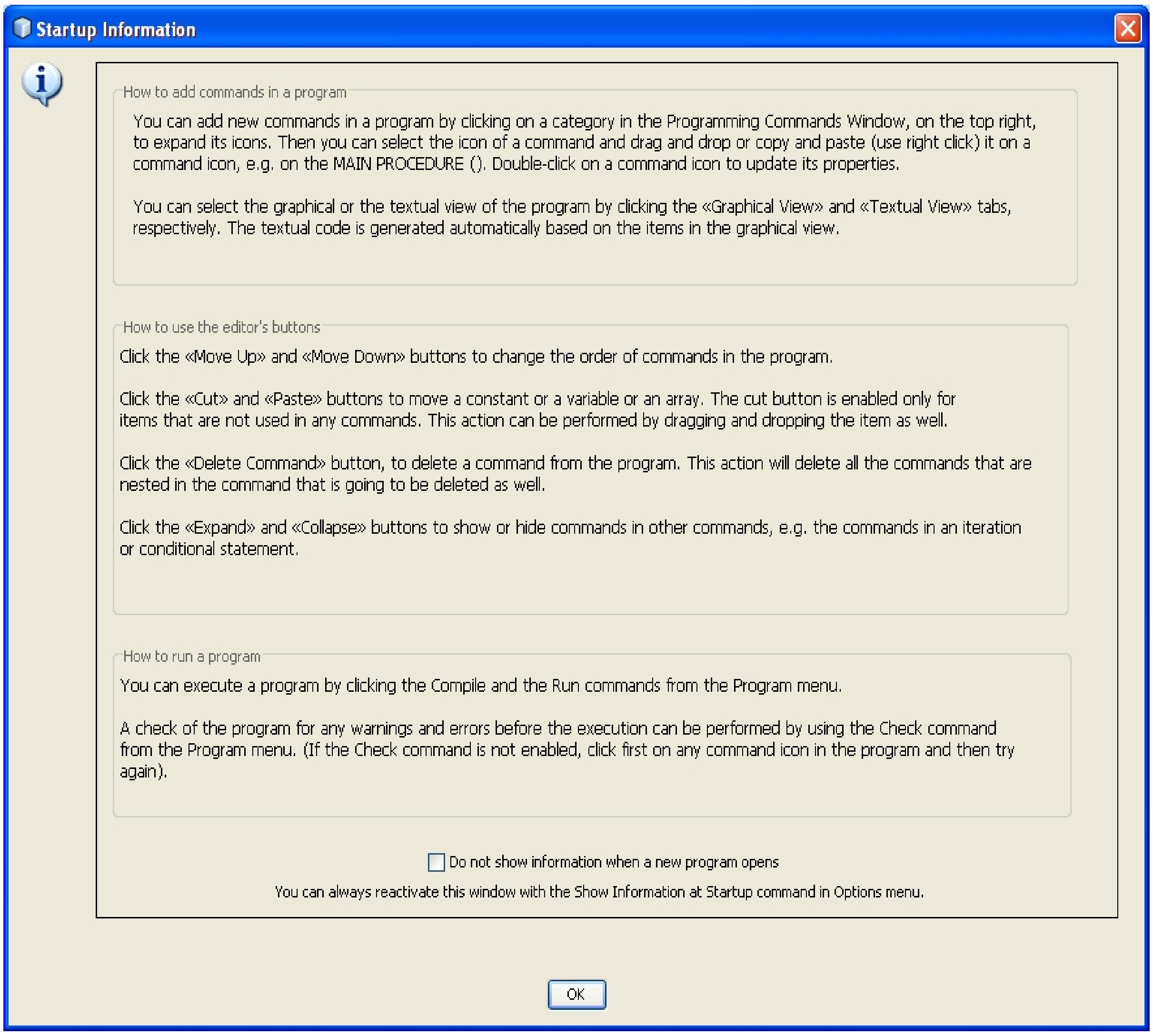
Three versions of Koios have been developed at present. Two versions of Koios - Version 0 and Version 1 (see Section 3.4) - are in Greek and were used in the two rounds of data collection, respectively (See Section 4.3). The development of the Koios prototype Version 0 started in February 2009 and ended in September 2009, while Version 1 was developed between June 2010 and October 2010. The third one is a translation in English of the Greek version of Koios Version 1. It was developed for illustrative reasons as well as for people that may be interested in this software (users, researchers and others) and are unfamiliar with the Greek language. In this section, snapshots of the English version of Koios are used to demonstrate the implementation and present the software. This demonstration aims at highlighting important features of the Koios programming environment that are intended to facilitate novices in the creation and execution of their programs.

3.3.1 Koios' User-interface

Koios was developed to have a simple user-interface, in order to make it easier for novice programmers to use it and avoid exposing them to more complex features, such as managing a project workspace (DePasquale, Lee & Pérez-Quiñones, 2004). When Koios runs for the first time and the *New Program* button is clicked the welcome screen in Figure 3.1 appears.

This screen provides information about how to use the interface, and how to create and run programs. This screen can be set not to appear anymore when it is not needed. When the *OK* button of the welcome screen is clicked, the main window of Koios appears (see Figure 3.2) and a new program can be created. Thus, from this point on Koios is in Program Creation mode.

As can be seen from Figure 3.2, the user-interface of Koios in Program Creation mode displays a menu bar, a toolbar, a status bar and three main windows: (a) the Program window, (b) the Programming Commands window and (c) the Properties window. The menu bar contains six drop-down menus. The toolbar contains 12 icons that are shortcuts for frequently-used menu items, such as saving files or compiling and running programs. The items of each drop-down menu are shown in Table 3.2.


Startup Information
How to add commands in a program
How to use the editor's buttons
How to run a program
OK

You can add new commands in a program by clicking on a category in the Programming Commands Window, on the top right, to expand its icons. Then you can select the icon of a command and drag and drop or copy and paste (use right click) it on a command icon, e.g. on the MAIN PROCEDURE (). Double-click on a command icon to update its properties.

You can select the graphical or the textual view of the program by clicking the «Graphical View» and «Textual View» tabs, respectively. The textual code is generated automatically based on the items in the graphical view.

Click the «Move Up» and «Move Down» buttons to change the order of commands in the program.

Click the «Cut» and «Paste» buttons to move a constant or a variable or an array. The cut button is enabled only for items that are not used in any commands. This action can be performed by dragging and dropping the item as well.

Click the «Delete Command» button, to delete a command from the program. This action will delete all the commands that are nested in the command that is going to be deleted as well.

Click the «Expand» and «Collapse» buttons to show or hide commands in other commands, e.g. the commands in an iteration or conditional statement.

You can execute a program by clicking the Compile and the Run commands from the Program menu.

A check of the program for any warnings and errors before the execution can be performed by using the Check command from the Program menu. (If the Check command is not enabled, click first on any command icon in the program and then try again).

☐ Do not show information when a new program opens

You can always reactivate this window with the Show Information at Startup command in Options menu.

Figure 3.1. Welcome screen of Koios.

The Program window can show programs in a graphical or a textual view. The desired view can be selected from the tabs that are located in the lower left corner of the window. This window also provides a number of buttons for manipulating the icons of programming items. These buttons are located on the right side of the window.

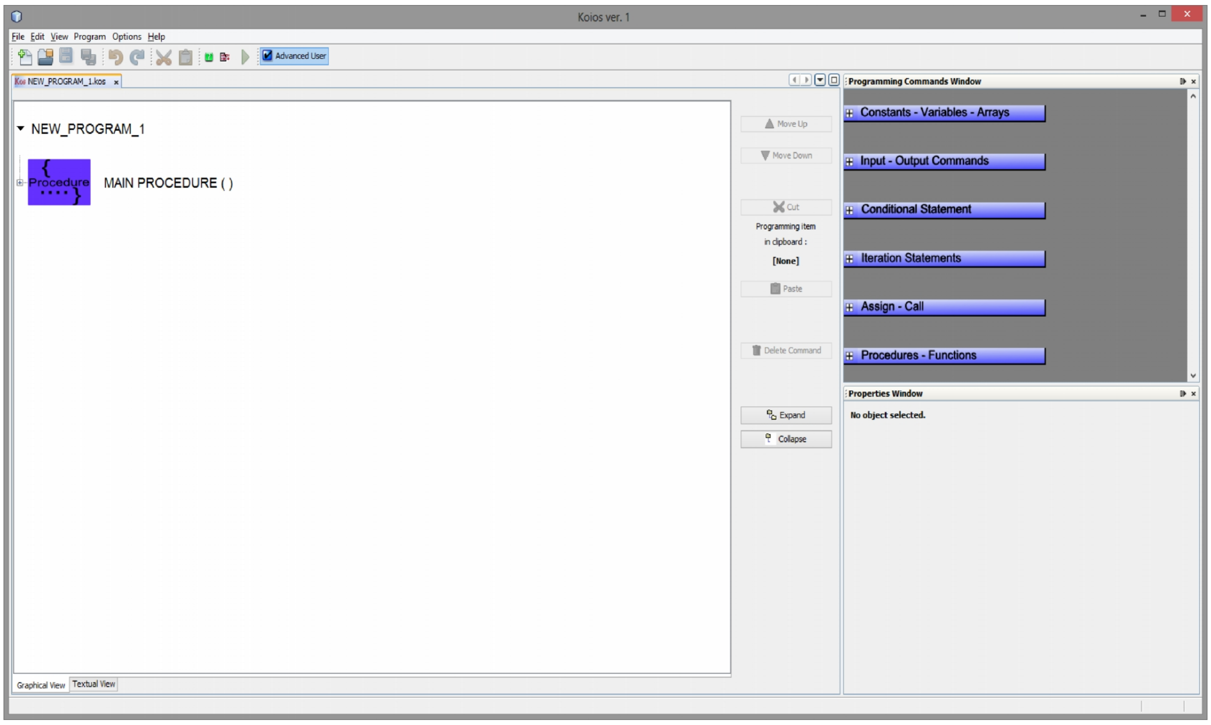


Figure 3.2. The main window of Koios programming environment.

Table 3.2
Menus and Menu Items of Koios

Menu	Menu Items
File	New Program Open Save Save As Save All Page Setup (print preferences) Print Exit
Edit	Undo Redo Cut Paste Find
View	Show Programming Commands window Show Properties window Show Line Numbers Full screen
Program	Check (see Section 3.3.3) Compile Run
Options	Advanced User Export Java Program Export Program as Image File Show Information at Start-up
Help	About

The Programming Commands window contains all the available programming commands, which are grouped in six categories. Based on the programming constructs that were chosen to be supported by Koios (see Section 3.2.1), the programming commands/items that were implemented are *constants*, *variables*, *arrays*, *write*, *read*, *assign*, *call*, *if*-statement, *for*-statement, *while*-statement, *do-while*-statement, *return procedures* and *functions*. Each programming command is represented by an icon. The icons of the available programming commands and their grouping can be seen in Figure 3.3.

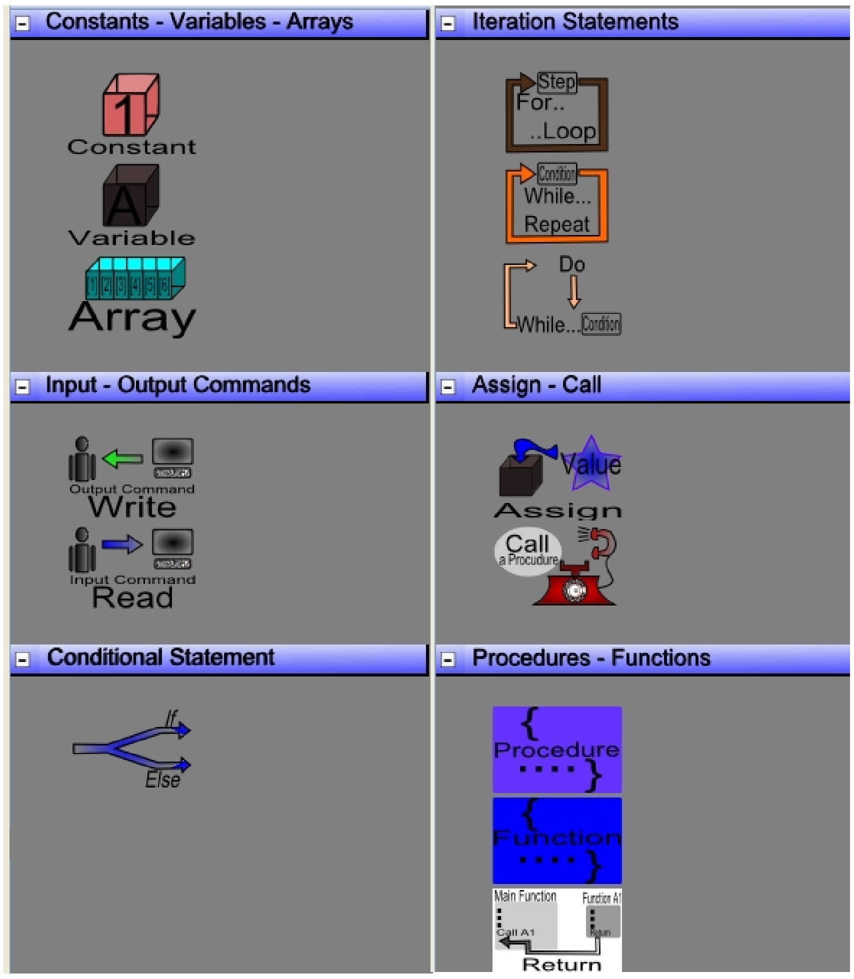


Figure 3.3. Icons of programming items.

The Properties window shows all the available properties of the programming item that is selected, for example name and value. This can be used to quickly show the properties of an item by selecting its icon. The properties cannot be changed from the Properties window. This can be done only by double-clicking on the item's icon. The reasons for this choice are explained in Section 3.3.2.

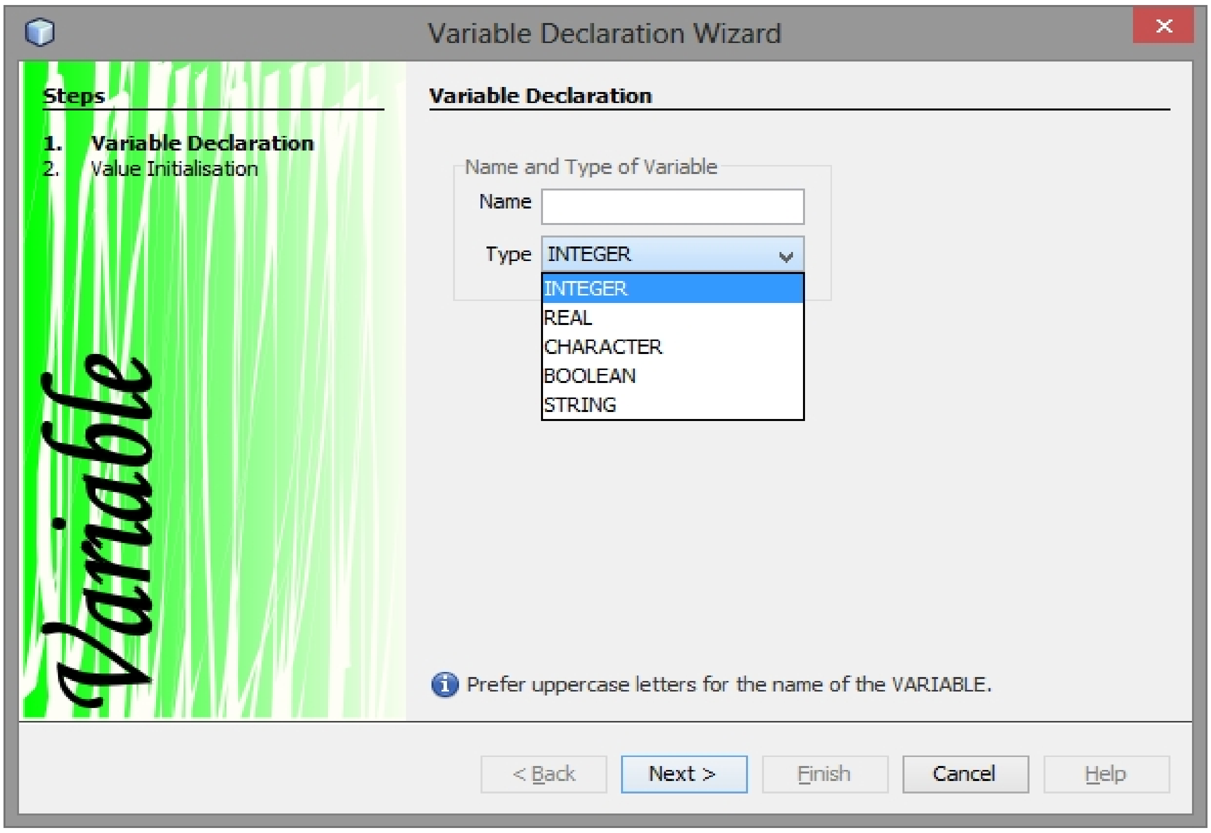
3.3.2 Creating Programs with the Koios Programming Environment

In order to further increase the usability of Koios, two user-modes are offered, namely Simple User mode and Advanced User mode. Advanced User mode enables all programming items (see Figure 3.3), while Simple User mode makes available only the programming items of *variable*, *input*-statement, *output*-statement, *call*-statement, *assign*-statement, *if*-statement, *for*-statement and *procedure*. This distinction was made because, according to the planned experiments (See Section 4.2.2), Koios was going to be evaluated in secondary and tertiary educational institutes Greece. The programming constructs that are included in the CS curriculum of secondary institutes are those available in the Simple User mode, while those included in the CS curriculum of tertiary institutes are available in the Advanced User mode.

New commands/items can be added to a program by selecting an icon from the Programming Commands window and dragging-and-dropping or copying-and-pasting it to the program. In order to prevent errors, only valid commands are allowed to be entered in programs, for example the icon of a return statement can be added in functions but not in procedures. When a new item is dropped or pasted in the program a dialogue between Koios and users is triggered to guide them through the creation of the command and determine its properties.

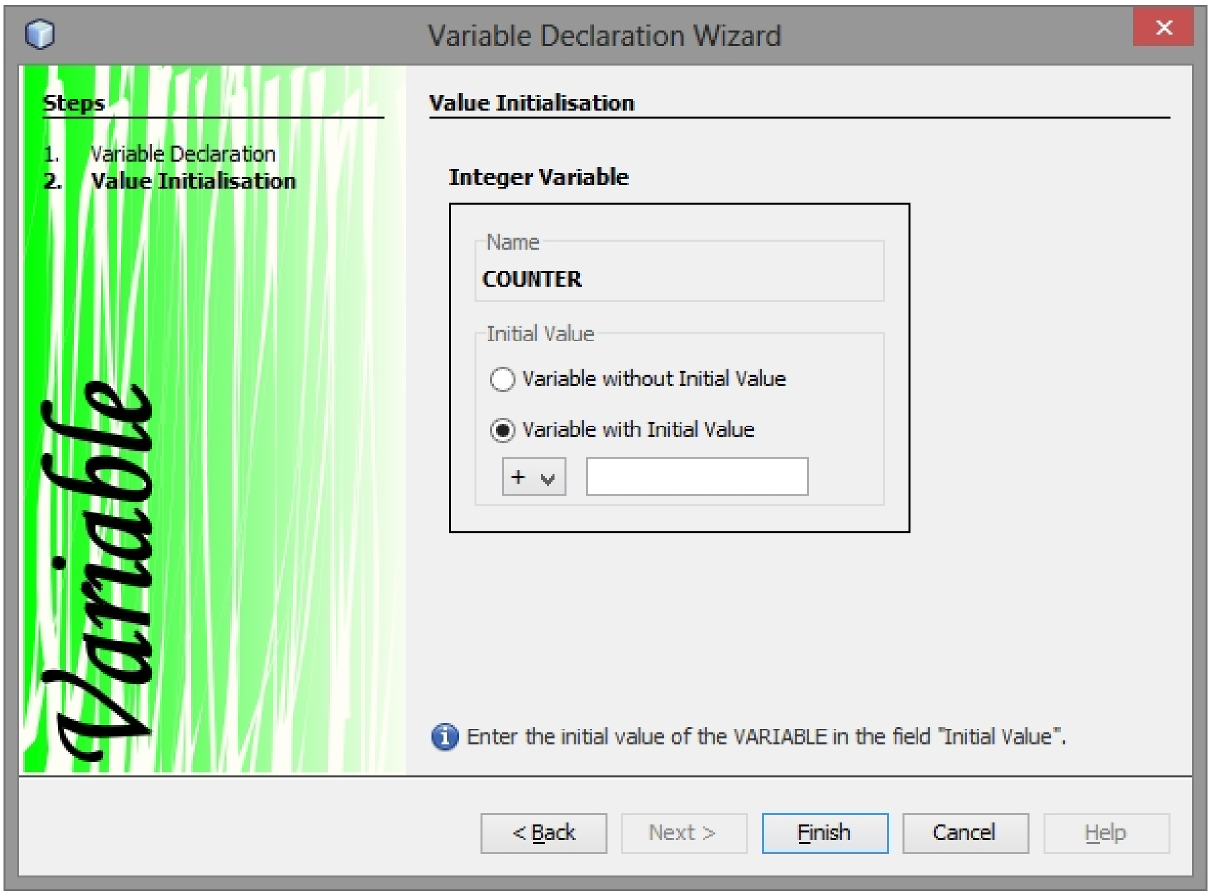
In order to demonstrate the process of program creation with Koios, the creation of a program that was used in data collection (See Appendix C.5) is presented here.

This is a simple program often used in introductory programming courses and its aim is to display the numbers from 1 to 10 using a *for*-statement. This program requires an integer variable, a *for*-statement and an *output* statement. First, an integer variable with the name COUNTER must be declared. Selecting the variable icon from the Programming Commands window and dragging-and-dropping it to the MAIN PROCEDURE () icon in the Program window initiates a dialogue between Koios and users that guides them through the declaration of variables. The dialog windows involved in this action can be seen in Figure 3.4 and Figure 3.5.



Prefer uppercase letters for the name of the VARIABLE.

Figure 3.4. The first dialogue window for declaring a variable.



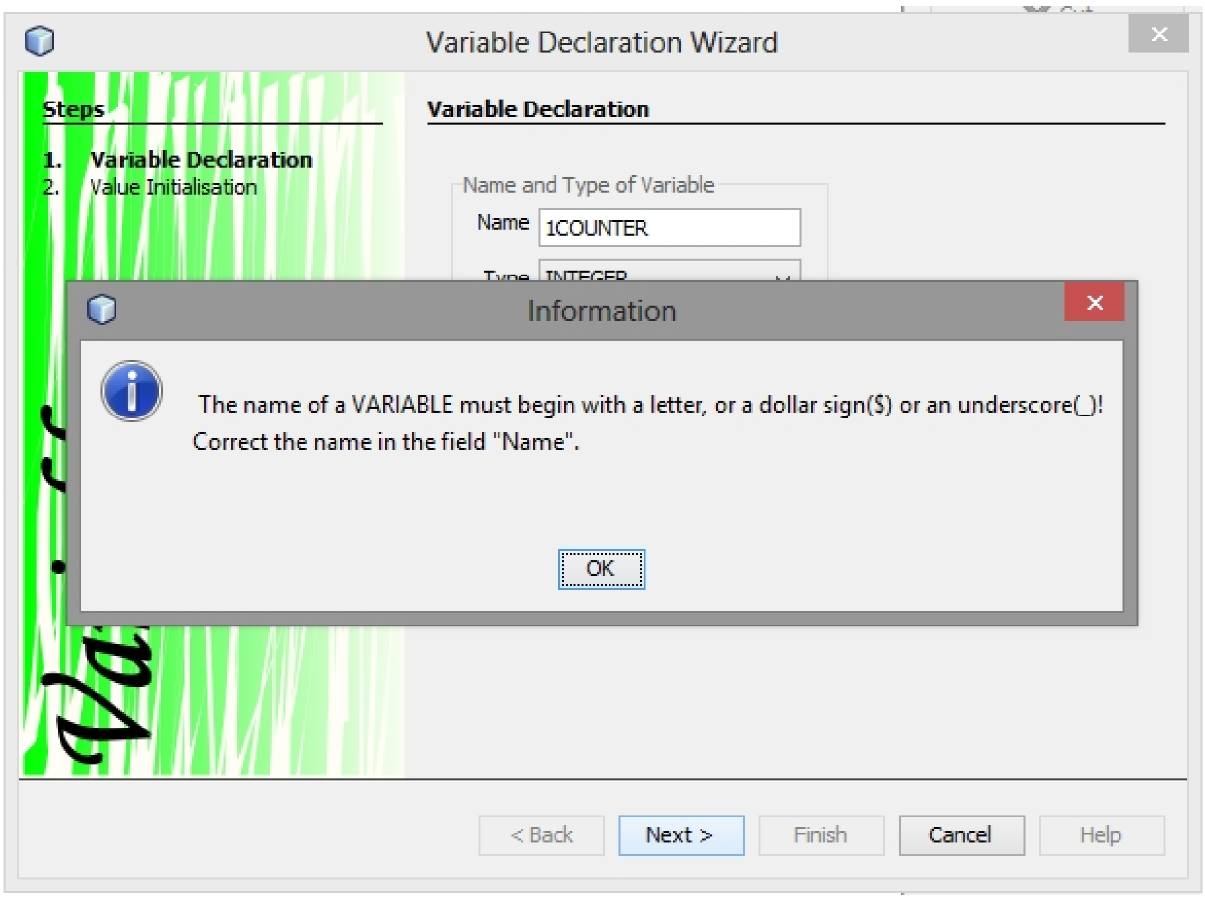
COUNTER

( ) Variable without Initial Value
(●) Variable with Initial Value

Enter the initial value of the VARIABLE in the field "Initial Value".

Figure 3.5. The second dialogue window for declaring a variable.

Whenever the *Next* and *Finish* buttons are clicked, the properties entered so far are checked. If a property is not entered correctly, then an error message appears in a dialogue window. For example, if an identifier (that is a name of a variable, constant, array, function or procedure) that begins with a number is entered as the name for a variable, then the error message in Figure 3.6 appears.



Name: 1COUNTER

Type: INTEGER

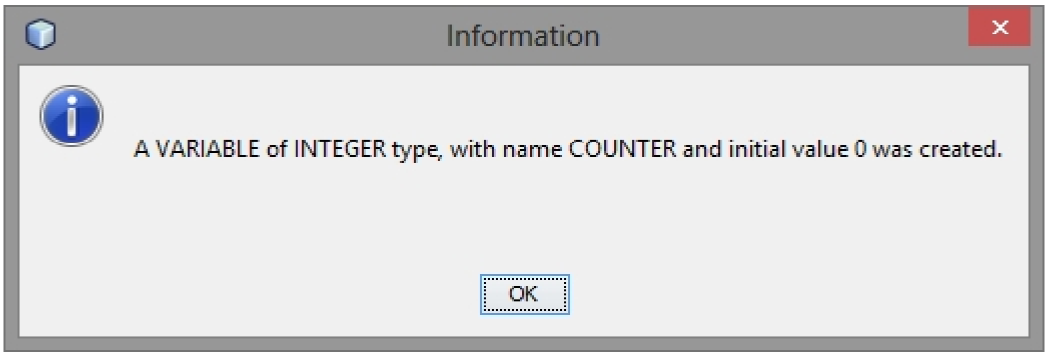
Information

The name of a VARIABLE must begin with a letter, or a dollar sign($) or an underscore(_)!
Correct the name in the field "Name".

OK

Figure 3.6. An error message regarding the name of a variable.

When all the properties have been entered correctly, a variable icon is created in the MAIN PROCEDURE () and the declaration of the variable is completed. A message appears informing the user that a variable has been successfully declared (see Figure 3.7).

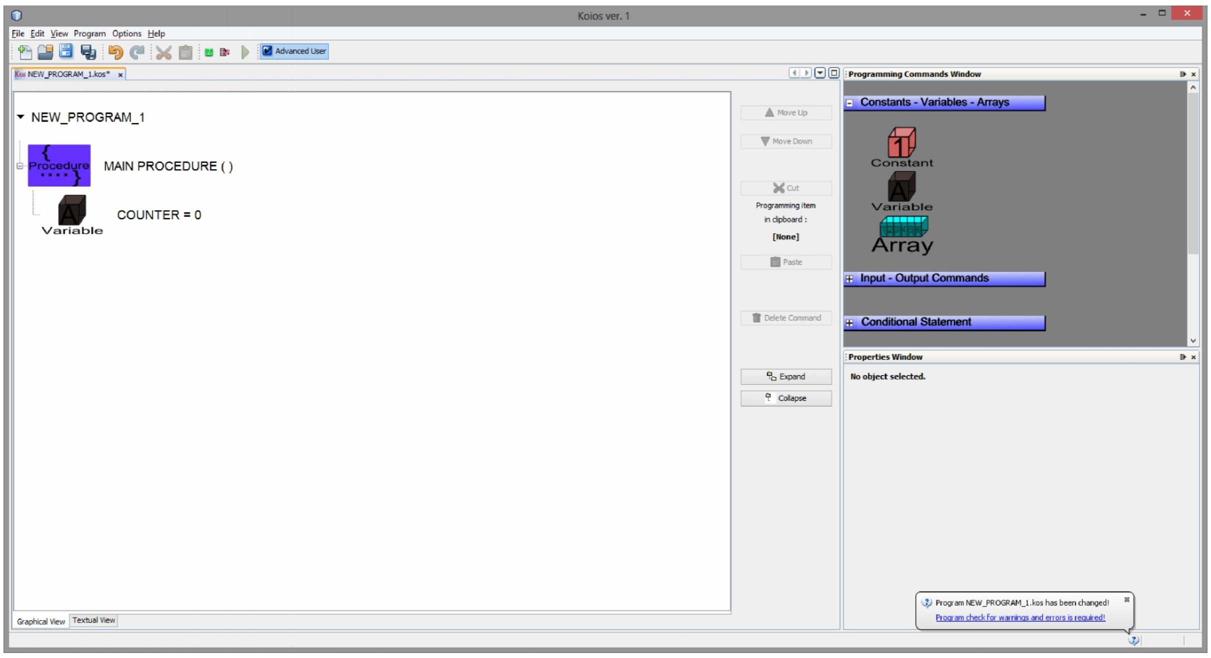

Information

A VARIABLE of INTEGER type, with name COUNTER and initial value 0 was created.

OK

Figure 3.7. An information message for successful variable declaration.

Hence, only items with no errors are entered in the program; therefore the program is always in a syntax-error-free state. The result of the addition of the variable in the program is shown in Figure 3.8.



NEW_PROGRAM_1
MAIN PROCEDURE ( )
COUNTER = 0

Programming item in clipboard : [None]

No object selected.

Program NEW_PROGRAM_1.kos has been changed!
Program check for warnings and errors is required!

Figure 3.8. A program with a variable.

In the status bar a message appears on the left side that describes the last action that was performed. On the right side of the status bar a balloon appears that provides information about the status of programs. This balloon appears any time that a change in a program occurs. A change in a program can be an addition of, a deletion of, cutting and pasting, and moving up or down programming items. These actions can be performed by the buttons provided on the right side of the Program window.

As can be seen in Figure 3.8, the items of a program are represented as icons and are associated with brief text, which reads programming items' most important properties. The aim of this is not only to help users to identify different programming items that have the same icon, but also to prevent them from splitting their attention between graphical view and Properties window or even textual view by providing critical information regarding items' properties (Shaffer et al., 2003). Each programming item is represented by a node, which consists of an icon and the associated text. The nodes of programs are organised in a tree structure. This structure is used for the representation of files and folders by the majority of operating systems. This choice of organising programming items/nodes in a tree structure was made for three reasons. First, it is possible that this view can convey the concept that there can be two categories of nodes/programming items (a) compound statements, which are represented by nodes/programming items that can 'include' one or more nodes, like the *if*-statement and *for*-statements and (b) simple statements, which are represented by nodes/programming items that cannot, like an output statement. Hence, a parallel could be drawn; compound statements resembles the functionality of folders, while simple statements the functionality of files. The term functionality here is used with respect to their ability of containing or not containing other nodes/items. Second, users may be already familiar with the tree view from their interaction with computers' operating system. Even if this is not the case, this representation could be a good metaphor for distinguishing the functionality of these two categories. Third, this view can provide a good mechanism for showing/hiding the nodes/programming items within a node/item, such as the commands within a *for*-statement. Thus, the size of the visible program can be adjusted. This can be accomplished either for individual nodes by clicking on the handle of nodes (represented by a + or a – sign within a rectangle box) or for the entire program by using the *Expand* and *Collapse* buttons in the Programming window.

The next action is to add a *for*-statement. Following the same steps with the creation of variables, the dialogue for the addition of a *for*-statement begins. The first dialogue window is shown in Figure 3.9.

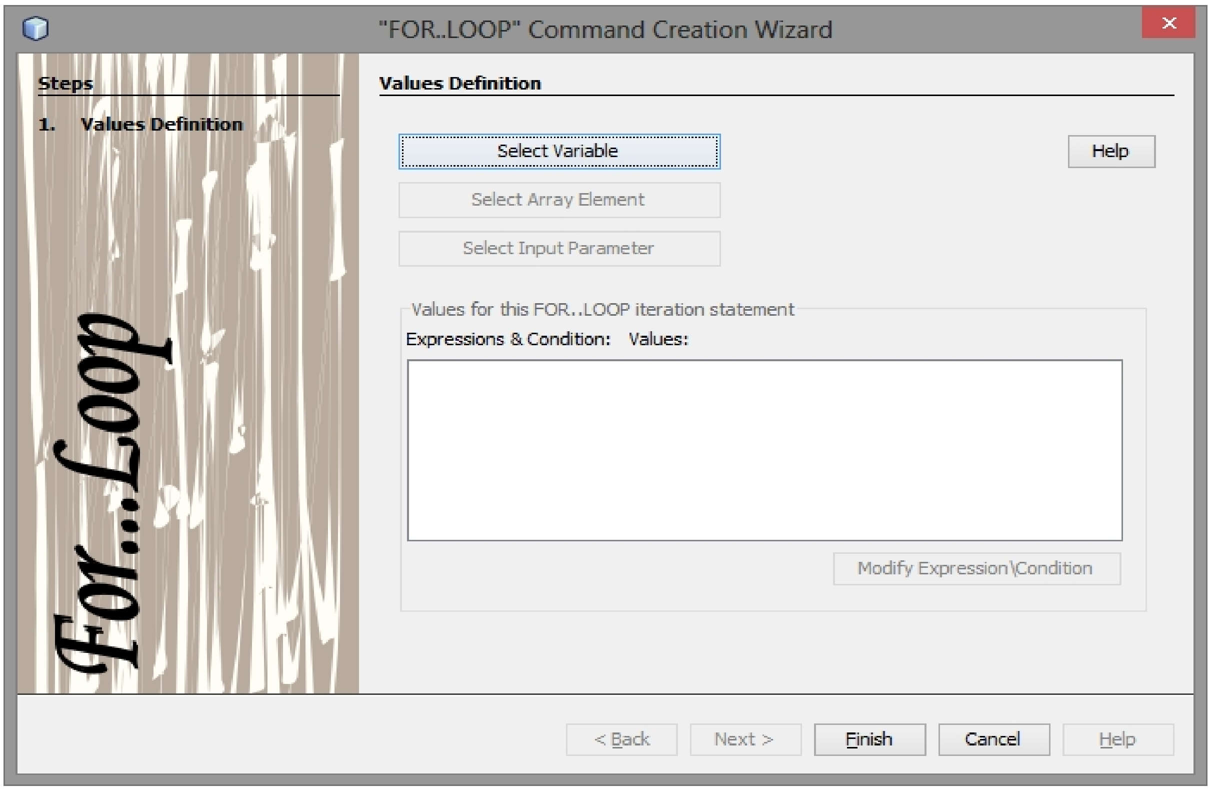
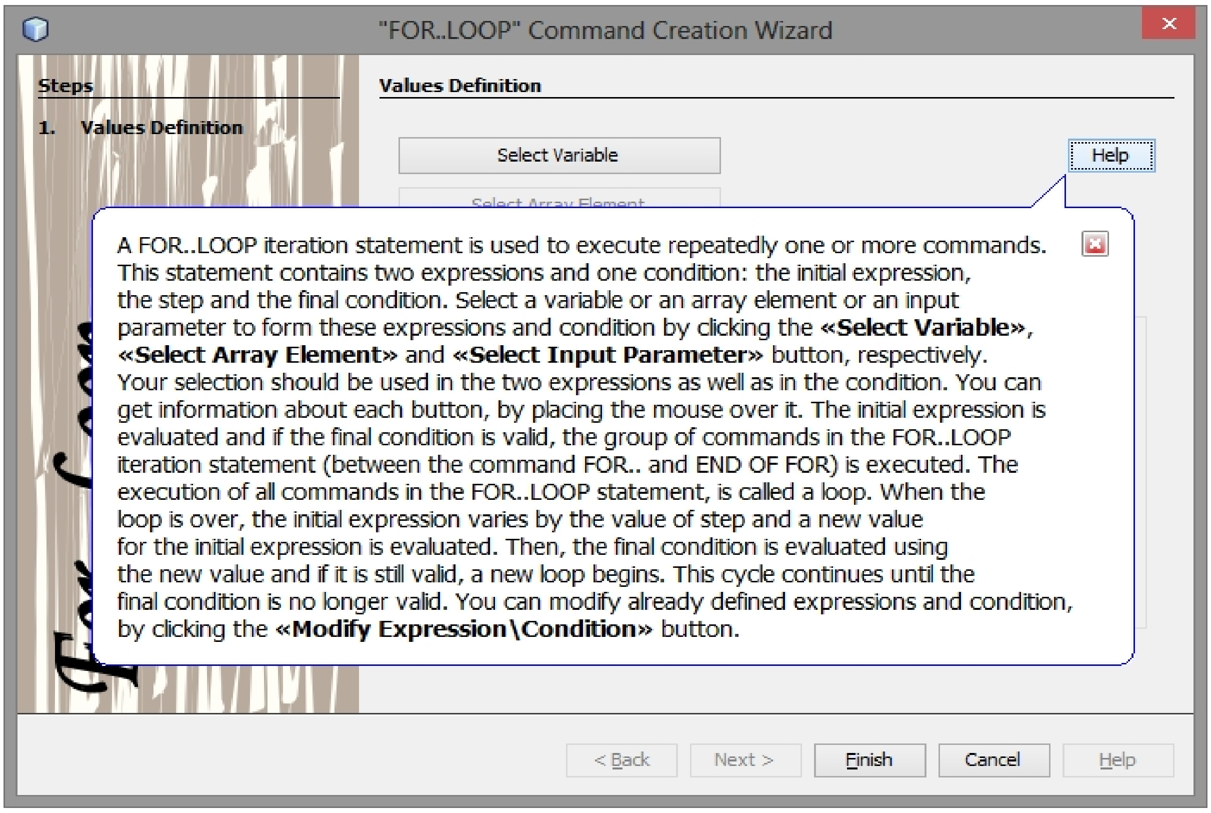


Figure 3.9. The first dialogue window of a for-statement.

In this dialogue window the name of the variable that will be included in the *for*-statement is selected. From the three buttons that can be used for selecting programming items, only the *Select Variable* button is enabled, because in this program the only item that has been declared is a variable. If input parameters or arrays had been declared as well, the associated buttons would have been enabled. If users need help with properties of a *for*-statement, they can click on the *Help* button. In this case a help message appears and can be seen in Figure 3.10. All the dialogue windows of every programming item include a *Help* button when it is required.



A FOR..LOOP iteration statement is used to execute repeatedly one or more commands. This statement contains two expressions and one condition: the initial expression, the step and the final condition. Select a variable or an array element or an input parameter to form these expressions and condition by clicking the **«Select Variable»**, **«Select Array Element»** and **«Select Input Parameter»** button, respectively. Your selection should be used in the two expressions as well as in the condition. You can get information about each button, by placing the mouse over it. The initial expression is evaluated and if the final condition is valid, the group of commands in the FOR..LOOP iteration statement (between the command FOR.. and END OF FOR) is executed. The execution of all commands in the FOR..LOOP statement, is called a loop. When the loop is over, the initial expression varies by the value of step and a new value for the initial expression is evaluated. Then, the final condition is evaluated using the new value and if it is still valid, a new loop begins. This cycle continues until the final condition is no longer valid. You can modify already defined expressions and condition, by clicking the **«Modify Expression\Condition»** button.

Figure 3.10. A help message regarding a for-statement.

When the *Select Variable* button is clicked, a new window appears that shows all the variables that are declared in this program and are global or within the scope of the subprogram that the *for*-statement is added to. This window is shown in Figure 3.11.

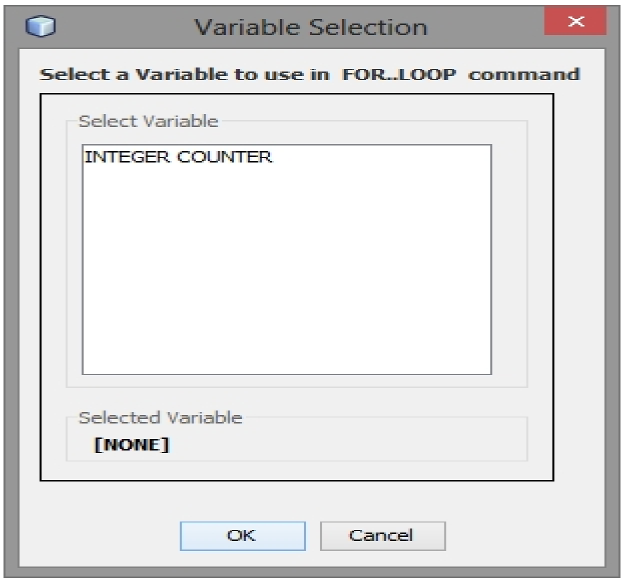
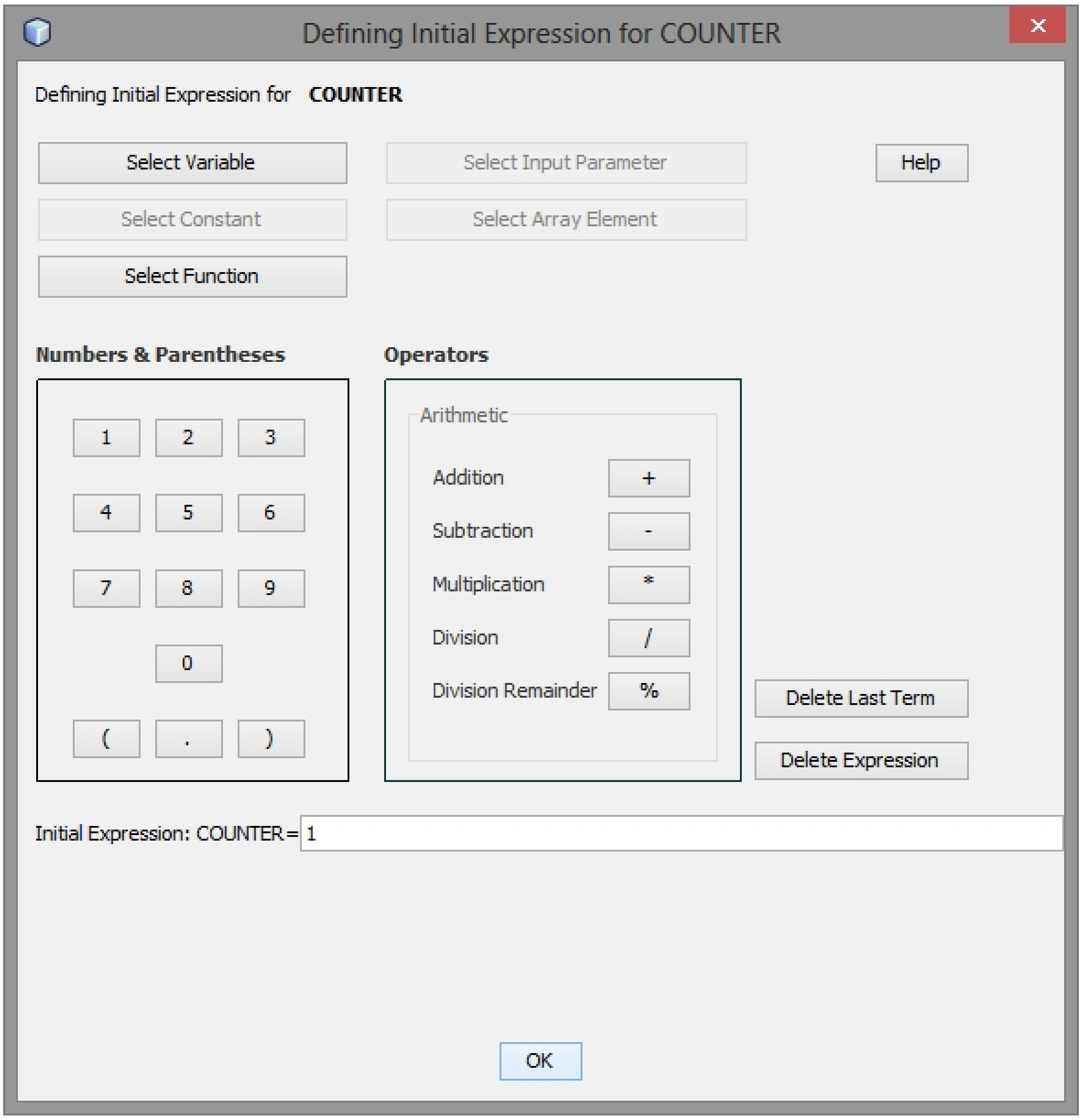


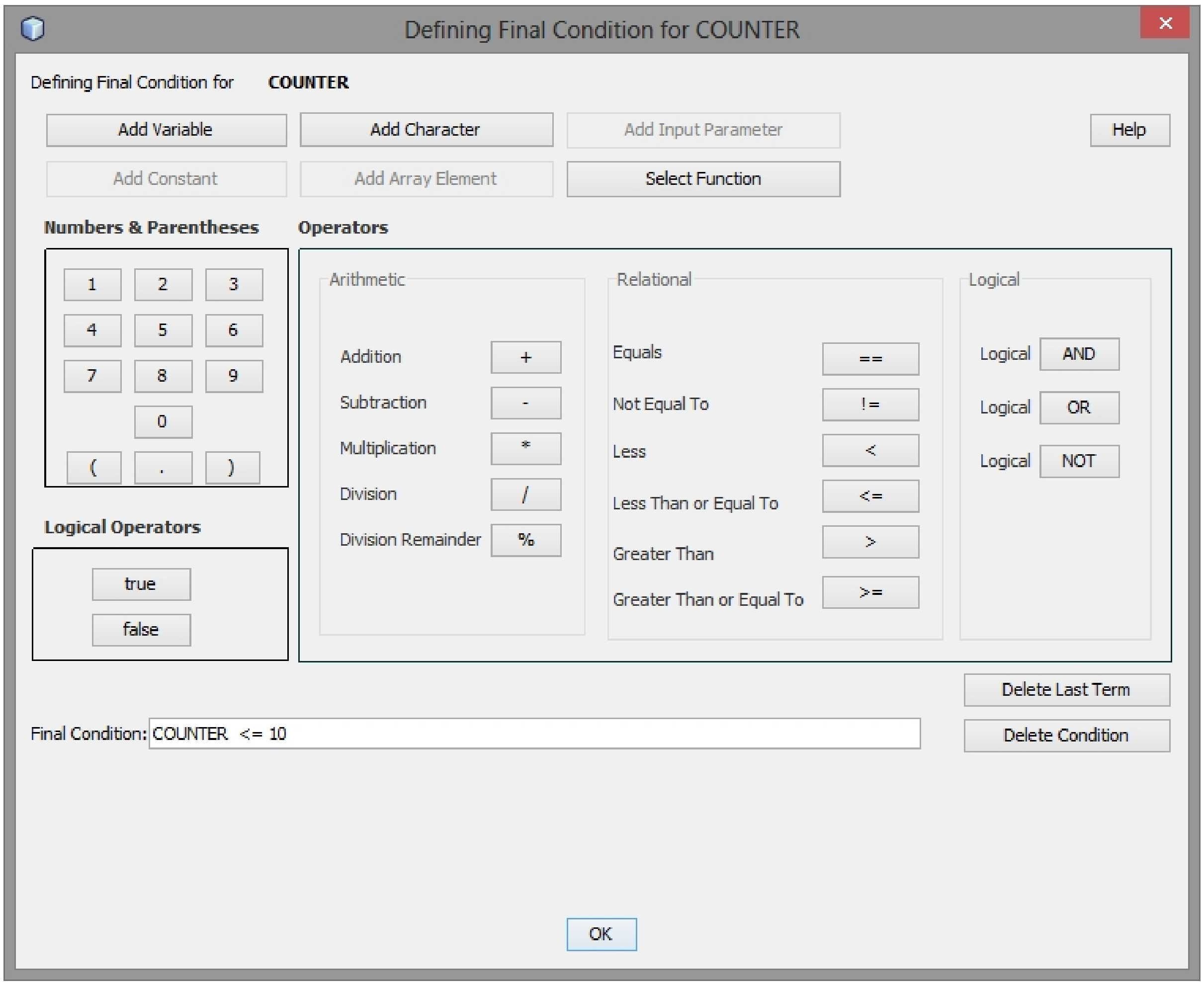
Figure 3.11. The dialogue window for selecting variables for a for-statement.

In this program only the variable COUNTER is declared and therefore only this variable is shown in the window of Figure 3.11. When the variable is selected three new windows appears consecutively. Each window asks users to enter the initial expression, the condition and the step that are required for the *for*-statement. The three windows, in their order of appearance are shown in Figure 3.12, Figure 3.13 and Figure 3.14, respectively.



Initial Expression: COUNTER = 1

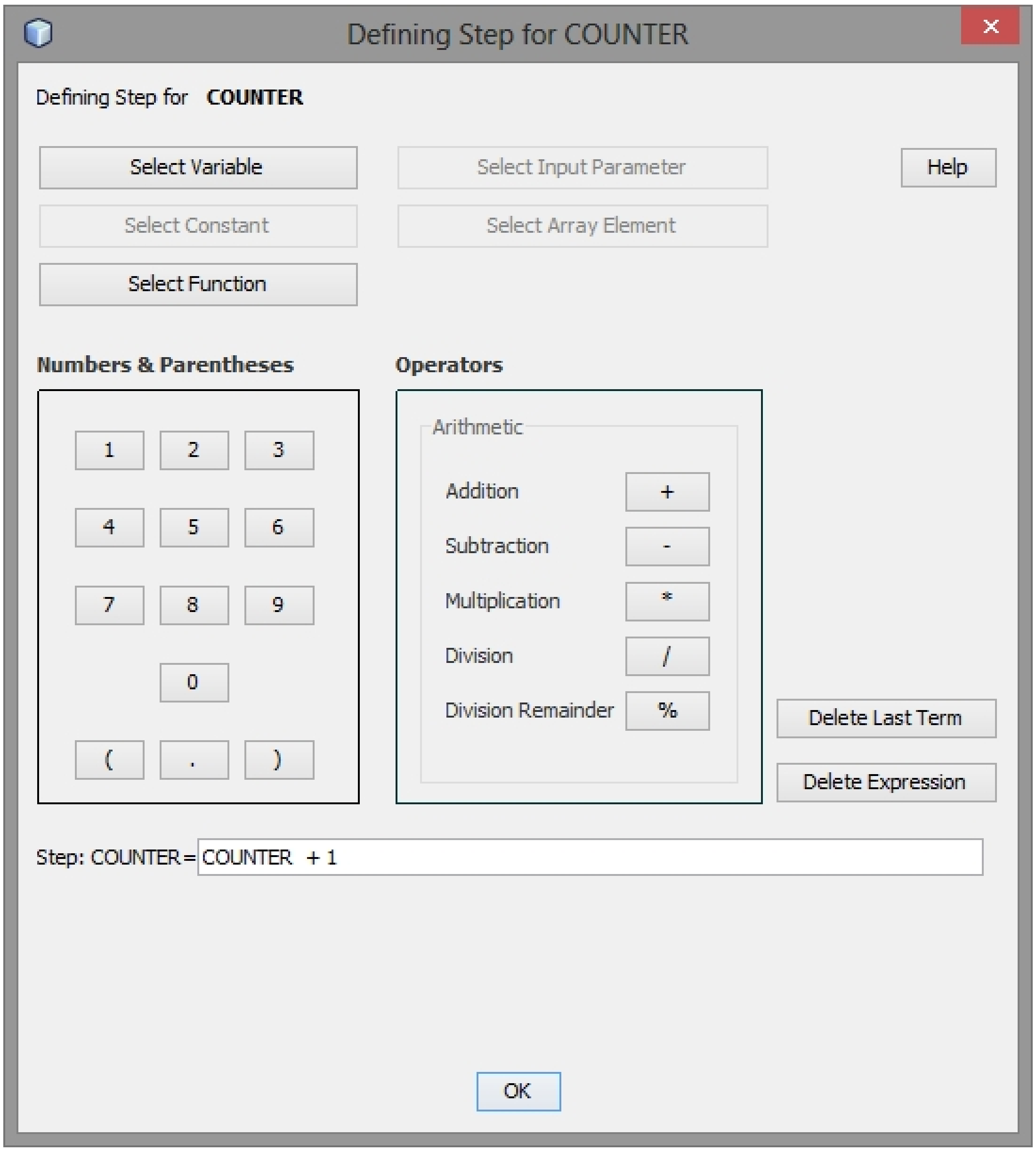
Figure 3.12. The dialogue window for entering the initial expression of a for-statement.



Final Condition: COUNTER <= 10

Figure 3.13. The dialogue window for entering the condition of a for-statement.

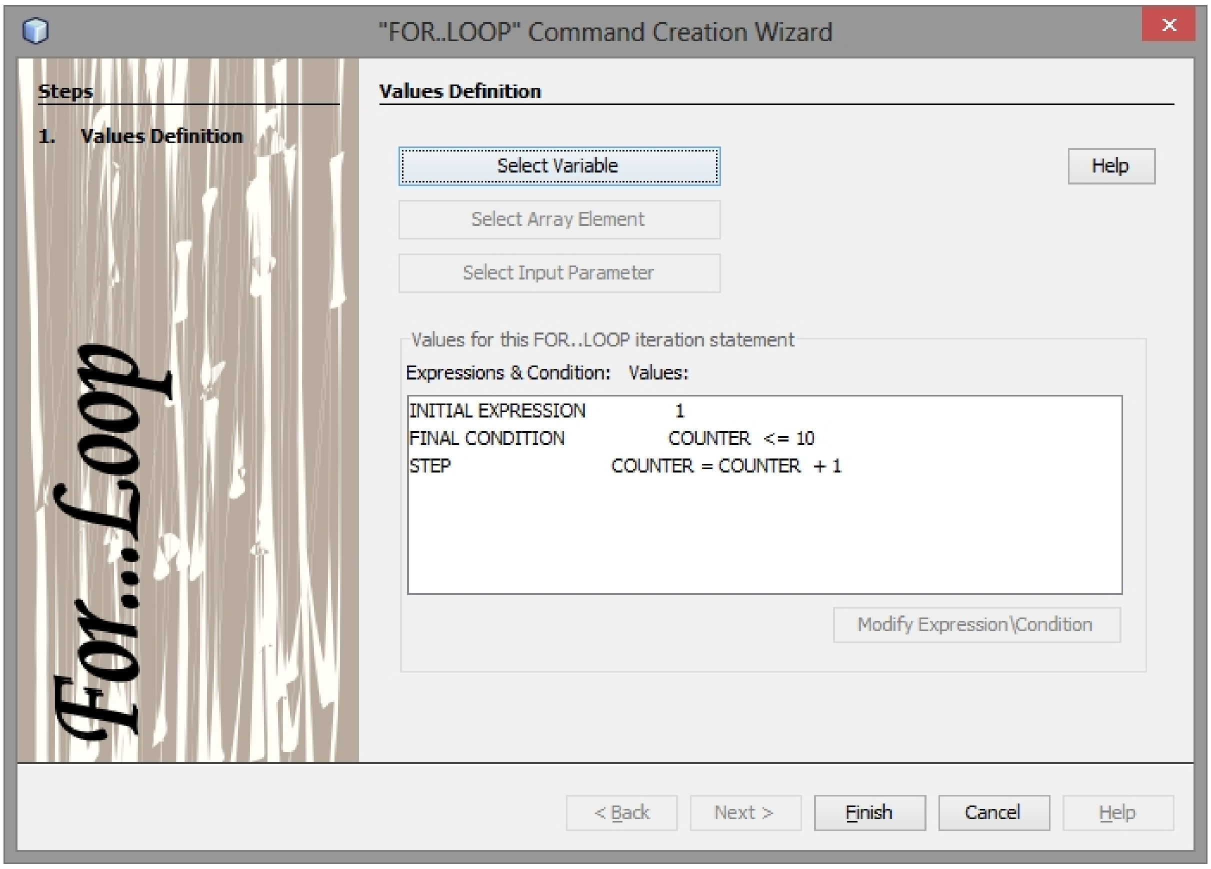
In these dialogue windows, the expressions and conditions are checked and if errors are detected, an error message appears and each dialogue window remains visible until a correct expression or condition is entered. As can be seen from Figure 3.12, Figure 3.13 and Figure 3.14, the *Select Function* button is enabled, although there are no user-defined functions in the program. This is because Koios further assists users by providing nine built-in functions: (a) integer_part, (b) square_root, (c) cosine, (d) sine, (e) tangent, (f) e_to_power_of, (g) logarithm, (h) power and (i) absolute_value.



Step: COUNTER= COUNTER + 1

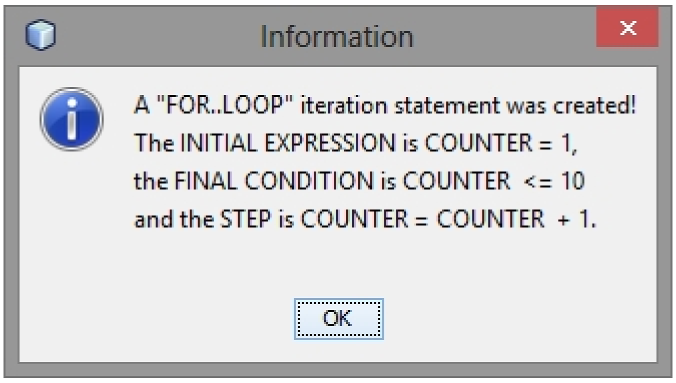
Figure 3.14. The dialog window for entering the step of a for-statement.

Once the initial expression, condition and step have been entered correctly, the window of Figure 3.9 appears again with the completed information (see Figure 3.15). After the *Finish* button of the dialog window has been clicked, an information message (see Figure 3.16) appears and the statement is added in the program.



```
INITIAL EXPRESSION          1
FINAL CONDITION           COUNTER  <= 10
STEP              COUNTER = COUNTER  + 1
```

Figure 3.15. The completed dialogue window of a for-statement.



A "FOR..LOOP" iteration statement was created!
The INITIAL EXPRESSION is COUNTER = 1,
the FINAL CONDITION is COUNTER <= 10
and the STEP is COUNTER = COUNTER + 1.

Figure 3.16. An information message for successfully adding a for-statement.

Next, an output statement will be added within the node of the *for*-statement. The first dialogue window for adding a *write* statement can be seen in Figure 3.17. This statement should be used to print the value of COUNTER together with a message. Therefore, a string is added first in the statement by clicking the *Add String* button.

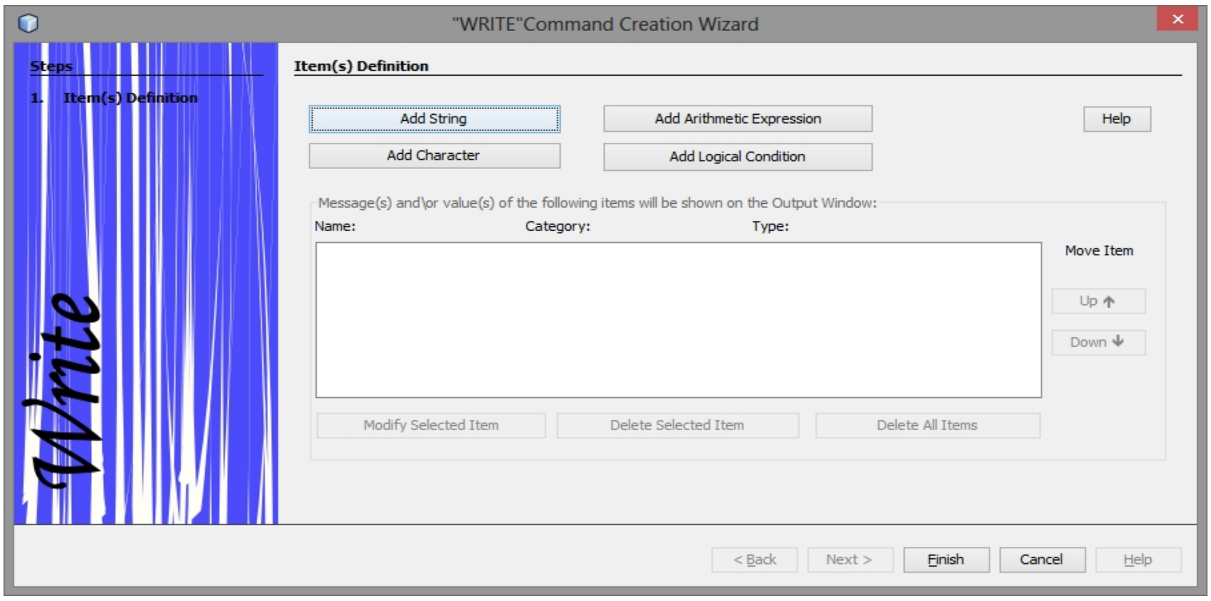
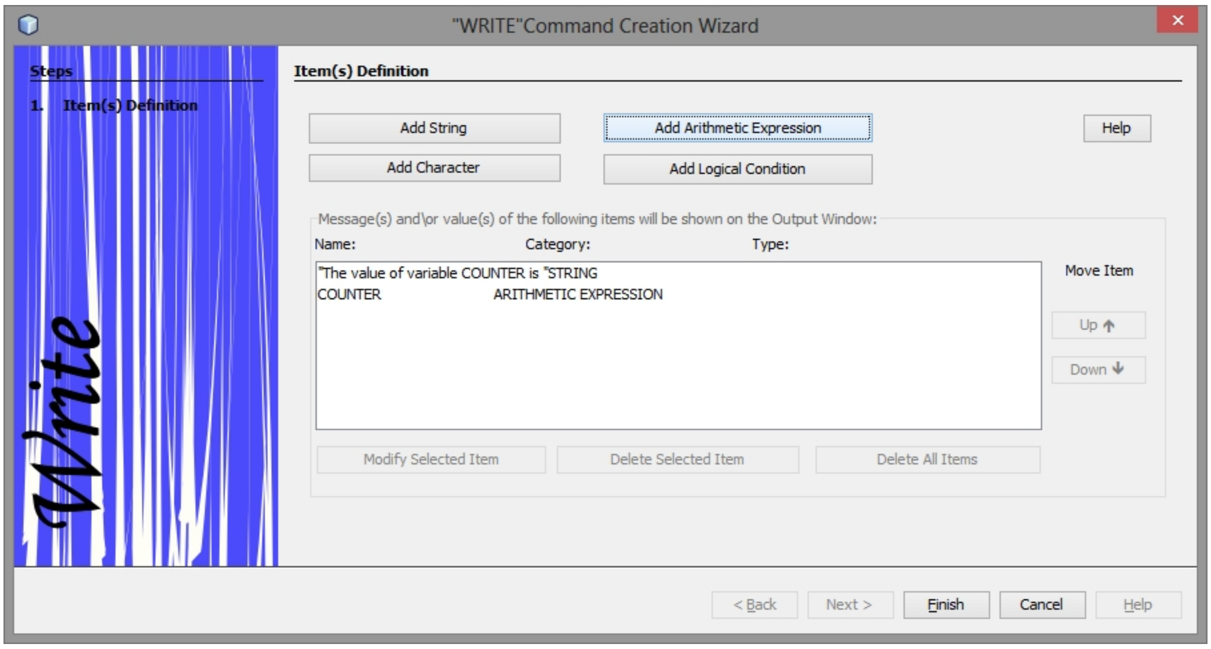


Figure 3.17. The first dialogue window of a write statement.

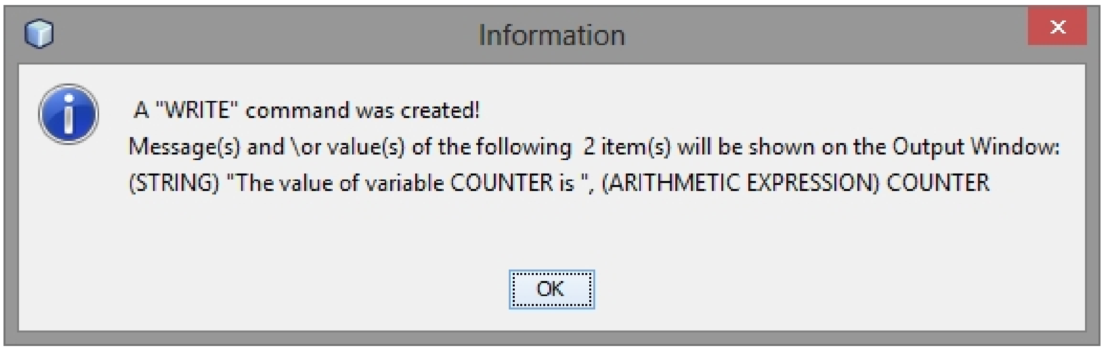
Because the value of COUNTER is an integer number the *Add Arithmetic Expression* button is clicked in order to include the variable COUNTER in this *write* statement. After adding the two items, the dialog window is updated and can be seen in Figure 3.18.



```
"The value of variable COUNTER is "STRING
COUNTER                   ARITHMETIC EXPRESSION
```

Figure 3.18. A dialogue window of a write statement with two items.

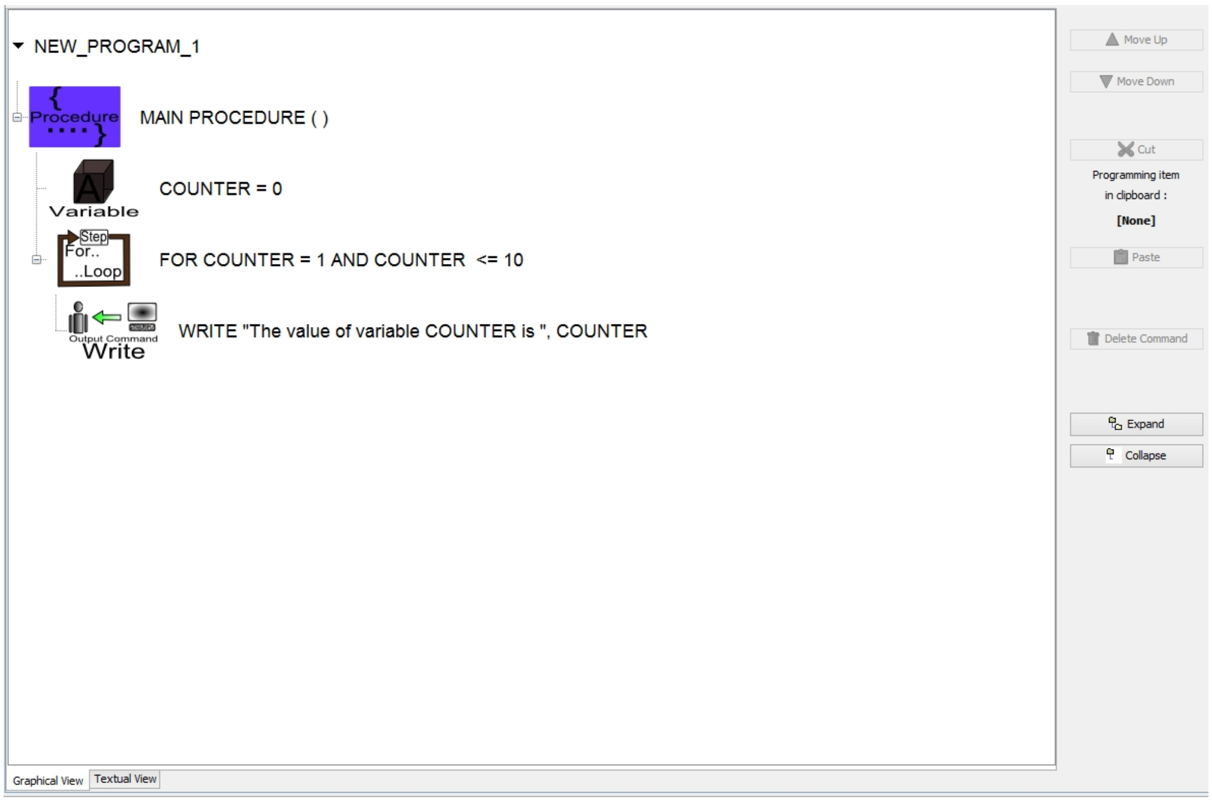
The order that the items are printed can change using the *Up* and *Down* buttons. Again, when the *Finish* button is clicked an information message regarding the addition of a *write* statement in the program appears (see Figure 3.19).



A "WRITE" command was created!
Message(s) and \or value(s) of the following 2 item(s) will be shown on the Output Window:
(STRING) "The value of variable COUNTER is ", (ARITHMETIC EXPRESSION) COUNTER

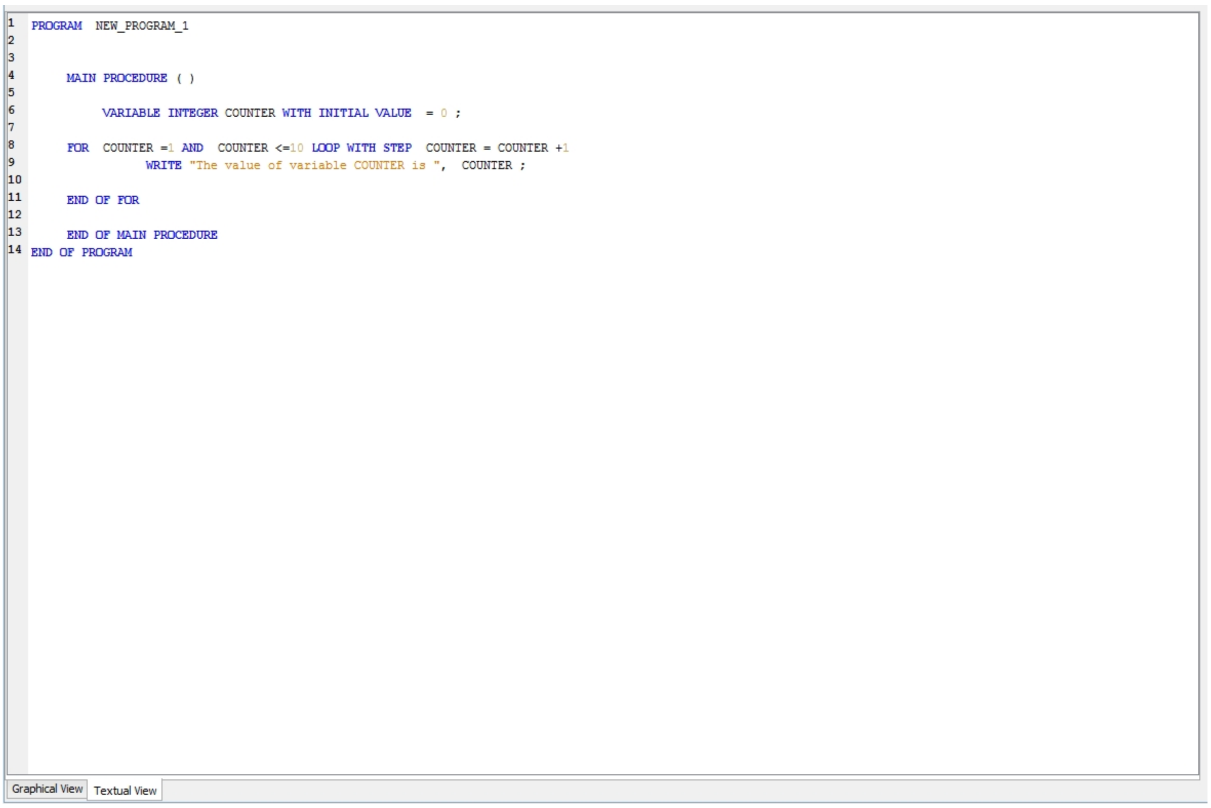
Figure 3.19. An information message for successfully adding a write statement.

The program is now ready. The graphical and textual view of this program can be seen in Figure 3.20 and Figure 3.21, respectively.



```
NEW_PROGRAM_1
  MAIN PROCEDURE ( )
    COUNTER = 0
    FOR COUNTER = 1 AND COUNTER  <= 10
      WRITE "The value of variable COUNTER is ", COUNTER
```

Figure 3.20. The graphical view of a program.



```
PROGRAM  NEW_PROGRAM_1


    MAIN PROCEDURE ( )

        VARIABLE INTEGER COUNTER WITH INITIAL VALUE  = 0 ;

    FOR  COUNTER =1 AND  COUNTER <=10 LOOP WITH STEP  COUNTER = COUNTER +1
             WRITE "The value of variable COUNTER is ",  COUNTER ;

    END OF FOR

    END OF MAIN PROCEDURE
END OF PROGRAM
```

Figure 3.21. The textual view of a program.

Figure 3.21 shows the text code that is automatically generated from the graphical view of the program. Different combination of colours and font styles are used for formatting the text in order to underline the different programming constructs. Table 3.3 presents the colour and font style that are used to represent each programming construct in Koios.

Table 3.3
Colour and Font Style of Each Programming Item

Programming Item	Colour	Font Style
Global constant	Green	Italics
Local constant	Black	Italics
Global variable/array	Green	Plain
Local variable/array	Black	Plain
Reserved word	Light blue	Bold
Name of function/procedure	Black	Bold
Number	Beige	Plain
Character/string	Dark orange	Plain
Boolean	Grey	Plain

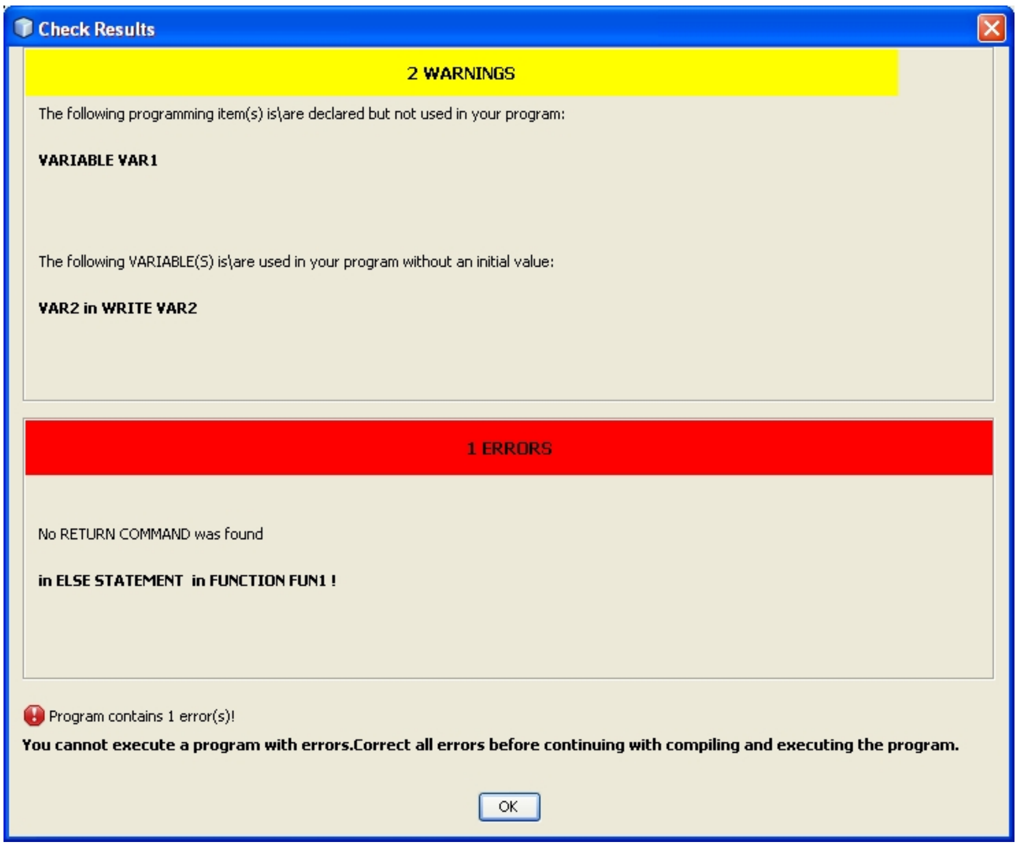
The data presented in Table 3.3 show that the same combination of font style and colour is used for both local variables and local arrays. This is also the case for the categories of global variables and arrays, functions and procedures, and characters and strings. The choice to represent local and global programming items of these categories using the same combination of font style and colour for each category was made so that the semantic relation between the items of each category could be denoted. However, the items of each category can be further distinguished by their attributes, for example a left square bracket always comes after the name of an array, but never after the name of variable. The combinations presented in Table 3.3 resemble those that are used in NetBeans.

Finally, the properties of a programming item can only be updated by double-clicking its node. This action initiates a dialogue between Koios and users, similar to the one that it is initiated during the creation of the item, which guides users through the update process. Users can update the properties of a programming item only by double-clicking on its node; this choice was made so they can correctly complete the update process through dialogue windows. The next section is concerned with programs' execution with Koios.

3.3.3 Executing Programs with the Koios Programming Environment

In order to demonstrate how programs are executed in the Koios programming environment, execution of the program (see Figure 3.20 and Figure 3.21) that was created in Section 3.3.2 is presented. This program displays the numbers from 1 to 10 using a *for*-statement.

In order to foster good programming habits in novices and to help them produce efficient programs, Koios provides a *Check* command. This command performs three actions on the current program. First, it reorders the nodes of constants, variables and arrays and moves them at the beginning of the procedure or function that are declared. Because declarations of these items are allowed everywhere in the program, this action guarantees that every item is declared before it is used. Second, it checks whether are there any constants, variables or arrays that are declared but not used and any variables or array elements that are used without initialisation. In the first case the name of the item is reported, while in the second the name of the item as well as the statement that includes it are reported, see Figure 3.22. Third, it detects any non-syntax errors. For a detailed presentation of the errors detected by Koios see Appendix B.4.



2 WARNINGS

The following programming item(s) is\are declared but not used in your program:

VARIABLE VAR1

The following VARIABLE(S) is\are used in your program without an initial value:

VAR2 in WRITE VAR2

1 ERRORS

No RETURN COMMAND was found

in ELSE STATEMENT in FUNCTION FUN1 !

Program contains 1 error(s)!

You cannot execute a program with errors.Correct all errors before continuing with compiling and executing the program.

Figure 3.22. A dialogue window with warnings produced by the Check command.

Users are advised to run the *Check* command during program creation and before compiling the program. However, if this action is omitted the *Compile* command automatically runs the *Check* command before the compilation of the program. When the *Check* or *Compile* command are executed the balloon in the status bar informs users whether warnings or/and errors exist or the program has no errors (see Figure 3.23 and Figure 3.24).

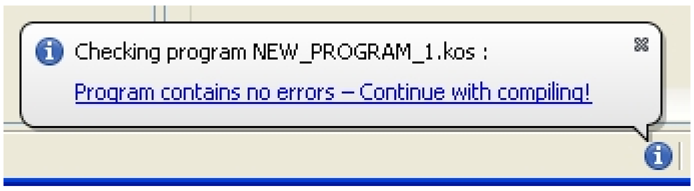

Checking program NEW_PROGRAM_1.kos :
Program contains no errors – Continue with compiling!

Figure 3.23. Message after running the Check command.

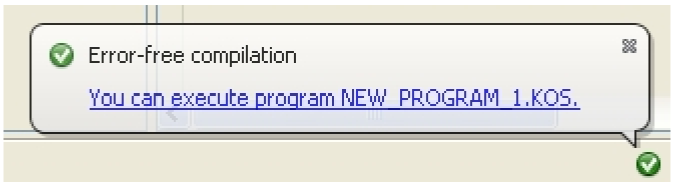

Error-free compilation
You can execute program NEW_PROGRAM_1.KOS.

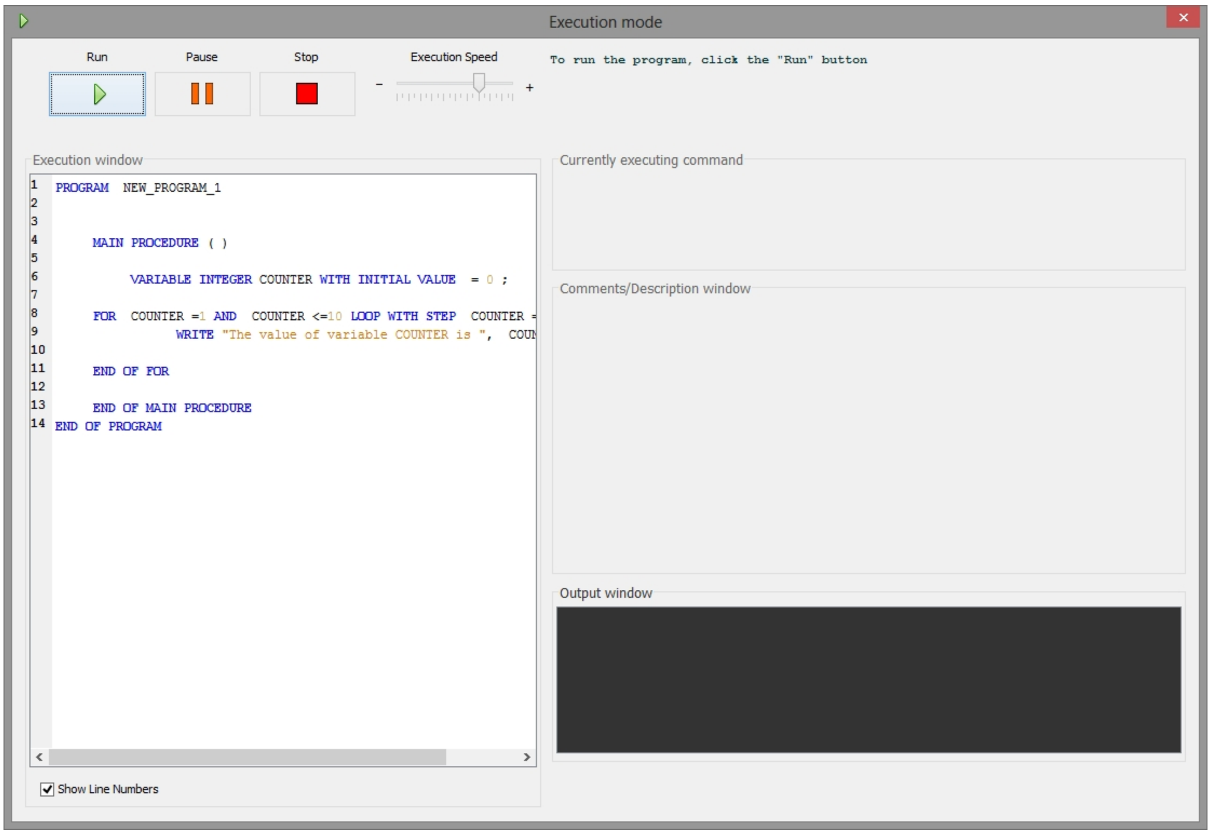
Figure 3.24. Message after running the Compile command.

An important decision regarding the development of the Koios programming environment was with respect to the implementation of its compiler. As discussed in Section 3.2.1.3, the design choice regarding the creation of programs with Koios was that programs would be created only by means of visual programming. Given that (a) programs created with Koios are always syntax-error-free and (b) the textual code is syntax-error-free as well - because it is automatically generated -, the necessity of compiler mechanisms that check for syntactic and grammatical correctness is questionable. In fact, what is required is the mechanism of compilers that produce executable code. However, this is a big research topic in itself. Therefore, and in order to avoid reinventing the wheel, the chosen solution was to use an existing compiler. Deciding which compiler would be appropriate though, is a matter that comes with a number of technical parameters. In order not to further complicate this matter, I came up with a simple but efficient solution. As explained in the beginning of Section 3.3, Koios can be executed only in a system that has the Java Virtual Machine installed. In fact, a system that has the Java Virtual Machine installed can execute any program in Java. Therefore, the solution was to transform the programs that are created with Koios to equivalent Java programs. Thus, every time the *Compile* command of Koios runs, the efficient and powerful compiler of Java is used to compile the programs that have been created with Koios and produce the executable files of these programs. These programs can in turn be executed within the Koios programming environment.

This choice was also made because it can familiarise novices with the advanced form of programs of professional PLs. Thus, Koios includes the *Export Java Program* command, which can produce the equivalent Java version of the programs created using Koios. Naturally, these Java programs can be viewed and executed using any Java programming environment.

Once a program contains no errors it can be executed by running the *Run* command. This action changes Koios' mode from Program Creation to Program Execution - a mode used for tracing programs' execution. The main window of Program Execution mode is shown in Figure 3.25.

At the top of the window there are control buttons for starting, pausing and stopping the execution and a slide bar for adjusting its speed. When the *Play* button is clicked the execution of the program begins. During execution the statement that is being executed is highlighted in yellow in the Execution window of the Program Execution mode.



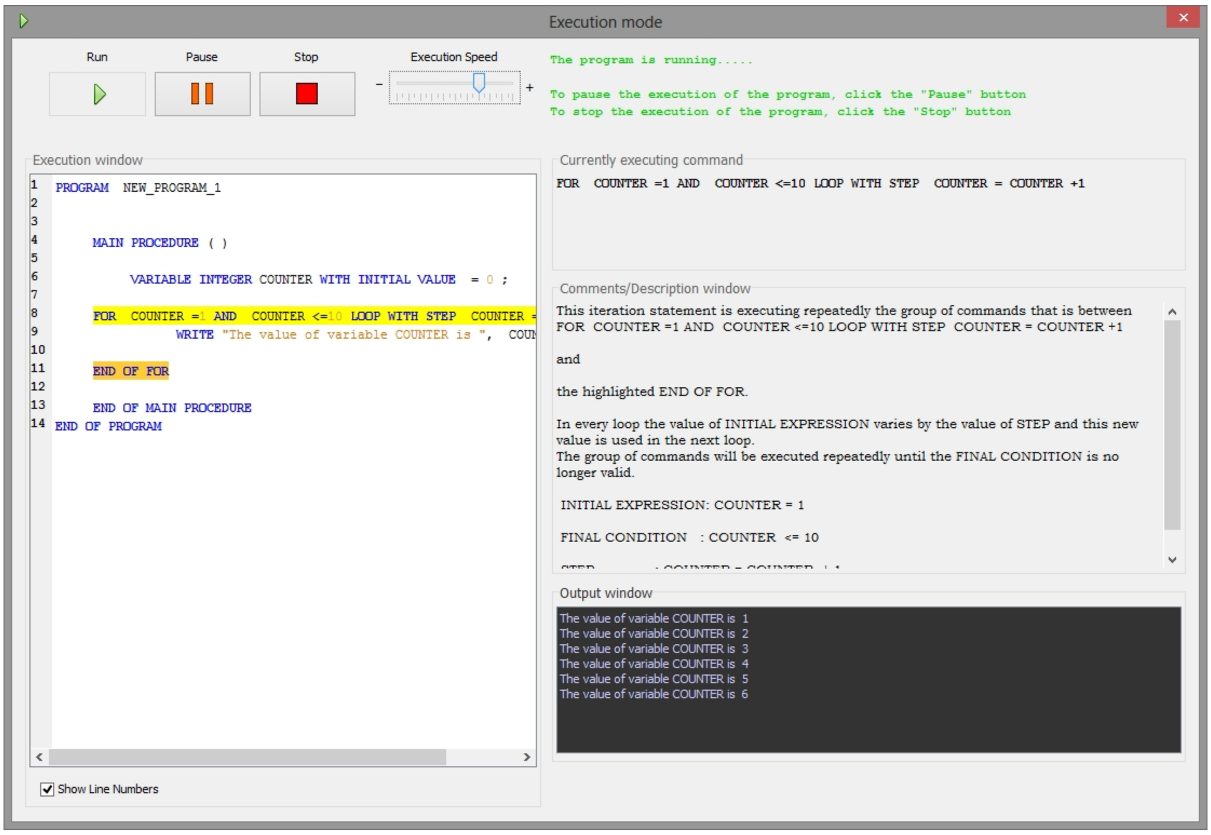
```
To run the program, click the "Run" button
```

```
PROGRAM  NEW_PROGRAM_1


     MAIN PROCEDURE ( )

          VARIABLE INTEGER COUNTER WITH INITIAL VALUE  = 0 ;

     FOR  COUNTER =1 AND  COUNTER <=10 LOOP WITH STEP  COUNTER =
                WRITE "The value of variable COUNTER is ",  COUN

     END OF FOR

     END OF MAIN PROCEDURE
END OF PROGRAM
```

Figure 3.25. A program in Program Execution mode.

In the case of a compound statement, for example a *for*-statement, the end statement is highlighted in orange (see Figure 3.26). The output of running programs is shown in the Output window of the Program Execution mode.

An important and unique feature of Koios is the Comments/Description window. This window provides messages with overall information about the statement that is being executed at the time. This information includes a brief description of its properties, how this statements works, what is the result of its execution and how it affects the control flow of the program (if applicable).



```
The program is running.....

To pause the execution of the program, click the "Pause" button
To stop the execution of the program, click the "Stop" button
```

```
PROGRAM  NEW_PROGRAM_1


     MAIN PROCEDURE ( )

          VARIABLE INTEGER COUNTER WITH INITIAL VALUE  = 0 ;

     FOR  COUNTER =1 AND  COUNTER <=10 LOOP WITH STEP  COUNTER =
                WRITE "The value of variable COUNTER is ",  COUN

     END OF FOR

     END OF MAIN PROCEDURE
END OF PROGRAM
```

```
FOR  COUNTER =1 AND  COUNTER <=10 LOOP WITH STEP  COUNTER = COUNTER +1
```

This iteration statement is executing repeatedly the group of commands that is between FOR COUNTER =1 AND COUNTER <=10 LOOP WITH STEP COUNTER = COUNTER +1

and

the highlighted END OF FOR.

In every loop the value of INITIAL EXPRESSION varies by the value of STEP and this new value is used in the next loop.
The group of commands will be executed repeatedly until the FINAL CONDITION is no longer valid.

INITIAL EXPRESSION: COUNTER = 1

FINAL CONDITION : COUNTER <= 10

```
The value of variable COUNTER is  1
The value of variable COUNTER is  2
The value of variable COUNTER is  3
The value of variable COUNTER is  4
The value of variable COUNTER is  5
The value of variable COUNTER is  6
```

Figure 3.26. A snapshot of the execution of a program.

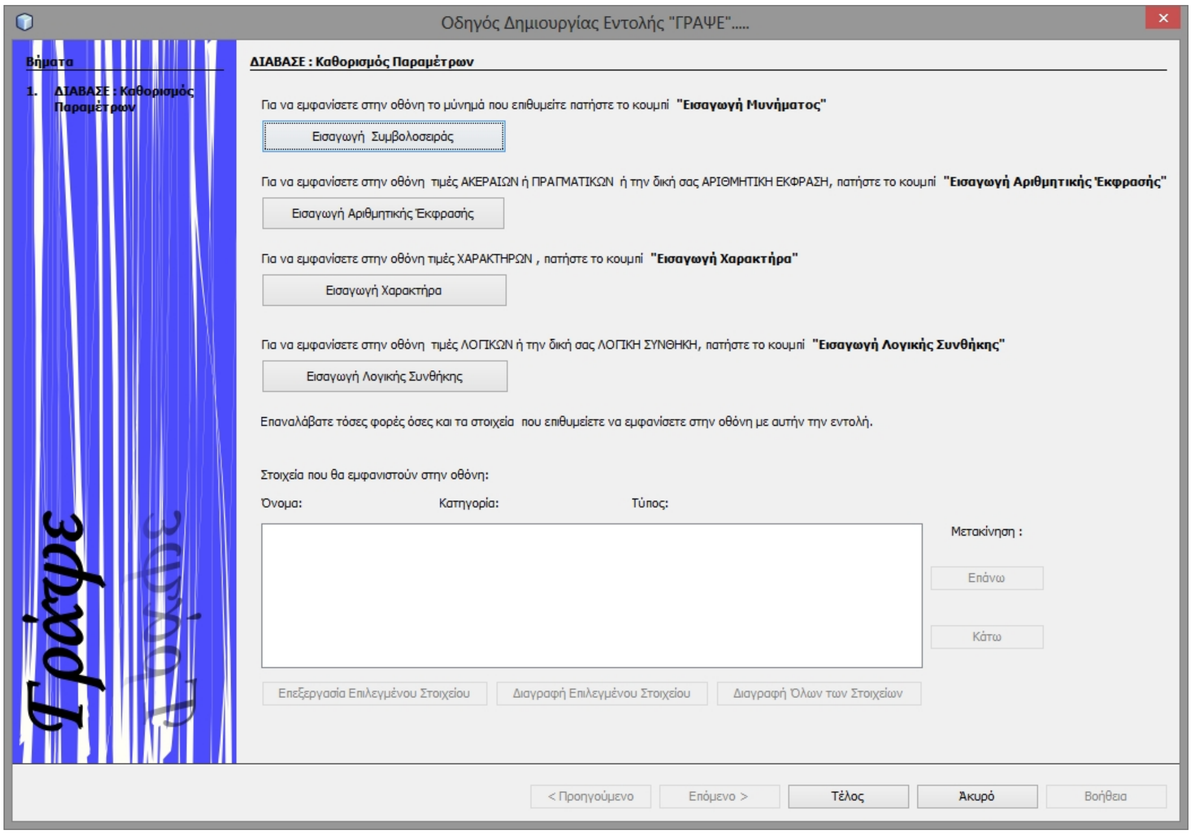
Thus, using the *Play*/*Pause* buttons and these messages users are facilitated in understanding and tracing the execution of the code in a constructivist way, at their own pace and with minimum help from instructors. A step-by-step execution of programs is not supported explicitly by this version of Koios; however, the *Play*/*Pause* buttons could be used to simulate this type of execution.

This section presented some important features of Koios through a case of creating and executing a simple program. These features were designed and developed with the primary aim of producing a programming environment that is based on the combined set of the seven design principles and thus could serve as an efficient tool for improving the teaching and learning of introductory programming. However, a secondary aim was that this tool could also serve as a stepping stone for the transition of novices to more advanced and powerful professional PLs.

3.4 Differences between Versions of Koios

As mentioned in Section 3.3, two Greek versions of Koios have been developed until today, namely Version 0 and Version 1. Both versions were used in data collection and more specifically, Version 0 in the first round of data collection and Version 1 in the second round (see Section 4.3). A discussion of the differences between the two versions follows.

Version 1 was the result of redesigning Version 0 based on students' comments from the first round of data collection (see Section 4.3.2.1) and suggestions made by the supervisory team. The improvements in Version 1 included (a) the replacement or redesign of a number of dialogue windows in order to make their use simpler, (b) the replacement, where appropriate, of words with symbols, (c) the inclusion of a *Help* button in dialog windows for creating/modifying statements, which would provide information regarding the functionality and the creation/modification process of statements, (d) the addition of tooltips to all buttons that inform users about them and (e) the visibility of buttons regarding constants, arrays, and functions only in Advanced User mode. These buttons are visible only in Advanced User mode because the programming items of constants, arrays, and functions are available only in this mode (see Section 3.3.2). Hence, in Simple Use mode there is no need to show buttons that do not provide any functionality. This way Koios' interface remains as simple as possible. It is important to note that a visual artist was involved in the redesign of these dialogue windows. He contributed to overall design matters. More specifically, he was consulted on matters regarding colour combination, font-size, type and colour, and spatial arrangement of items on dialogue windows. Two samples of the modifications made can be seen in Figure 3.27, Figure 3.28, Figure 3.29 and Figure 3.30. Although these dialogue windows are from the Greek versions of Koios, they can illustrate the rationale and results of modifications. Thus, the dialogue window for creating an output statement in Koios Version 0 and Version 1 can be seen in Figure 3.27 and Figure 3.28 , respectively. The new dialogue window (Figure 3.28) was intended to be simpler, more compact and pleasant to the eye than the old one (Figure 3.27). In particular, the following modifications took place (a) the buttons were grouped closer, (b) the text above each button in the old window was removed and incorporated in the new window as the tool tip of the specific button, (c) a help button was added, (d) symbols were added in the captions of the up and down buttons and (e) the font colour of parts of text was changed. Similarly, Figure 3.29 and Figure 3.30 show the main window during Program Execution mode in Koios Version 0 and Version 1, respectively.


Οδηγός Δημιουργίας Εντολής "ΓΡΑΨΕ".....

Βήματα

1. **ΔΙΑΒΑΣΕ : Καθορισμός Παραμέτρων**

ΔΙΑΒΑΣΕ : Καθορισμός Παραμέτρων

Για να εμφανίσετε στην οθόνη το μύνημά που επιθυμείτε πατήστε το κουμπί **"Εισαγωγή Μυνήματος"**

Για να εμφανίσετε στην οθόνη τιμές ΑΚΕΡΑΙΩΝ ή ΠΡΑΓΜΑΤΙΚΩΝ ή την δική σας ΑΡΙΘΜΗΤΙΚΗ ΕΚΦΡΑΣΗ, πατήστε το κουμπί **"Εισαγωγή Αριθμητικής Έκφρασής"**

Για να εμφανίσετε στην οθόνη τιμές ΧΑΡΑΚΤΗΡΩΝ , πατήστε το κουμπί **"Εισαγωγή Χαρακτήρα"**

Για να εμφανίσετε στην οθόνη τιμές ΛΟΓΙΚΩΝ ή την δική σας ΛΟΓΙΚΗ ΣΥΝΘΗΚΗ, πατήστε το κουμπί **"Εισαγωγή Λογικής Συνθήκης"**

Επαναλάβατε τόσες φορές όσες και τα στοιχεία που επιθυμείτε να εμφανίσετε στην οθόνη με αυτήν την εντολή.

Στοιχεία που θα εμφανιστούν στην οθόνη:

Όνομα: Κατηγορία: Τύπος:

Μετακίνηση :

Εισαγωγή Συμβολοσειράς
Εισαγωγή Αριθμητικής Έκφρασής
Εισαγωγή Χαρακτήρα
Εισαγωγή Λογικής Συνθήκης
Επάνω
Κάτω
Επεξεργασία Επιλεγμένου Στοιχείου
Διαγραφή Επιλεγμένου Στοιχείου
Διαγραφή Όλων των Στοιχείων
< Προηγούμενο
Επόμενο >
Τέλος
Άκυρό
Βοήθεια
Γράψε

Figure 3.27. The dialogue window for creating a write statement in Koios Version 0.

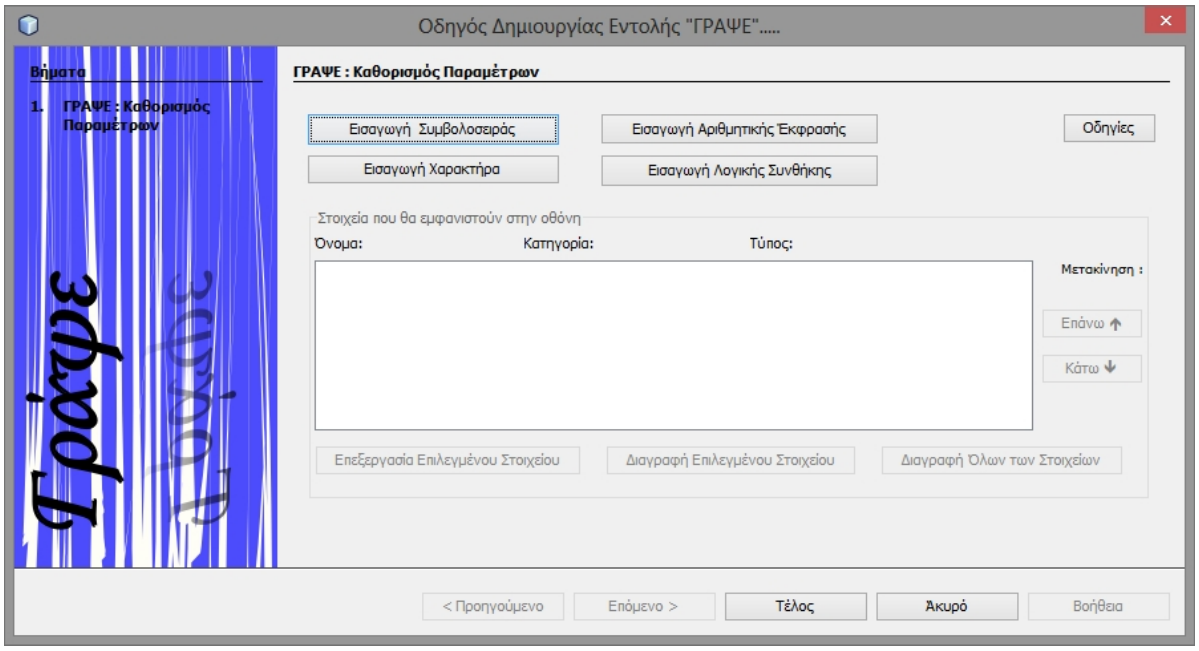
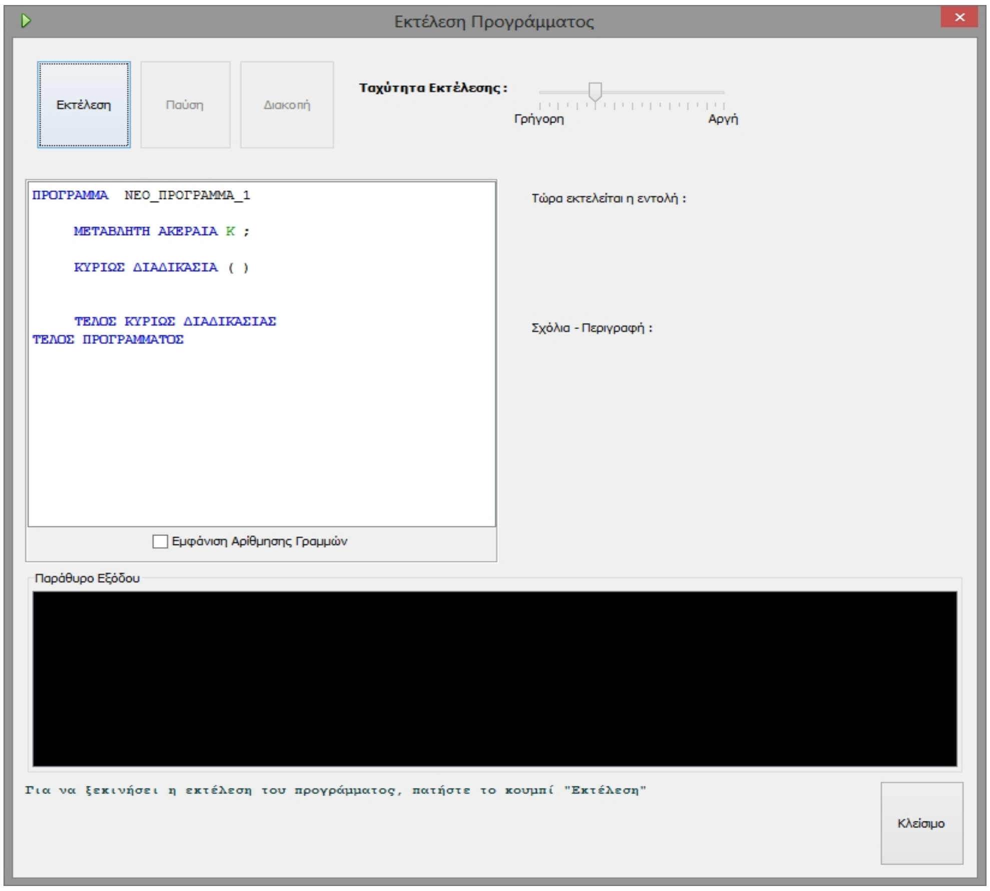

Οδηγός Δημιουργίας Εντολής "ΓΡΑΨΕ".....
Βήματα
1. ΓΡΑΨΕ : Καθορισμός Παραμέτρων
Γράψε
ΓΡΑΨΕ : Καθορισμός Παραμέτρων
Εισαγωγή Συμβολοσειράς
Εισαγωγή Αριθμητικής Έκφρασής
Οδηγίες
Εισαγωγή Χαρακτήρα
Εισαγωγή Λογικής Συνθήκης
Στοιχεία που θα εμφανιστούν στην οθόνη
Όνομα:
Κατηγορία:
Τύπος:
Μετακίνηση :
Επάνω
Κάτω
Επεξεργασία Επιλεγμένου Στοιχείου
Διαγραφή Επιλεγμένου Στοιχείου
Διαγραφή Όλων των Στοιχείων
< Προηγούμενο
Επόμενο >
Τέλος
Άκυρό
Βοήθεια

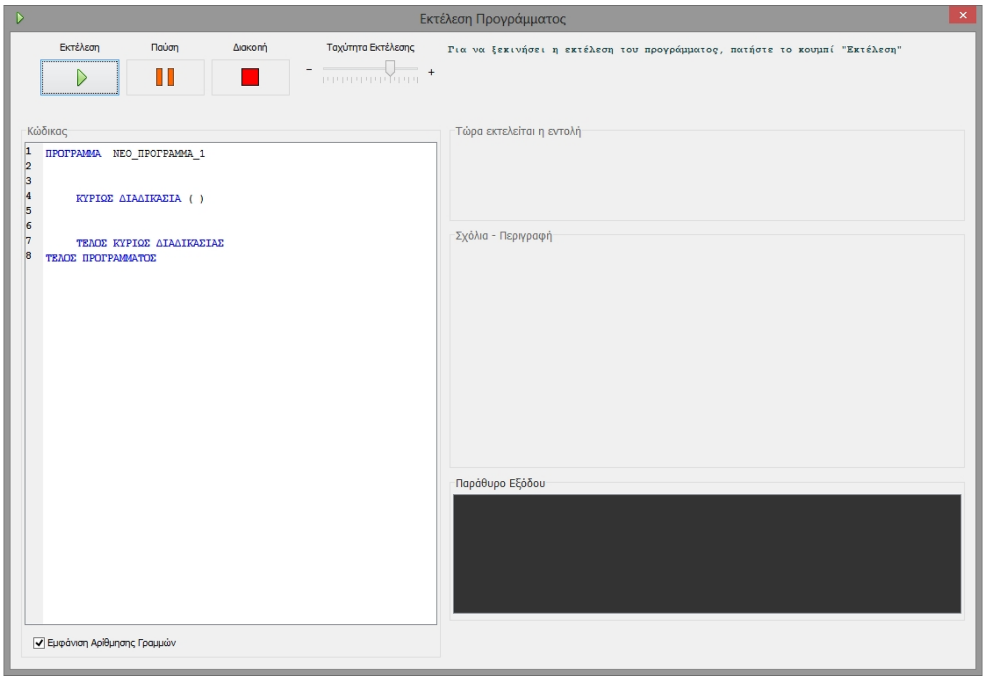
Figure 3.28. The dialogue window for creating a write statement in Koios Version 1.



```
ΠΡΟΓΡΑΜΜΑ  ΝΕΟ_ΠΡΟΓΡΑΜΜΑ_1

     ΜΕΤΑΒΛΗΤΗ ΑΚΕΡΑΙΑ Κ ;

     ΚΥΡΙΩΣ ΔΙΑΔΙΚΑΣΙΑ ( )


     ΤΕΛΟΣ ΚΥΡΙΩΣ ΔΙΑΔΙΚΑΣΙΑΣ
ΤΕΛΟΣ ΠΡΟΓΡΑΜΜΑΤΟΣ
```

```
Για να ξεκινήσει η εκτέλεση του προγράμματος, πατήστε το κουμπί "Εκτέλεση"
```

Figure 3.29. A program in Program Execution mode in Koios Version 0.



Για να ξεκινήσει η εκτέλεση του προγράμματος, πατήστε το κουμπί "Εκτέλεση"

```
ΠΡΟΓΡΑΜΜΑ  ΝΕΟ_ΠΡΟΓΡΑΜΜΑ_1


    ΚΥΡΙΩΣ ΔΙΑΔΙΚΑΣΙΑ ( )


    ΤΕΛΟΣ ΚΥΡΙΩΣ ΔΙΑΔΙΚΑΣΙΑΣ
ΤΕΛΟΣ ΠΡΟΓΡΑΜΜΑΤΟΣ
```

Figure 3.30. A program in Program Execution mode in Koios Version 1.

3.5 Conclusions

This chapter discussed the design and development of the Koios programming environment. Design and implementation issues regarding the set of the seven design principles as well as issues beyond this set were presented. The process of addressing these issues and the final decisions, based on which the interface and functionality of Koios was developed, were reported. In order to highlight important features of Koios that support and facilitate novices during the programming process, an example of creating and executing a program with Koios was demonstrated. The following chapter presents the evaluation of Koios in Greek educational institutes.

Chapter Four

Method, Data Collection and Data Analysis

Contents

4.1 Overview

The three primary aims of the research project reported in this thesis are (a) to identify a set of design principles for educational programming software, (b) to design and develop a programming tool as an embodiment of this combined set of principles and (c) to evaluate this new programming tool and assess its impact on novice programmers. The way that the first two aims were accomplished was discussed in the previous two chapters. In particular, Chapter Two reviewed the relevant literature and identified a set of seven design principles. The design and development of Koios, the programming tool that embodied this set was presented in Chapter Three. The fourth chapter of this thesis discusses the method that was employed to study the effect of programming tools' compliance with this set on novice programmers' knowledge and skills via the evaluation of the Koios programming environment. Furthermore, this chapter reports the process of collecting data for this study as well as the techniques that were employed for the analysis of the collected data.

4.2 Method

This section discusses the three research hypotheses and the research method followed to test them. First, the research hypotheses are presented, whilst the research designs that were produced to test them follow.

4.2.1 Research Hypotheses

As reported in Section 2.4.8, at the time this study was conducted, my profession was teaching CS subjects in Greek secondary schools and tertiary institutes. One of the subjects that I taught was introduction to programming. This motivated me to research the current state of international and, particularly, Greek introductory programming education and to identify a set of design principles for educational programming software. Using this combined set of the seven design principles (see Section 2.5), I designed and implemented Koios (see Chapter Three), a new programming tool that could help novices succeed in their first programming course. Hence, the following research question was crucial for the research reported in this thesis.

How important is compliance with this set of the seven design principles as a pedagogical consideration for educational software engineering?

The importance of compliance of educational programming tools with the combined set of design principles can be assessed from its pedagogical impact on their users. Naturally, the effect of compliance with a set of design principles on people cannot be directly measured. However, the effect of software that highly complies with this set on outcomes in its users could be used as an indirect measurement of the effect of compliance with the set. The pedagogical aim of an educational programming tool is to facilitate the acquisition of knowledge in the domain of programming. Thus, the expected pedagogical impact for novice programmers who use Koios is the acquisition of programming knowledge (see Sections 2.3). Knowledge in the programming domain can be categorised in declarative and procedural programming knowledge (see Section 2.7.1). More specifically, declarative programming knowledge can be represented by learners' mental models (see Sections 2.7.1 and 2.7.2.1), while procedural programming knowledge can be measured by the level of learners' programming skills (see Section 2.7.1 and 2.7.2.2). Therefore, the importance of compliance with the set of design principles can be determined by measuring the direct effect of Koios on the quality of mental models and programming skills of its users. Furthermore, the application of the mental-model hypothesis (Kellog & Breen, 1990) in the domain of programming instruction predicts that students with a rich mental model develop a higher level of programming skills than students with a poor mental model. In order to test the mental-model hypothesis, the existence of any indirect effect of Koios on users' programming skills, which could be mediated by their mental models should, therefore, be examined as well. Thus, to formally address the question presented in the beginning of this section, the following three research hypotheses are proposed.

H1: novice students who use a programming environment with a high level of compliance with the set of the seven design principles develop a higher level of programming skills than novice students who use programming environments that partially comply with this set.

H2: novice students who use a programming environment with a high level of compliance with the set of the seven design principles construct richer mental models in the domain of programming than novice students who use programming environments that partially comply with this set.

H3: the effect of programming environment on programming skills in novice students is mediated by the quality of their mental models.

4.2.2 Research Design

This section presents the research design that was produced in order to test the three research hypotheses. First a basic research design is presented. A number of reasons for modifying and refining the basic research design follow. Finally, the results of these refinements, two quasi-experimental designs, which were produced for conducting the study, are discussed.

4.2.2.1 Basic research design.

In order to test the three research hypotheses, the following basic research design was produced. The variable under investigation is compliance with the combined set of design principles. As discussed in the previous section, the effect of compliance with a set of design principles cannot be directly measured, but it can be measured by the effect that software, which was designed with a high level of compliance with this set, has on outcomes in its users.

Hence, to study the effect of compliance with the seven design principles, at least one programming tool with partial or no compliance and one with a high level of compliance with the set are required and need to be compared. Therefore, the independent variable (IV) of this design was the level of compliance with the set of design principles. The different levels of the independent variable (Pedhazur & Pedhazur Schmelkin, 1991) were programming tools with different levels of compliance with the set. Koios was designed as a programming tool with a high degree of compliance with the set of principles (see Chapter Three). Thus, Koios was assigned to the level of the independent variable that highly complies with the set of principles. The dependent variable (DV) (Pedhazur & Pedhazur Schmelkin, 1991) was the level of programming skills of novice programmers. The quality of novices' mental models was the mediator (M) (Baron & Kenny, 1986; MacKinnon, 2008). Figure 4.1 presents this basic research design graphically.

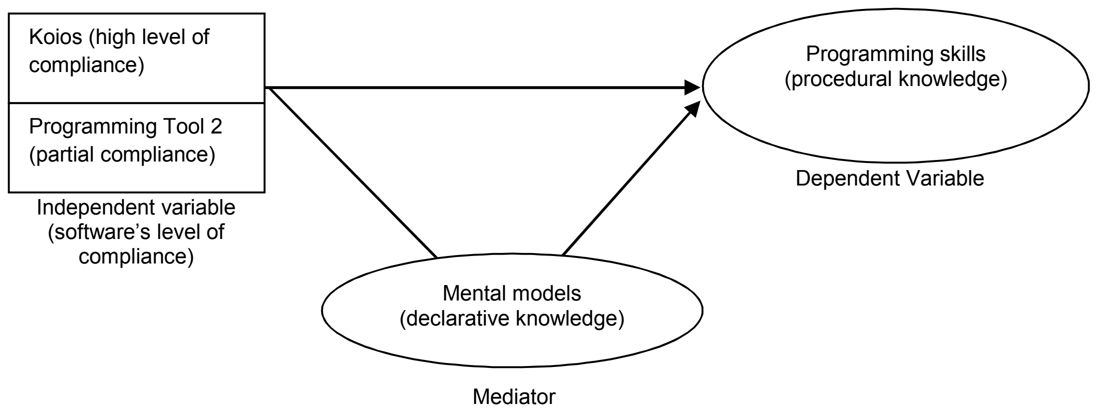

Koios (high level of compliance)
Programming Tool 2 (partial compliance)
Independent variable (software's level of compliance)
Programming skills (procedural knowledge)
Dependent Variable
Mental models (declarative knowledge)
Mediator

Figure 4.1. Basic research design.

The assignment of the Koios programming environment to a level of the independent variable not only allowed the testing of the three research hypotheses, but it also enabled (a) Koios' evaluation as a programming tool, (b) the assessment of its impact on novices and (c) its comparison with 'standard' software that is already used for teaching programming. This was accomplished by planning and conducting an empirical study based on the basic research design. Hence, this basic research design was further refined in two quasi-experimental designs in order to meet the emerging requirements of the empirical study.

4.2.2.2 Refinement of the basic research design.

The reasons for and details of the refinement of the basic research design are as follows. First, the evaluation of Koios was planned to take place in Greek educational institutes. A first reason for this decision was that Koios had specifically been designed to support Greek novice programmers, as, in my experience, there was a lack of suitable tools for this purpose at the time the research that is reported in this thesis started. A second reason was that I was certified to teach students of Greek secondary and tertiary institutes because my profession was teaching CS subjects in Greek educational institutes. This population of students was identified as appropriate for providing the sample of research participants that could be involved in the evaluation of Koios. Thus, I was more likely to get approval for evaluating Koios in Greek institutes than in other international institutes.

Second, for the empirical study and Koios' evaluation, I decided to use quasi-experimental designs (Pedhazur & Pedhazur Schmelkin, 1991) instead of traditional experimental designs, because random assignment of participants in experimental groups was not practically possible. In order to increase the probability of an educational institute accepting to participate in this study, Koios' evaluation had to cause as less interference as possible with its programme of CS studies. This included working with the class-groups of students that were predefined by the administration services of the institute. Thus, a random allocation of students in experimental groups was not possible.

Third, Koios was developed as a programming tool for the initial stages of teaching and learning programming in both secondary and tertiary level. However, a difference in programming knowledge between students of secondary and tertiary education was expected. The first reason for this is that the curriculum of Greek secondary education includes fewer programming concepts than the majority of curricula of Greek tertiary institutes. The second reason is that the teaching period for introductory programming courses in secondary and tertiary education is on average at least 12 and 14 weeks, respectively. Finally, the minimum number of lectures and laboratory courses for introductory programming in tertiary institutes is two and two per week, respectively and their duration is 45 minutes. In contrast, in secondary schools, there is only one lesson per week, which ranges from 35 to 45 minutes. All these factors suggested that the development of programming skills in secondary and tertiary education would be different. More specifically, students in tertiary education were expected to develop richer declarative and procedural programming knowledge.

Fourth, this empirical study was based on the CS curricula of participating institutes and incorporated in actual lessons of introductory programming. The aim of this decision was twofold. The first and more important aim was that, this way, the study took place in ‘real’ learning environments. The second aim, as explained in a previous paragraph, was to comply as much as possible with the programme of CS studies of educational institutes, in order to facilitate their participation in this study. The decision of this study taking place in Greek secondary and tertiary institutes presented no particular methodological problems or drawbacks for the following reasons. First, all secondary schools in Greece follow the same CS curriculum, which is dictated by the Greek Ministry of Education. Second, tertiary institutes in Greece can form their own CS curriculum, but in practice, their curricula for introductory programming include all programming concepts under investigation by this study. The implications of this decision with respect to conducting this study in secondary schools were that, according to the official CS curriculum, (a) the duration of a quasi-experiment should be at least 12 weeks of lessons, (b) the duration of each lesson would be determined by the official timetable, which is the same for all secondary schools and ranges from 35 to 45 minutes, and (c) a specific set of programming concepts (see Section 3.3.2) had to be taught. The implications of this decision regarding this study in tertiary-education institutes were that (a) its duration should be between 14 and 16 weeks, (b) a minimum of two lectures and two laboratories courses should take place per week and (c) each lecture/laboratory course should last for 45 minutes.

Therefore, in order to test the three research hypotheses and evaluate Koios two quasi-experimental designs were produced. The first quasi-experimental design was used to study the effects of compliance with the set of principles, via the use of Koios, on the declarative and procedural programming knowledge of pupils that were studying in Greek secondary schools. Accordingly, the second quasi-experimental design was going to be used to study the effects of compliance with these principles, via the use of Koios, on the declarative and procedural programming knowledge of students that were studying in Greek tertiary institutions. A presentation of the two quasi-experimental designs follows.

4.2.2.3 Method of first quasi-experiment – secondary education.

The first quasi-experimental design was used for evaluating Koios and testing the three research hypotheses with secondary-school students. First, the software that was included in this quasi-experimental design is briefly introduced, followed by the details of this design.

At the time the research reported in this thesis started, there were two popular programming environments in Greek secondary-level education that were used for teaching introductory programming: Glossomatheia (Nikolaidhs, 2008) and MicroworldsPro (LCSI, 2008). Glossomatheia was a programming tool that was widely used in secondary schools for teaching introductory programming. It supports the procedural paradigm and Greek Pascal-like syntax and semantics. The second programming tool, MicroworldsPro, was introduced in 2009 by the Greek Ministry of Education as the official recommendation for teaching introductory programming. MicroworldsPro is a LOGO-based programming environment. The Greek version of MicroworldsPro was used in the experiment. LOGO (Papert, 1980) was one of the first PLs that was specially designed for children with no programming experience (McIver, 2000). However, the creators of LOGO did not give special consideration to its linguistic attributes (Murnane, 1993) and its importance has been decreasing over the years, mostly because of its design, which seems to target only infant users (Grandell et al., 2005). Glossomatheia and MicroworldsPro were considered to partially comply with the set of the seven design principles. For more details about how the compliance of the programming tools with the design principles was evaluated see Appendix C.1.

Design. The first quasi-experimental design was based on the basic research design that was discussed in the previous section (Section 4.2.2.1). The independent variable was the level of compliance with the set of the seven design principles and had three levels, which varied in their degree of compliance. Koios represented the level of the independent variable with high compliance, while Glossomatheia and MicroworldsPro represented each of the two levels of the independent variable with partial compliance. The different levels of compliance of these tools with the set of design principles are presented in Table 4.1.
The data presented in Table 4.1 show that Glossomatheia and MicroworldsPro fully comply with the first and fifth design principles, namely they are in the Greek language and provide a small set of instructions. In contrast, the error messages provided by MicroworldsPro were designed with a low level of compliance with the sixth design principle, while Glossomatheia partially complies with this principle. The two programming tools partially comply with the remaining four design principles.

Table 4.1
Satisfaction Levels of Design Principles (Independent Variable) for the Four Programming Environments Used in the Quasi-Experimental Designs

	Design Principle						
Name	Natural Language of PL and IDE	Syntax and Semantics	Visualisation	Level of Abstraction	Small Set of Instructions	Provision of Error Messages	Level of Interaction with the IDE
Koios	√	√	√	√	√	√	√
Glossomatheia	√	½	½	½	√	½	½
MicroworldsPro	√	½	½	√	√	X	½
C	X	X	X	½	X	X	X

Note. Koios was used by secondary-school pupils and was going to be used by university students. Glossomatheia and MicroworldsPro were used only by secondary-school pupils. C was going to be used only by university students.

As was previously discussed, Koios was designed with a high level of compliance with the seven design principles. Finally, C was designed with a low level of compliance with the set of design principles, with the exception of partially complying with the design principle regarding the level of commands' abstraction.

The level of students' programming skills (procedural knowledge) was the dependent variable. The quality of pupils' mental models (declarative knowledge) was the mediator. *Participants' GPA without CS*, *Participants' GPA from previous year*, *Gender*, *Day of Week*, *Hour of Day*, *Size*, *Gaps between Lessons*, *Homework Frequency* and *Duration of Lesson* were variables that were considered for candidate covariates and tested for possibly affecting the effect of independent variable on dependent variable (see Section 4.4.1.4).

Participants. Participants were third-grade secondary-school students with no previous formal instruction in programming. Students' demographic details are presented in Section 4.3.1.2 and 4.3.2.2 . Statistical power analysis revealed that a total of 66 secondary-school pupils were required in order to detect a large effect size (η^2= 0.138) with a significance level of 0.05 and a statistical power of 0.80.

Procedure. Each independent-variable level, namely each programming tool, corresponded with one quasi-experimental group of secondary-school students. Based on the results of statistical power analysis, each of the three quasi-experimental groups should consist of at least 22 participants – students. Each group used the corresponding programming tool for its introductory programming course for nine lessons (12 weeks including three knowledge measurements, see Section 4.2.2.2). The programming concepts, teaching method, classroom examples, homework and teaching material were the same for all experimental groups. However, there was a necessary slight variation in the teaching material for each of the three groups in order to appropriately present the corresponding programming tool.

The validity of the hypotheses H1 and H2 was tested via procedural-knowledge and declarative-knowledge measurements, respectively. The programming knowledge of the following 13 programming concepts that were taught was measured: *integer, arithmetic expression, string, output-statement, input-statement, variable, assignment-statement, iteration-statement, procedure, procedure call, input parameter, logical condition* and *conditional statement*. The knowledge of these concepts is considered to provide a basic understanding of introductory programming, at least at the level of lower secondary-education. Three declarative-knowledge measurements of students' mental models took place. These measurements were scheduled for the beginning, the middle (after five lessons since the beginning) and the end (after nine lessons since the beginning) of the quasi-experiment, respectively. Although it was expected that students would not possess declarative programming knowledge in the beginning of the experiment, the three measurements used the same set of pairs of programming concepts so that the development of the students' mental models could be monitored throughout the three mental models' measurements.

The measurement of students' programming skills was planned as well. Three procedural-knowledge measurements were scheduled and administered after three, five and nine lessons since the beginning of the quasi-experiment, respectively. A procedural-knowledge measurement was not scheduled for the beginning of the study, because it was expected that students would not possess procedural programming knowledge. Instead, the first procedural-knowledge measurement took place after three lessons, when, in principle, a low level of procedural programming knowledge should have been developed. The development of procedural-knowledge was not monitored in the same way as the development of declarative knowledge, because different items had to be used in the three procedural-knowledge tests, in order to detect the level of students' programming skills. Nevertheless, the second measurement of declarative and procedural programming knowledge took place successively in the same lesson. This was also the case for the third measurement of declarative and procedural programming knowledge. Measuring procedural and declarative knowledge in the same lesson allowed for testing hypothesis H3. This hypothesis examines whether programming tool has an indirect effect on students' programming skills by mediation of their mental models.

Thus, the collection of data regarding students' programming skills and mental models, at the same occasion, facilitated the testing of this hypothesis' validity.

Material. Students' mental models were measured using ratings of concept pairs (see Section 2.7.2.2). The programming skills of students were assessed with procedural-knowledge tests (see Section 2.7.2.3). More details about this material are presented in the section regarding data collection (see Section 4.3).

Data Analysis. For the analysis of collected data, correlation analysis, analysis of variance, simple and hierarchical multiple regression (Field, 2009; Pedhazur, 1997; Pedhazur & Pedhazur Schmelkin, 1991) were employed. These techniques were used for testing the direct and indirect effect of the independent variable on the quality of students' mental models and level of programming skills. The Pathfinder scaling algorithm (see Section 2.7.2.2) was used to compare novices' and experts' mental models in order to determine the quality of students' mental models (Cooke & Schvaneveldt, 1988). Mediation analysis (Baron & Kenny, 1986; MacKinnon, 2008) was used to test the indirect effect of mental models on programming skills. Assessments of homework, grades from other relevant subjects (for example mathematics and physics), and grade point average were used as covariates. More details about the data analysis techniques that were employed can be found in the section that discusses data analysis (see Section 4.4).

4.2.2.4 Method of second quasi-experiment – tertiary education.

The first quasi-experiment was produced with respect to studying the effect of compliance with the set of principles on secondary-education students. The second quasi-experiment aimed to study the same effect on tertiary-education students. As with the case of the first quasi-experiment, this design is a refinement of the basic research design, which was discussed in Section 4.2.2.1, in order to meet the requirements of the empirical study. Hence, this design has a number of similarities with the first quasi-experimental design. Because the details of the first design were thoroughly discussed in the previous section (see Section 4.2.2.3), this section mostly focuses on the rationale, issues and details that differentiated this quasi-experimental design from the first one.

Design. The independent variable of this design was the level of compliance with the set of the seven design principles as well. However, in this design, it had only two levels that varied in their degree of compliance. Koios was the level of the independent variable with high compliance, while C (Kernighan & Ritchie, 1988) was the level of the independent variable with partial compliance. C together with Java (Gosling & McGilton, 1996) are two general-purpose, powerful, professional and popular PLs. Moreover, they are the two most widely used PLs for introductory programming courses in tertiary education worldwide. Although both of these PLs could be used in this design, it was decided to include only C. Java supports the OO programming paradigm, while C the procedural paradigm. For reasons that were discussed in Section 3.2.2.1, the procedural paradigm is preferred over the OO paradigm for novices. Thus, Java was excluded and C was the only level of the independent variable with partial compliance. Glossomatheia and MicroworldsPro were excluded from this design as well. MicroworldsPro is considered a PL for children and not complex enough for tertiary education (Grandell et al., 2005; Jenkins, 2002; McIver, 2000). Glossomatheia is not totally unsuitable for tertiary programming courses, but professional PLs, such as Java and C, are usually preferred for the majority of these courses. In any case, Glossomatheia and MicroworldsPro were already used in the first quasi-experimental design and thus, their inclusion in this design as well, was deemed not necessary. Table 4.1 presents the different levels of compliance of Koios and C with the set of design principles. As previously reported, Koios was designed and developed with a high level of compliance with the set of design principles. Contrary, C seems to have a low level of compliance with this set, with the exception of the level of commands' abstraction. Appendix C.1 provides more details about how the compliance of programming tools with the set of principles was evaluated. As in the first quasi-experiment, the level of students' programming skills (procedural knowledge) was the dependent variable, while the quality of their mental models (declarative knowledge) was the mediator.

Participants. Participants were going to be university students, who had no previous programming experience at a tertiary level. Statistical power analysis revealed that a total of 44 university students were required in order to detect a large effect size (η^2= 0.138) with a significance level of 0.05 and statistical power of 0.80.

Procedure. As in the first quasi-experiment, each independent-variable level, namely each programming tool, was going to correspond with a quasi-experimental group. Based on the results of power analysis, each quasi-experimental group was going to consist of at least 22 university students. Each group was going to use its corresponding programming tool for the first half of the winter semester. In the second half of the same semester, both groups were going to use C as their programming tool, in a follow-up study. This way two additional aims were going to be accomplished. First, the pedagogical impact of Koios and C as introductory PLs was going to be assessed. Furthermore, the comparison between the pedagogical impact of Koios and C could provide evidence about the superiority of one programming tool over the other, regarding their suitability as programming tools for tertiary introductory-programming courses. Second, as novices progress, the use of professional, non-educational PLs, such as C, becomes a necessity. The degree of preparation, which is provided by the use of Koios, for this transition could be assessed by this quasi-experimental design as well.

As in the case of the first design, declarative-knowledge and procedural-knowledge measurements were planned for testing hypotheses H2 and H1, respectively. The knowledge of 19 programming concepts was going to be measured in this design: *integer, arithmetic expression, string, output-statement, input-statement, variable, array, array index, assignment-statement, iteration-statement, procedure, procedure call, function, function call, input parameter, return-statement, return argument, logical condition* and *conditional statement*. The concepts in this set are considered to provide a fundamental understanding of introductory programming at tertiary level. However, increasing the size of this set would result in an unrealistic duration of declarative-knowledge measurement, due of the increasing numbers of possible concept pairs (see Section 2.7.2.2). Again, three declarative-knowledge measurements were going to take place. The first one was scheduled for the beginning of the semester, before any experimental manipulation. The second one was scheduled for the middle of the semester, before both groups were going to use C. The third one was scheduled for the end of the semester. Three procedural-knowledge measurements were planned as well. The three measurements were scheduled for the beginning, the middle and the end of the semester, respectively. In this design, the first procedural-knowledge measurement was scheduled for the beginning of the semester, because university students were expected to have some previous programming experience from secondary education. In each of the three measurements, the declarative-knowledge and procedural-knowledge measurement were going to take place successively. Thus, the data collected in each measurement were going to be used for testing hypothesis H3 (see Section 4.2.2.3). Feedback provided at the end of the winter semester was going to be used to create an improved version of Koios. This design was planned to be repeated in the spring semester, with one variation; the improved version of Koios was going to be the independent-variable level with high compliance.

Material. As in the case of the first design, the quality of students' mental models was going to be measured using ratings of pair concepts (see Section2.7.2.2). Similarly, the level of students' programming skills was going to be assessed with procedural-knowledge tests (see Section 2.7.2.3).

Data Analysis. The analysis techniques used in the first design were going to be used for this design as well.

For reasons that are explained in the next section, the second quasi-experiment was eventually not conducted. Only the first quasi-experiment was conducted. However, both quasi-experiments received research ethics approval by Teesside University's Research Ethics Committee. The next section discusses how this design was implemented in the data collection procedure.

4.3 Data Collection

This section reports matters regarding the collection of data that was conducted based on the first quasi-experimental design. Data collection consisted of two rounds. The second round was added in the data-collection scheme, in order to provide further evidence to test the three research hypotheses. Section 4.3.1 presents the first round of data collection, while Section 4.3.2 presents the second round. The differences between the two rounds are reported in Section 4.3.2.1.

4.3.1 First Round of Data Collection

4.3.1.1 Preparations and arrangements for the first round of data collection.

Initially, I contacted several secondary and tertiary institutes to discuss their participation in this study. Staff from a high-school in the city of Lefkas and two tertiary institutes, the Technical Educational Institute of Patras and the Technical Educational Institute of Lefkas, were contacted and agreed to participate. However, data were collected only in the secondary school for the following reasons.
A formal letter from the Director of Studies for conducting the study was requested by the tertiary institute in Patras. The Director of Studies sent the letter, but a reply was never returned and thus data collection could not be conducted in this institute. In the case of the tertiary institute in Lefkas, three students volunteered to take part in the study. Obviously, this sample size (*N* = 3) would yield results with low statistical power. Therefore, this experiment had to be cancelled and thus, no data collection was conducted in tertiary institutes during the first round.

4.3.1.2 First round of data collection in secondary education.

The first round of data collection began in the 23rd of November 2009 and ended in the 07th of May 2010. A total of 72 third-grade participants – 41 male and 31 female students – were informed about the study before its beginning. Participants lived in the wider area of Lefkas city and their average age was 14 years. Due to participants' age, a consent form (see Appendix C.2), together with information about the study (see Appendix C.3), were given to be signed by each participant's parent(s)/guardian(s) before the study.

The school had four classes in the third grade, namely G1, G2, G3 and G4 and each class had 22, 23, 17 and 10 students, respectively. According to the directions of the Greek Ministry of Education for CS lessons in high-schools, each class is further divided in two subclasses - groups. Thus, each class was divided in two, forming eight school-groups in total, namely G1A, G1B, G2A, G2B, G3A, G3B G4A and G1B. In order to follow the quasi-experimental design for secondary education, the school-groups G1A and G1B, G2A and G2B, and G3A, G3B, G4A and G4B used Glossomatheia, the Koios prototype (Version 0) and MicroworldsPro, respectively, as their programming tool. The exact day and school hour that CS lessons took place for each school-group were scheduled by the administration of the school.

During the study, the absence of students, remarks and comments made by students and any gaps between lessons were recorded. Each school-group had a school hour of CS lesson per week. If for some reason, for example a school trip, a school-group missed its lesson, then this was recorded as a gap between lessons. The particular lesson was not actually missed – it was held later in the schedule – rather the opportunity to be held on the particular occasion was missed.

In order to minimise the differentiation of conditions between the experimental groups, I decided to give all the lessons for the eight school-groups myself. This decision could ascertain that the teaching material, classroom examples, homework and teaching method would remain as similar as possible for all school-groups. Moreover, having an active role in the learning environment could make my presence in the classroom seem more ‘natural’ than just being a passive observer. However, this decision could increase bias because the person giving the lessons and conducting the experiment was the same. Bias that could be introduced by this decision as well as the efforts made to decrease potential bias in this research project are discussed in Section 6.2.3.

The study was conducted according to a plan for lessons and knowledge measurements that I prepared and can be found in Appendix C.4. This plan was based on the official CS curriculum for third grade of High School in Greece as well as on the quasi-experimental design for secondary education. I also prepared the teaching material for the lessons for all groups.

The teaching material for each lesson included (a) some theoretical notes regarding the programming concepts in discussion, (b) how this programming concept is represented by a programming statement in a programming tool, (c) one or more programs in which the programming statement is used, as classroom example(s) and (d) homework. The only necessary difference in teaching material between groups was the information regarding the programming tool that each group was using. For example, the same theoretical notes, examples and homework regarding variables were used for all groups. However, the demonstration of a variable declaration was different for each group according to the programming tool in use.

The classroom examples that were used in the first round of data collection can be found in Appendix C.5. These examples were based on directions and suggestions from the Greek CS student's book for the third grade (Pedagogical Institute, 2004). Similarly, homework assignments of the first round of data collection can be found in Appendix C.6. Homework assignments were given at the end of each lesson, were not compulsory and aimed to engage students in further reflection on programming concepts. Each assignment was based on the example program that was demonstrated during the lesson in classroom (see Appendix C.5). In each homework assignment, students were asked to complete a program using either natural language or the commands of the programming tool that they were using in classroom. The first commands of each program were provided in order to encourage students and avoid the often-cited problem of not knowing how to begin with program creation. The provided commands, however, were not crucial for completing the program, thus the key programming concepts had to be completed by students. Students were asked to complete these programs using only paper and pencil. This decision was made because it was expected that many students may not have access to a computer with 'their' programming tool installed at home and school's computer lab was available only during lessons. Because students could use help to complete programs at home, the mark of their homework was not considered as a measure of students' performance, but only as a gauge of their motivation. Each assignment was due for submission at the beginning of the following lesson. However, overdue assignments were accepted, because they were only used to measure motivation. The submitted assignments were corrected by me and returned to students with comments. I noted that almost all students who returned homework assignments preferred to use programming-language commands rather than natural language.

It is important to note that the material for lessons as well as for procedural-knowledge measurements that are presented in this thesis are in a generic form. This means that the actual material used in the study was further modified to be consistent with the programming tool of each experimental group. The discussion of declarative-knowledge and procedural-knowledge data collection follow.

4.3.1.2.1 Declarative-knowledge data collection (first round).

Based on the first quasi-experimental design (see Section 4.2.2.3), participants' declarative knowledge (mental models) of 13 programming concepts: *integer, arithmetic expression, string, output-statement, input-statement, variable, assignment-statement, iteration-statement, procedure, procedure call, input parameter, logical condition* and *conditional statement* was measured. Three declarative-knowledge measurements took place during the first round of data collection. The three measurements were scheduled for the beginning, after five lessons and at end of the study, respectively, and ratings of pairs of concepts were used as a declarative-knowledge elicitation method (see Section 2.7.2.2). Students' declarative knowledge of all 13 concepts was measured in every occasion, although in the first and second measurement students had not been taught all 13 concepts. Because the assessment of students' mental models and the monitoring of their development required that the same baseline – experts' mental models – had to be used to interpret every measurement, the baseline should incorporate all 13 concepts. Therefore, in order to make meaningful and valid comparisons between students' and experts mental models, students' mental models of all 13 concepts had to be measured in every declarative-knowledge measurement.

In each measurement, a randomised set of all possible pairs of the 13 concepts (see Appendix C.7) was presented to the students of each school-group. Using presentation software, each pair was presented in a slide (for an example of these slides, see Figure C.1 in Appendix C.7), which changed automatically every 10 seconds. The total duration of every declarative-knowledge measurement was 13 minutes, because 13 concepts can form a number of 78 different pairs and each pair was presented for 10 seconds. Students rated each pair and noted their rating on an answer sheet, which at the end of the measurement was returned to me. This procedure was followed in each declarative-knowledge measurement, with the exception of the first measurement. During the first occasion and before the actual measurement, an example was demonstrated with non-CS concepts so that students could familiarise themselves with the rating task.

4.3.1.2.2 Procedural-knowledge data collection (first round).

According to the quasi-experimental design (see Section 4.2.2.3), three procedural-knowledge measurements were scheduled and administered during the study with secondary-education students. The first one was scheduled after three lessons, the second after five lessons and the third after nine lessons since the beginning of the study. The second and third procedural-knowledge measurements were administered immediately after the second and third declarative-knowledge measurements, respectively. The reasons for this are explained in the description of the quasi-experimental design (Section 4.2.2.3).

In the first and second procedural-knowledge measurement, unlike the declarative-knowledge measurements, only a subset of the 13 concepts was measured. This subset was determined by the concepts that had been taught by the time of each measurement. However, the third procedural-knowledge test measured all 13 programming concepts. The items used in each test to measure procedural knowledge included tasks of predicting program output, modifying program statement(s) and completing missing statement(s) (see Section 2.7.2.3). The first, second and third procedural-knowledge test of the first round as well as the concepts based on which procedural knowledge was measured by each test are presented in Appendices C.8, C.9 and C.10, respectively.

The third measurement of declarative and procedural knowledge concluded the first round of data collection in May 2010. The time until the beginning of the following academic year was used to prepare the second round of data collection, which is discussed in the following section.

4.3.2 Second Round of Data Collection

4.3.2.1 Preparations and arrangements for the second round of data collection.

Using observations, feedback and the collected data of the first round of data collection, three important improvements were made for the second round of data collection: (a) an improved version of Koios was developed, (b) revised versions of the three procedural-knowledge tests were produced and (c) a practical test was added in the data-collection scheme. More details on these improvements follow.

Participants of the first round did not make any comments about the Koios prototype Version 0, with one exception. During the first two weeks, there were a few comments by students regarding the complexity of the graphical user interface (GUI) of Koios. Thus, it was decided to design and develop an improved version of Koios (Version 1). The improvements of Version 1 were discussed in Section 3.4. Students who used Koios Version 1 did not make any particular comments regarding GUI complexity or any other aspect of Koios.

The results of the analysis of the procedural-knowledge data were used to refine the three procedural-knowledge tests of the second round. The aim of this refinement was to make the revised versions better in discriminating students' procedural knowledge among the three different programming tools. In order to accomplish this, the results of procedural-knowledge measurements of the first round were statistically analysed. Cross-tabulation analysis was used to identify relatively easy and relatively difficult items of tests, which were modified accordingly. The results of cross-tabulation analysis are reported in Appendix C.11.

Finally, a practical test was added to the data-collection scheme. The aim of this test was to measure students' performance while they were using the programming tool assigned to their group to create new programs on their own. This is deemed to be very important, as the declarative- and procedural-knowledge tests did not involve using any programming tool. Therefore, the practical test could provide evidence regarding the effectiveness of each programming tool and demonstrate how helpful each tool could be for students during actual program creation.

As in the case of the first round, I contacted staff from several secondary and tertiary educational institutes in Greece to inform them about the study and discuss their participation. This time, staff from a high-school in the city of Patras accepted to take part in the study. Unfortunately, no replies were received from tertiary institutes. Thus, the second round of data collection was conducted only in a secondary school as well.

4.3.2.2 Second round of data collection in secondary education.

The second round of data collection took place between the 01st of November 2010 and the 02nd of May 2011 and 75 third-grade high-school students participated. Participants – 42 male students and 33 female students – lived in the city of Patras and their average age was 14 years. As in the case of the first round (see Section 4.3.1.2), students were informed about the study (see Appendix C.3) and consent forms (see Appendix C.2) were given to be signed by their parent(s)/guardian(s) before the beginning of the study.

This school had five classes in the third-grade, but after administration's decision only three classes, namely G1, G3 and G5, participated in the study. G1, G3 and G5 had 24, 26 and 24 students, respectively. For CS lessons, these classes were divided in two, according to the directions of the Greek Ministry of Education. Thus, the school-groups G1A, G1B, G3A, G3B, G5A and G5B were formed. This quasi-experiment was based on the quasi-experimental design for secondary-education (see Section 4.2.2.3). Hence, groups G1A and G1B, G3A and G3B, and G5A and G5B used Koios Version 1, Glossomatheia and MicroworldsPro, respectively, as their programming tool for the duration of the study. Each school-group had one school hour of CS lesson per week. The day and school hour of CS lessons per group were determined by the administration of the school.

Because the aim of the second round of data collection was to gather more evidence to test the three hypotheses, the second round was conducted based on the material used for the first round of data collection. More particularly, a practical test was added after the final knowledge measurement in the plan of lessons and knowledge measurements (see Appendix C.4) of the first measurement. The classroom examples (see Appendix C.5) and homework (see Appendix C.6) used in the first round were used in this round as well. This was the case for the material for declarative-knowledge measurements, but not for the material for procedural-knowledge measurements. In their place, the revised procedural-knowledge material was used. The material for the knowledge measurements is further discussed in the following sections.

4.3.2.2.1 Declarative-knowledge data collection (second round).

The three declarative-knowledge measurements took place at the beginning, after five lessons and after nine lessons, just as in the case of the first round of data collection. Likewise, declarative-knowledge was measured using ratings of concept pairs and, thus, the material (see Appendix C.7) used in first round was used in this round as well.

4.3.2.2.2 Procedural-knowledge data collection (second round).

As in the first round of data collection, this round's three procedural-knowledge tests were administered after three, five and nine weeks. However, in the second round, some or all subitems of 10 items for measuring students' procedural knowledge were modified – three, four and three items were changed in the first, second and third procedural-knowledge test, respectively. These items were identified by conducting cross-tabulation analysis on the scores of procedural-knowledge tests of the first round (see Appendix C.11) and were revised versions of the ones used in the first round. The items were revised so they could better discriminate procedural knowledge among experimental groups. The first, second and third procedural-knowledge tests with revised items are presented in Appendices C.12, C.13 and C.14 , respectively.

4.3.2.2.3 Practical test (second round).

During the first round of data collection, a number of lessons were missed and thus, there was not any time available for a practical exercise with the programming tools in classroom. However, this was feasible during the second round and therefore, a practical test was scheduled for the following week after the final measurements of procedural and declarative knowledge. During this test, students were asked to create, in class, three programs with the programming tool they had been using during the quasi-experiment. The practical test can be found in Appendix C.15. Students were asked to save the programs they had created. All programs were collected from students' computers by me.

The administration of the practical test concluded the second round of data collection. The following section discusses the data collection from teachers for producing a baseline knowledge network.

4.3.3 Data Collection for Declarative-Knowledge Baseline

In order to evaluate students' mental models of programming knowledge, a reference model was required as a baseline. For the creation of this baseline, the mental models regarding the 13 programming concepts of 10 CS teachers were measured. Like in the case of students' declarative-knowledge measurements, it involved the use of ratings of concept pairs and hence the same material (see Appendix C.7) was used to measure CS teachers' mental models. Before the rating task, teachers were given information and consent forms for their participation in the study as well as instructions for performing it.

The ratings of the 10 teachers were not collected at one occasion. Teachers performed the rating task in batches, during both rounds of data collection. On every occasion, after completing the task, they returned their ratings to me.

The data collected during both rounds through the process described in this section were submitted for analysis. The following section reports the techniques that were used for analysing these data.

4.4 Data Analysis

This section discusses the techniques used for analysing the data of declarative and procedural knowledge. First, the variables included in the analysis are reported. The techniques and statistical tests used to analyse the data of knowledge measurements and the practical test follow. Finally, the checks that were made to determine whether the data met the assumptions of parametric tests are presented.

4.4.1 Variables Used In Data Analysis

In this section, the variables used in data analysis are presented and explained. The variables regarding procedural knowledge are presented first, and are followed by the variables regarding declarative knowledge and the practical test. Finally, the variables considered as covariates are discussed.

4.4.1.1 Variables regarding procedural-knowledge data.

The scores of each procedural-knowledge test were used as the variable that represented the levels of students' procedural knowledge at the time of measurement. The tests were marked by me, but an external marker was asked to mark the tests as well, for reasons of reliability. The rationale for marking each item was the following. If the answer was completely correct, it received 100% of points, answers with minor errors received 75% of points, answers with medium errors received 50% of points, answers with serious errors received 25% of points, while if there was no answer or it was irrelevant it received zero points. A total score was calculated by summing the marks per item or subitem (see Appendices C.8, C.9, C.10, C.11, C.13 and C.14 for maximum numbers of marks per [sub]item).

4.4.1.2 Variables regarding declarative-knowledge data.

In Section 2.7.2.2, the process that the Pathfinder scaling algorithm (PSA) uses to analyse pair ratings was described. The PCKNOT software (Sitze, 1992) was used to run the PSA, with students' ratings of the 78 programming-concept pairs as input data. These ratings were used to produce data proximity files, which is a specific format for inputting ratings to PCKNOT. The output of PCKNOT for a data proximity file is a PFNET – a knowledge organisation network. Thus, a PFNET was produced by PCKNOT for each participant, which represented the participant's mental model of the 13 concepts at the time of each measurement. The produced networks were compared with a baseline PFNET and their similarity was calculated by PCKNOT. The baseline network was created by averaging the proximity data files of two teachers. Although 10 teachers completed the rating task and therefore 10 proximity data files were available, only two teachers' ratings were included in the baseline. The ratings of eight teachers were excluded due to their low values of coherence (see Section 2.7.2.2). The coherence index of a proximity data set is calculated by PCKNOT and measures the consistency of the data in the set. The PCKNOT manual (Sitze, 1992) suggests that a coherence value less than 0.20 indicates that the rating task was not or could not be taken seriously. The two proximity data files used had coherence values above or equal to 0.20. The two proximity data files could be averaged in two ways. The first was to use PCKNOT to produce the average PFNET of the two proximity data files. The second was to average the ratings of the two proximity data files and then use the averaged ratings to produce a PFNET. Both PFNETS were created, but the PFNET of the first case was selected because it presented a higher coherence value (= 0.25). This approach was used by Rose et al. (2007) to produce the baseline PFNET in their study as well. The PFNET that was used as a baseline network is shown in Figure 4.2.

The baseline PFNET presented in Figure 4.2, displays the presumed relationships between the 13 programming concepts that were produced by averaging the two PFNETs. A link that connects two programming concepts denotes that, according to the rater(s), a relationship is supposed to exist between them. The strength of this relationship is represented by the spatial distance of the two concepts; the closer the two concepts are, the stronger their relationship is.

Because students and teachers were asked to rate the strength of the relationship between pairs of concepts using a minimum and maximum value, namely one (unrelated) and ten (strongly related), respectively, these values were used as cut-offs.

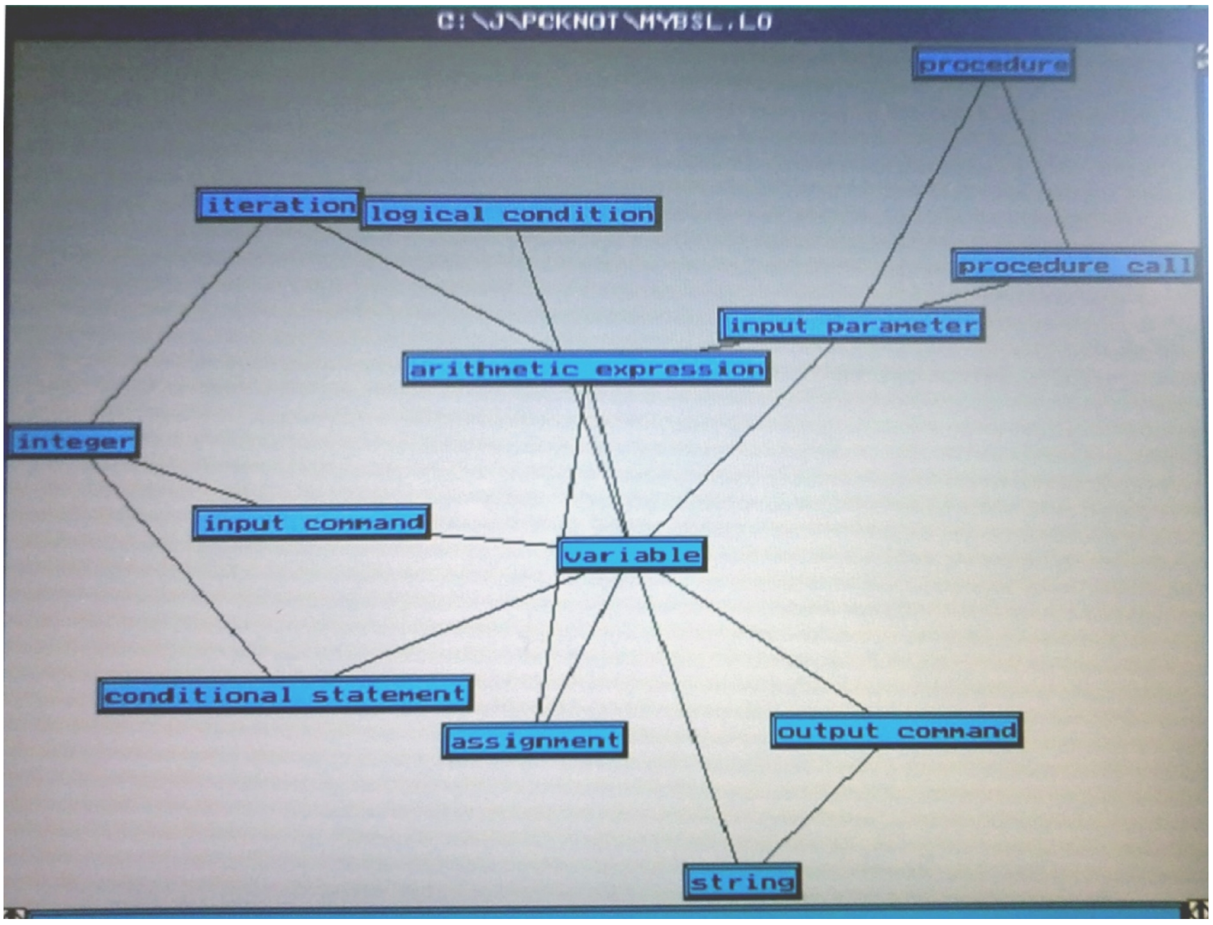


Figure 4.2. The baseline PFNET.

This means that if the strength of a relationship between two concepts was rated with a value outside the range of the cut-off values, then the distance between the two concepts was treated as infinite and no link would be created between them. Missing ratings from students' and teachers' answer sheets were substituted by imputation. Using this technique, the missing ratings in an answer sheet were replaced by the average value of the existing ratings in the sheet. After the comparison of each PFNET with the baseline PFNET, PCKNOT produced the following nine measures regarding the similarity of comparing two networks (for example, the baseline and a student's network).

1. The observed number of links that the two networks have in common.

2. The expected number of links that the networks could have in common by chance.
3. The difference between the observed and the expected by chance number of links in common of the two networks.
4. Similarity, which equals to common links / (total links – common links).
5. The expected similarity by chance.
6. The difference between the observed and the expected similarity by chance.
7. The probability of exactly this number of links in common by chance.
8. The probability of this many or more links in common by chance (TailProb).
9. Information, which is associated with the TailProb and equals to $\log_2$ (1/TailProb).

In order to choose which of the eight measures, in addition to *Similarity*, would be used as variables of declarative knowledge, the bivariate correlations between all measures were calculated. These correlations were calculated based on the results of two comparisons: (a) every teacher's PFNET with the rest of teachers' PFNETs and (b) all PFNETs produced by students' ratings over the three measurements of the first round with the baseline PFNET. The correlation matrix of the nine measures revealed that the measures that presented a high correlation ($r > 0.90$, $p < 0.01$) with *Similarity* were: *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Difference between Observed and Expected Similarity by Chance* and *Information*. Thus, it was decided to use these five measures as variables of declarative knowledge for statistical analysis. The high correlation values between *Similarity* and the selected variables could cause issues with collinearity in statistical analysis. However, each of the five declarative-knowledge variables was included in a different regression model and thus collinearity was not an issue in this data analysis.

4.4.1.3 Variables regarding practical-test data.

The scores of the practical test was the variable that represented students' performance in the practical test. I marked the programs produced during the practical test using a marking scheme, which can be found in Appendix C.16. The same scheme was used by an external marker to mark the programs of the practical test for reasons of reliability.

4.4.1.4 Variables considered for covariates.

For the analysis of the collected data, the following variables were considered as candidate covariates.

1) *Gender* – students' gender. Although gender has not been identified as a predictor of programming performance, it was decided to examine its effect on programming knowledge to establish the generality of the findings.
2) *Participant's GPA without CS* – the grade point average during the current school year was considered as a possible covariate, because it could provide a measure of participants' overall learning performance. However, GPA included the mark for CS, which might had been affected by the quasi-experiment. In order to increase the independence between this co-covariate and the DV, the mark for CS was removed and GPA was recalculated without it. The final grade in mathematics was another covariate candidate, but it

correlated highly with *Participant's GPA without CS*. Further investigation proved *Participant's GPA without CS* to have higher correlations with the outcome variables than the final math grade and therefore to be a better predictor. Thus, *Participant's GPA without CS* was included in the analysis, but the math grade was not.

3) *GPA from previous year* – the grade point average according to the performance of the student during the previous school year (the year before the year of the study). This variable could provide a gauge of students' overall learning performance during the school year before their participation in the study.

4) Homework Frequency – the number of homework assignments returned by a student could be a measure of a student's motivation in the course. In fact, this variable was used in the analysis in three parts in order to be more precise: *Homework Frequency until the first procedural-knowledge measurement, Homework Frequency until the second procedural-knowledge measurement and Homework Frequency until the third procedural-knowledge measurement.* Each of the three parts was the number of homework assignments returned by a student before the first, second and third procedural-knowledge measurement, respectively.

5) Duration of Lesson – this variable represented lessons' duration in minutes for each group per week. The school hours differed to their duration, according to their order in the timetable of the school. As a general rule, the first school hours lasted longer than the school hours towards the end of the school day. This differentiation could be considered as an unsystematic factor that could affect the performance of the experimental groups.

6) *Size* – the number of participants in each group varied due to school administrative reasons. This variation could influence groups' performance and therefore, it was included in the analysis. Furthermore, the correlation between size and the final procedural-knowledge measurement was statistically significant, r (68) = 0.34, p < 0.01.

7) *Day of Week* – the day of the week that lessons took place for each group. Different groups having lessons on different days should be tested for affecting, as an unsystematic factor, the performance of the experimental groups. Because no linear relationship between covariate and dependent variable or mediator was expected, this variable was coded as categorical.

8) *Hour of Day* – the data of this variable represented the school hour that lessons took place for each group. For the same reasons as with *Day of Week*, this covariate was treated as categorical.

9) *Gaps between Lessons* – this variable represented any gaps between consecutive lessons, except for Christmas and Easter breaks. While the lessons order was the same for all groups, for a number of reasons beyond my control, each lesson did not take place in the same week for all groups. For example, some groups had no gaps between the first procedural-knowledge measurement and the lesson before this measurement, while others had one or two gaps between the measurement and the previous lesson. The temporal distance between lessons could be an unsystematic factor and should be tested for. Therefore, the temporal gaps between lessons were included in the analysis as a categorical covariate. As with *Homework Frequency*, this variable was used in the analysis in three parts: *Gaps until the first procedural-knowledge measurement*, *Gaps until the*

second procedural-knowledge measurement and *Gaps until the third procedural-knowledge measurement*. Naturally, each of the three parts represented the gaps between lessons for each group until the first, second and third procedural-knowledge measurement, respectively.

4.4.2 Data Analysis Plan

This section explains the techniques and statistical tests used for analysing the collected data. The techniques and statistical tests used for analysing declarative-knowledge data are presented first, while for procedural-knowledge and practical-test data follow.

4.4.2.1 Analysis plan for procedural-knowledge data.

The aim for analysing procedural-knowledge data, namely students' scores in procedural-knowledge tests, was to provide evidence for testing (a) the first step of mediation analysis and (b) hypothesis H1. The first step of testing the mediation hypothesis is to establish whether the independent variable has a significant effect on the dependent variable, while according to hypothesis H1, Koios users develop a higher level of procedural knowledge than the users of Glossomatheia and MicroworldsPro. Furthermore, whether the candidate covariates had a significant effect on the dependent variable was also examined.

Therefore, the results of analysing procedural-knowledge data were used to determine whether (a) programming tool (independent variable) had a significant effect on procedural-knowledge scores (dependent variable), (b) there were any specific significant differences in the mean programming performance between quasi-experimental groups and (c) any covariate explained a significant percentage of variance in the procedural-knowledge scores. The statistical tests that can produce these results are: (a) analysis of variance (ANOVA), when there is no interest in including covariates in analysis and (b) analysis of covariance (ANCOVA) and hierarchical multiple regression (HMR) when covariates should be included (Field, 2009). However, running ANCOVA as HMR was preferred, because of its flexibility and because its results include model fit, percentage of explained variance, *b*-values and their statistical significance for the variables included in every regression model. Thus, the effects of the independent variable and covariates on the dependent variable could be better interpreted. The final decision concerned which covariates should be included in the analysis. This decision was made based on the data regarding the covariates acquired in each round of data collection, in particular only covariates that were significantly correlated with the dependent variable were selected. Thus, the selection and analysis of covariates is examined separately for each round in the following sections.

4.4.2.1.1 Analysis plan for procedural-knowledge data of the first round.

A simple regression analysis was conducted for each one of the nine covariates as the independent variable and the procedural-knowledge scores as the dependent variable, in order to determine which of the covariates were significant. Several variables were significant predictors of the procedural-knowledge scores in each measurement. However, three important observations were made. First, some covariates presented high values of variance inflation factor (VIF), which indicated a strong linear relationship with other covariates and thus, possible bias in the regression model. Second, the only common significant predictor in the three measurements was *Participant's GPA without CS*. Third, when the programming tool was included in the regression model of each measurement, all covariates that were significant before the inclusion ceased to be, with the exception of *Participant's GPA without CS*. Therefore, and in order to provide more cohesive and comparable analysis results across the three procedural-knowledge tests, *Participant's GPA without CS* was included as the only covariate in the regression model. The plan used for the data analysis of the three procedural-knowledge measurements during the first round is reported in Table 4.2.

Table 4.2
Analysis Plan for Procedural-Knowledge Data

Procedural-Knowledge Measurement	Independent Variable	Dependent Variable	Statistical Test	Covariates
First	Programming tool	Scores of the first procedural-knowledge test	HMR	*GPA without CS*
Second	Programming tool	Scores of the second procedural-knowledge test	HMR	*GPA without CS*
Third	Programming tool	Scores of the third procedural-knowledge test	HMR	*GPA without CS*

Each row of Table 4.2 refers to a procedural-knowledge measurement of the first round. The first column reports the measurement, during which the data were collected. The second and third columns report the variables used as the independent variable and the dependent variable, respectively. The fourth column reports the statistical test(s) used for analysis, while the last column reports the covariates included in the analysis of each measurement's data.

4.4.2.1.2 Analysis plan for procedural-knowledge data of the second round.

The covariates that were significant predictors of procedural-knowledge scores of the second round were determined using the method used for the first round. The results presented patterns, which were similar to the ones of the first round. More precisely, the only covariate that remained significant, in all three measurements, after the inclusion of programming tool in the regression model, was *Participant's GPA without CS*. Hence, the analysis plan for the procedural-knowledge data of the first round was used for the second round as well (see Table 4.2.)

4.4.2.2 Analysis plan for declarative-knowledge data.

The aim of analysing declarative-knowledge data was to provide evidence for testing (a) the second step of mediation analysis and (b) hypothesis H2. The second step of mediation analysis determines if the mediator has a significant effect on the dependent variable. Hypothesis H2 predicts that Koios users will acquire richer mental models than the users of Glossomatheia and MicroworldsPro. Moreover, whether the candidate covariates had a significant effect on mediator was investigated as well.

Therefore, analysis results of declarative-knowledge data were used to determine whether (a) programming tool (independent variable) had a significant effect on students' mental models (mediator), (b) there were any specific significant differences in the quality of mental models between experimental groups and (c) any covariate explained a significant percentage of variance in the declarative-knowledge data. Declarative knowledge was represented by *Links in Common, Difference between Observed and Expected Number of Links in Common, Similarity*, *Difference* *between Observed and Expected Similarity by Chance*, and *Information* (see Section 4.4.1.2). As discussed in the previous section, the statistical tests that can produce the desired results are: (a) ANOVA, if no covariates should be included in analysis and (b) ANCOVA and HMR when covariates should be included. For reasons explained in the previous section, ANCOVA was performed as HMR. Again, the final decision was to decide which covariates should be included in the analysis. This decision, just as in the case of procedural-knowledge data, was based on the collected data in each round. The following sections report this process as well as the analysis plan for each round.

4.4.2.2.1 Analysis plan for declarative-knowledge data of the first round.

First declarative-knowledge measurement

The first declarative-knowledge measurement took place in the beginning of the study, before any experimental manipulation. Thus, no covariates were considered for inclusion, with two exceptions. *Gender* and *Participant's GPA from previous year* could be significant predictors of the five declarative-knowledge variables, because students' rating task could be affected by their gender and/or their previous learning performance. Hence, simple regression analyses with *Participant's GPA from previous year* and *Gender* as independent variables and each of the five declarative-knowledge variables as dependent variables were conducted. *Gender* was not a significant predictor of the five variables, all $F < 1$. Therefore, *Gender* was not included in the regression model. However, *Participant's* GPA *from previous year* was a significant predictor and was included as a covariate in the analysis of the first declarative-knowledge measurement's data.

Second declarative-knowledge measurement

In the second declarative-knowledge measurement, the following analysis was used to determine which of the covariates, was a significant predictor of the outcome variables. A simple regression analysis was conducted with each of the outcome variables as the dependent variable and each of the covariates as the independent variable. The results of these analyses showed that *Day of Week* was a significant predictor of *Difference between Observed and Expected Number of Links in Common, F* (4,65) = 3.16, *p* < 0.025; *Difference between Observed and Expected Similarity by Chance, F* (4,65) = 3.25, *p* < 0.025; and *Information, F* (4,65) = 3.90, *p* < 0.01. *Gaps between Lessons* was determined to be a significant predictor of *Difference between Observed and Expected Similarity by Chance, F* (5, 64) = 2.81, *p* < 0.025. Based on these results the following decisions regarding the analysis of the second measurement's data were made.

Because no covariate was a significant predictor of *Links in Common* and *Similarity*, it was decided to run a one-way ANOVA for each variable. In both cases programming tool was the fixed factor. It was decided to include *Day of Week* as a covariate in the regression models of *Difference between Observed and Expected Number of Links in Common, Difference between Observed and Expected Similarity by Chance* and *Information*. It was decided to include *Gaps between Lessons* as a covariate in the regression model of *Difference between Observed and Expected Similarity by Chance*. However, the results of HMR analysis of *Difference between Observed and Expected Similarity by Chance* with *Day of Week*, *Gaps between Lessons* and programming tool as predictors presented high values of variance inflation factor (VIF), which suggested the existence of collinearity in the data. More specifically, the VIF values for *Day of Week* and *Gaps between Lessons* were 45.60

and 36.43, respectively. The correlations of these predictors with the outcome variable showed that *Gaps between Lessons* correlated higher with the outcome variable than *Day of Week*; therefore the latter variable was excluded from the regression model.

The data of *Day of Week and Gaps between Lessons* were entered in the analysis using the technique of criterion scaling (Pedhazur, 1997). This technique produces the predicted scores of a categorical variable for an outcome variable and is described in Section 4.4.2.4 .

Furthermore, the analysis results of the first declarative-knowledge measurement indicated that with respect to *Difference between Observed and Expected Number of Links in Common*, the participants of the Koios and the Glossomatheia groups came from different populations. In order to investigate whether this unsystematic variation between groups affected their performance in the second measurement of this variable, it was decided to use its scores in the first measurement as a covariate.

Third declarative-knowledge measurement

Likewise, in order to examine whether any covariates were significant predictors of the outcome variables in the third declarative-knowledge measurement, regression analysis was conducted. A simple regression analysis was conducted with each of the outcome variables as the dependent variable and each of the covariates as the independent variable. The results showed that *Day of Week* was a significant predictor of *Links in Common*, F (4, 65) = 2.72, $p < 0.05$. Based on these results, it was decided to include *Day of Week* as a covariate in the HMR analysis of this variable. For the coding of the categorical covariate *Day of Week*, the criterion scaling method was used (Pedhazur, 1997), which produced the predicted scores of

Day of Week for this outcome variable. For more details about and reasons for using criterion scaling see Section 4.4.2.4.

Moreover, data analysis of the first declarative-knowledge measurement showed that regarding *Difference between Observed and Expected Number of Links in Common*, the participants of the Koios and the Glossomatheia groups came from different populations. In order to account for any variation in groups' performance caused by this initial difference, it was decided to include the scores of *Difference between Observed and Expected Number of Links in Common* in the first measurement as a covariate in this HMR analysis of this variable.

For each of the remaining declarative-knowledge variables, a one-way ANOVA was conducted with programming tool as the fixed factor. Based on these findings, the analysis plan for the declarative-knowledge data of the first round is reported in Table 4.3. The information in this table follows the presentation scheme of Table 4.2.

Table 4.3
Analysis Plan for Declarative-Knowledge Data of the First Round

Declarative-Knowledge Measurement	Independent Variable	Dependent Variable	Statistical Test	Covariates
First		Links in common		
		Difference between observed and expected number of links in common		
	Programming tool	Similarity	HMR	*GPA from previous year*
		Difference between observed and expected similarity		
		Information		
Second		Links in common	ANOVA	
		Difference between observed and expected number of links in common	HMR	Difference Between Observed and Expected Number of Links in Common (first declarative-knowledge measurement) Predicted Value of Day of Week
	Programming tool	Similarity	ANOVA	
		Difference between observed and expected similarity	HMR	Predicted Value of Gaps Between Lessons
		Information	HMR	Predicted Value of Day of Week

Table 4.3 (continued)

Declarative-Knowledge Measurement	Independent Variable	Dependent Variable	Statistical Test	Covariates
Third	Programming tool	Links in common	HMR	Predicted Value of Day of Week
		Difference between observed and expected number of links in common	HMR	Difference Between Observed and Expected Number of Links in Common (first declarative-knowledge measurement)
		Similarity	ANOVA	
		Difference between observed and expected similarity	ANOVA	
		Information	ANOVA	

4.4.2.2.2 Analysis plan for declarative-knowledge data of the second round.

First declarative-knowledge measurement

Because declarative-knowledge was measured for the first time before any experimental manipulation, no covariates were considered for inclusion in the analysis, except for *Gender* and *Participant's GPA of previous year*. Both variables were non-significant predictors. Thus, a one-way ANOVA was conducted for each variable, with it as the dependent variable and programming tool as the fixed factor.

Second declarative-knowledge measurement

The data analysis of the first declarative-knowledge measurement revealed that the students assigned to the Koios and the MicroworldsPro groups and students assigned to the Glossomatheia group came from different populations. This was confirmed by the *t*-tests of the groups' mean scores. Naturally, these scores were achieved before any experimental manipulation. Their results showed that the means of the Koios and the Glossomatheia groups were significantly different, while the means of the Koios and MicroworldsPro groups were not. Because of the initial differences between the experimental groups, it was decided to use the variables of the first measurement as covariates in the HMR analyses of the second measurement's data. This way, it could be determined whether the unsystematic variation between groups played an important role in significantly differentiating the groups in following measurements.

Furthermore, a simple regression analysis was conducted with each outcome variable of this measurement as the dependent variable and each of the nine covariates as the predictor. The aim of this step was to test whether any of the covariates were a significant predictor. The outcome was that none of the covariates were significant predictors of this measurement's variables. Therefore, only the declarative-knowledge variables from the first measurement were included as covariates in the analysis of this measurement's data.

Third declarative-knowledge measurement

Because of the initial difference in populations of the Koios, the MicroworldsPro and the Glossomatheia groups, the variables of the first declarative-knowledge measurement were used as predictors in the analysis of this measurement's data as well. Moreover, a simple regression with each of this measurement's variables as the dependent variable and each of the covariates as the predictor was carried out. The purpose of these regression analyses was to investigate whether any of the covariates were significant predictors of this measurement's variables. The results showed that *Homework Frequency until the third procedural-knowledge measurement* was a significant predictor of *Similarity, Difference between Expected and Observed Similarity by Chance* and *Information*, F (1,68) = 4.58, $p < 0.05$; F (1,68) = 4.27, $p < 0.05$; and F (1,68) = 4.90, $p < 0.05$, respectively.

Therefore, the following decisions regarding analysis were made. In the regression model of *Links in Common,* this variable's scores in the first measurement was included as a covariate. This was the case for the regression model of *Difference between Observed and Expected Number of Links in Common*. The regression models for the three remaining variables included *Homework Frequency until the third procedural-knowledge measurement* together with the data of each variable in the first measurement as covariates. The analysis plan for the declarative-knowledge data of the second round is presented in Table 4.4.

Table 4.4
Analysis Plan for Declarative-Knowledge Data of the Second Round

Declarative-Knowledge Measurement	Independent Variable	Dependent Variable	Statistical Test	Covariates
First	Programming tool	Links in common	ANOVA	
		Difference between observed and expected number of links in common		
		Similarity		
		Difference between observed and expected similarity		
		Information		
Second	Programming tool	Links in common	HMR	Links in common (first declarative-knowledge measurement)
		Difference between observed and expected number of links in common		Difference between observed and expected number of links in common (first declarative-knowledge measurement)
		Similarity		Similarity (first declarative-knowledge measurement)
		Difference between observed and expected similarity		Difference between expected and observed similarity by chance (first declarative-knowledge measurement)
		Information		Information (first declarative-knowledge measurement)

Table 4.4 (continued)

Declarative-Knowledge Measurement	Independent Variable	Dependent Variable	Statistical Test	Covariates
Third		Links in common		Links in common (first declarative-knowledge measurement)
		Difference between observed and expected number of links in common		Difference between observed and expected number of links in common (first declarative-knowledge measurement)
		Similarity		Similarity (first declarative-knowledge measurement) Homework frequency until the third procedural-knowledge measurement
	Programming tool		HMR	
		Difference between observed and expected similarity		Difference between expected and observed similarity by chance (first declarative-knowledge measurement) Homework frequency until the third procedural-knowledge measurement
		Information		Information (first declarative-knowledge measurement) Homework Frequency until the third procedural-knowledge measurement

Finally, it is important to note that the effect of procedural knowledge on declarative knowledge was also investigated. Although, there is little evidence that procedural programming knowledge can affect declarative programming knowledge, this has been proposed as a plausible hypothesis, at least for the early stages of programming (McGill & Volet, 1997). Therefore, the five declarative-knowledge variables measured in the second and third occasion were regressed on procedural-knowledge scores of the second and third measurement, respectively. This was the case for the data of both rounds. The results revealed that the procedural-knowledge had a significant effect on declarative knowledge only in the second measurement of the first round. Therefore, no consistent evidence was found to support this hypothesis.

4.4.2.3 Analysis plan for practical-test data.

Analysis of the practical-test scores aimed to examine whether (a) programming tool significantly affected students' performance in the practical test, (b) any specific significant differences in performance existed between the experimental groups and (c) any covariates could explain a significant percentage of variance in the scores. In order to determine if any of the two types of knowledge was important in using the programming tool (practical test), the correlations between the scores of the practical test, the five variables of the final declarative-knowledge measurement and the scores of the final procedural-knowledge test were calculated. The correlations revealed that *Information* and the procedural-knowledge scores of the third measurement correlated significantly with the practical-test scores, r (66) = 0.31, $p <$ 0.01 and r (66) = 0.49, p < 0.001, respectively. However, the correlations between programming tool and *Information*, programming tool and procedural-knowledge scores of the third measurement, and *Information* and procedural-knowledge scores of the third measurement were extremely low, *r* (70) = 0.00, *p* > 0.05; *r* (70) = 0.01, *p* > 0.05 and *r* (70) = 0.06, *p* > 0.05, respectively. From the nine covariates only *Participant's GPA without CS* was used as a predictor. The first reason was that its correlation with practical-test scores was significant, *r* (66) = 0.51, *p* < 0.001. The second reason was that in analysis of procedural-knowledge scores *Participant's GPA without CS* was the only one of the nine covariates used (see Section 4.4.2.1). As already explained in previous sections, ANCOVA was conducted as HMR analysis to analyse the practical-test scores. The dependent variable was practical-test scores, while the programming tool was the independent variable. *Participant's GPA without CS*, the scores of *Information* and procedural-knowledge test measured in third occasion were included in the regression model as covariates.

4.4.2.4 Technical matters regarding data analysis.

This section provides technical information about the data analysis that was conducted. Furthermore, the method of criterion scaling that was used is explained. ANOVA, simple and hierarchical multiple regression analysis were performed using the SPSS software. Programming tool (independent variable) was coded as a categorical variable using dummy variable coding. Scores of declarative-knowledge, procedural-knowledge and practical-test were all treated as interval variables. With respect to covariates, *Gender*, *Day of Week*, *Hour of Day*, *Size* and *Gaps between Lessons* were coded as categorical variables. The covariates *Participants' GPA without CS*, *Participants' GPA from previous year*, *Homework Frequency* and *Duration of Lesson* were treated as interval variables.

As the number of categorical variables and/or their categories increases, the use of these variables in analysis becomes more difficult. A technique for managing a large number of categories is using a coding method called *criterion scaling* (Pedhazur, 1997). According to this method a categorical variable is transformed to a single vector. Each individual's score in this vector is represented by individual's predicted score, in other words the criterion mean of the group to which the individual belongs. Because the number of categories involved in data analysis was rather high, this technique was used to transform categorical covariates into single vectors. Thus, the criterion scaled covariates, instead of the original categorical covariates, were used in data analysis. This is the reason that results in Chapter Five report transformed covariates as *Predicted Value of*, followed by the name of the covariate. Criterion scaling was applied to the following covariates: *Hour of Lesson, Day of Week, Gaps between Lessons* and *Size*.

This technique, however, requires a further adjustment. Criterion scaling is used to transform a categorical variable with *k* categories to a signal vector, which in turn, is used in the place of the categorical variable in analysis. The degrees of freedom reported in analysis results include the degrees of freedom of the transformed variable, which is one. However, the correct calculation of degrees of freedom should include the degrees of freedom of the categorical variable, namely *k*-1, instead of one. Thus, the degrees of freedom, and consequently, the results of statistical tests must be adjusted to include the degrees of freedom of the original categorical variable.

4.4.3 Considerations Regarding Assumptions of Parametric Tests.

The results that are produced from the statistical tests reported in previous sections are valid only if the data used in these tests meet a set of assumptions. This set of assumptions is described as assumptions of parametric tests and assumes that the collected data (a) are normally distributed, (b) have homogenous variances, (c) are measured at least at interval level and (d) are independent across participants (Field, 2009). Because the third and fourth assumption were met due to the design of data collection, the collected data were checked for the first two assumptions.

The knowledge-measurement data of both rounds and practical-test scores were investigated by employing histograms, q-q plots, Kolomogorov-Smirnov test and Levene’s test (Field, 2009). This data exploration showed that many variables in both rounds violated either assumption or even both of them. Several transformations were tested for these variables, so their distribution would not violate the assumptions. However, the transformations resulted in no significant improvements. Based on these findings it was decided to use the variables in their original form.

Violation of normality did not raise any concerns, because ANOVA and ANCOVA run as HMR, which were used in this statistical data analysis, are robust against this type of violation. However, when variances are heterogeneous and groups’ sample sizes are not equal, the efficient use of these methods requires to set the significance level from 0.05 to 0.025, to compensate for the lack of homogeneity (Keppel, Saufley & Tokunaga, 1992). Because the sample sizes of the groups in all measurements were unequal, when homogeneity of variance was broken, the significance level was set to 0.025. A brief presentation of the cases of the collected data violating assumptions of normality and homogeneity of variance by is presented in Table 4.5.

Table 4.5
Dependent Variables that Violated Assumptions of Parametric Tests

Dependent Variable	First Round of Data Collection		Second Round of Data Collection	
	Normality	Homogeneity of Variance	Normality	Homogeneity of Variance
Links in Common (first measurement)	x	√	x	x
Difference between Observed and Expected Number of Links in Common (first measurement)	√	√	x	√
Similarity (first measurement)	√	√	x	x
Difference between Observed and Expected Similarity by Chance (first measurement)	√	√	x	x
Information (first measurement)	x	√	x	x
Procedural-knowledge Scores (first measurement)	x	√	√	√
Links in Common (second measurement)	x	x	x	√
Difference between Observed and Expected Number of Links in Common (second measurement)	√	x	√	x
Similarity (second measurement)	√	x	√	x
Difference between Observed and Expected Similarity by Chance (second measurement)	√	x	√	x
Information (second measurement)	x	x	x	x
Procedural-knowledge Scores (second measurement)	x	√	x	√
Links in Common (third measurement)	x	√	x	√
Difference between Observed and Expected Number of Links in Common (third measurement)	√	√	x	√
Similarity (third measurement)	√	√	√	√
Difference between Observed and Expected Similarity by Chance (third measurement)	√	√	x	√
Information (third measurement)	x	x	x	√
Procedural-knowledge Scores (third measurement)	x	x	√	√
Practical-test Scores	N/A	N/A	x	√

Note. N/A = not applicable, because a practical test was not administered in the first round of data collection.

4.5 Conclusions

This chapter discussed matters of designing and running an empirical study, during which data were collected for testing three research hypotheses and evaluating the Koios programming environment. More specifically, after presenting the three research hypotheses under investigation, research designs for testing these hypotheses were discussed. Based on this design, a scheme for data collection was planned. The details of, plans for and material used in the two rounds of data collection followed. Finally, the variables, statistical tests and techniques used in the analysis of the collected data were discussed. The results of this data analysis are reported in the following chapter.

Chapter Five

Results

Contents

5.1 Overview

In the first two chapters the aims and rationale of this research project were presented, the international literature was reviewed in this research's context and seven design principles were identified and discussed. The design, implementation and features of the Koios prototype programming environment were examined in Chapter Three. Chapter Four described the method, more particularly *Design*, *Participants*, *Material* and *equipment*, *Procedure*, data collection and data analysis, of this experiment. In this chapter, the results of two rounds of collection are presented. First, procedural-knowledge results and declarative-knowledge results of the first round are presented. Second, the same type of results is reported for the second round; the results of the practical test, which was only administrated in the second round, follow. This chapter concludes with a brief summary of results.

5.2 First Round of Data Collection

In this section, the results of the analysis of declarative- and procedural-knowledge data that were collected during the first round are presented. First, the results for the procedural-knowledge measurements are reported, followed by the results for the declarative-knowledge measurements. The analysis outcomes of the knowledge data were used to test hypotheses H1, H2 and H3 (mediation hypothesis).
In statistical tests, a significance level of 0.05 was used. For the analysis of variables which violated homogeneity of variance (see Table 4.5), the significance level was adjusted to 0.025 (Keppel, 1991). However, the inferential statistical tests used in these analyses (ANOVA and ANCOVA run as multiple regression) are robust

against the violation of the assumption of normality (Lorenzen & Anderson, 1993, p. 41).

5.2.1 First Step of Mediation Analysis – Procedural Knowledge

For mediation analysis, the procedure proposed by MacKinnon (2008) was used. The first step of mediation analysis establishes whether the independent variable (IV) significantly affects the dependent variable (DV). In this case, the effect of compliance with the set of principles represented by programming tool (IV) on procedural knowledge (DV) of students was tested, as per hypothesis H1 was tested. According to this hypothesis the users of Koios develop a higher level of procedural knowledge of programming concepts than the users of Glossomatheia and MicroworldsPro.

The level of procedural knowledge was measured via specific tests. The items of these tests required from participants to predict the output of programs, complete missing programming statements and modify existing programs. These items were designed to measure the procedural knowledge of concepts, which had already been taught up to the time of each measurement. The scores of these tests ranged from zero (minimum) to 20 (maximum). The three procedural-knowledge measurements were administered after three, five and nine weeks of lessons, since the beginning of the study. The final measurement was at the end of the study. For a detailed discussion on the design of this experiment and the procedure of the first round of data collection see Section 4.2.2.3 and Section 4.3.1, respectively. Moreover, a reliability analysis of marking the tests of the three procedural-knowledge measurements was carried out, as to ascertain the reliability of assessing procedural-knowledge scores. For more details about the procedure used to mark the procedural-knowledge tests as well as to assess its reliability, see Section 4.4.1.1. For reasons discussed in Section 4.4.2.1.1 , it was decided that *Participant's grade point average* (*GPA*) *without CS* would be the only covariate in the hierarchical multiple regression (HMR) analysis of all procedural-knowledge scores. This predictor was entered in the first regression model, followed by the two dummy variables representing the independent variable programming tool in the second model.

5.2.1.1 Descriptive statistics for procedural-knowledge measurements.

The mean scores of the three procedural-knowledge tests for all three groups (Koios, Glossomatheia and MicroworldsPro) are presented in Figure 5.1.

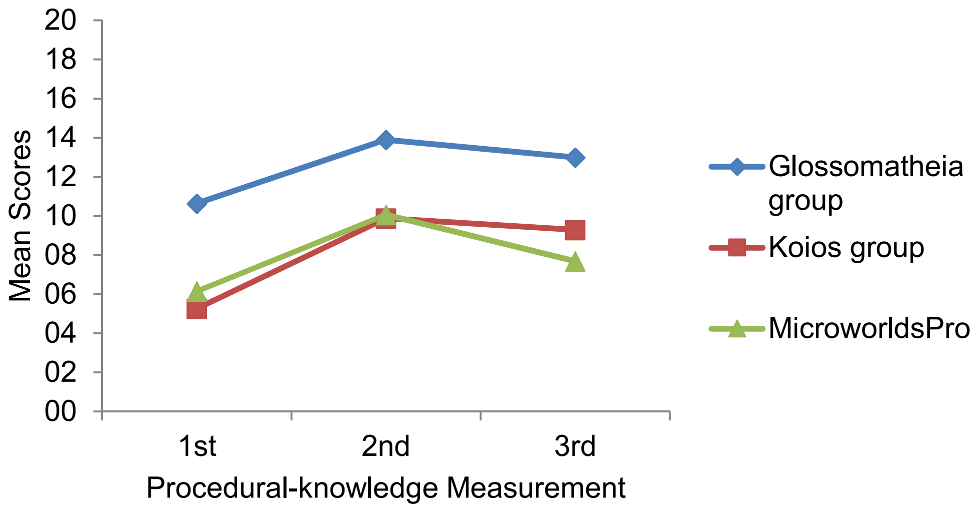


Figure 5.1. Mean scores for each group over the three procedural-knowledge measurements (first round).

The minimum and maximum values per test were 0 and 20, respectively. Therefore, the minimum and maximum values of y-axis are 0 and 20, respectively. Examination of Figure 5.1 seems to reveal that the mean performance of each group was better in the second measurement than its performance in the first one. Nonetheless, the mean score of each group in the third measurement seems slightly lower than its mean score in the second measurement, but higher than its mean score in the first one. In the following sections the results of inferential statistical analysis are presented separately for each measurement, based on the analysis plan presented in Section 4.4.2.1.1

5.2.1.2 First procedural-knowledge measurement.

The first measurement of procedural knowledge was taken in the beginning of the fourth week of lessons and it took approximately 10-15 minutes to complete. Six programming concepts were measured: *integer, arithmetic expression, string, output-statement, input-statement* and *variable.* The reliability of the eight items of this test was high, Cronbach's *alpha* = 0.72, while the marking of the first procedural-knowledge test was reliable, *r* (15) = 0.76. Table 5.1 presents descriptive statistics for this variable.

Table 5.1
Descriptive Statistics for the Scores of the First Procedural-Knowledge Measurement (First Round)

Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Koios	21	5.27	5.41	1.18	
Glossomatheia	22	10.64	3.58	0.76	-1.17
MicroworldsPro	24	6.15	4.42	0.90	-0.18
Total	67	7.35	5.03	0.61	

Note. Because not all pupils attended the test session, sample size was reduced in Koios group.

As can be seen in Table 5.1, the mean scores of all three groups were overall between relatively low and medium values, ranging approximately from five to 10. In particular, the *mean* score for the Koios, the Glossomatheia and the MicroworldsPro groups was 5.27, 10.64 and 6.15, respectively. Likewise, the *standard deviation* (*SD*) for each group was 5.41, 3.58 and 4.42, respectively. According to Cohen's (1988) recommendations[1], the standardised difference of the group scores between Koios and Glossomatheia represented a large effect size, while between the Koios and the MicroworldsPro groups indicated a rather small effect size. The outcome of HMR analysis, with *Participant's GPA without CS* in the first model and the dummy variables representing programming tool in the second, is reported in Table 5.2.

Table 5.2
Results of Regression Analysis for the Scores of the First Procedural-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-8.16	3.43		*-2.38
	Participant's GPA without CS	0.99	0.22	0.49	***4.58
Model 2					
	Constant	-8.64	3.23		*-2.68
	Participant's GPA without CS	0.89	0.20	0.44	***4.47
	Glossomatheia vs Koios	4.88	1.21	0.46	***4.02
	MicroworldsPro vs Koios	1.39	1.19	0.13	1.17

Note. R^2 = 0.24 for Model 1; ΔR^2 = 0.16 for Model 2 ($p < 0.001$). * $p < 0.05$, *** $p < 0.001$

Analysis of the data showed that 24% of variance in the scores of the first procedural-knowledge test was explained by *GPA without CS*, $R^2 = 0.24$, F (1, 65) = 20.96, $p < 0.001$. This percentage was significant and thus, *Participant's GPA without CS* was a significant predictor of procedural knowledge. The additional percentage of variance explained, when programming tool was added in the model, was 16% and was statistically significant, $\Delta R^2 = 0.16$, F_{change} (2, 63) = 8.61, $p < 0.001$. Therefore, programming tool was a significant predictor. *GPA without CS* was a significant predictor of the outcome variable in the second model as well, *t*

[1] According to Cohen's (1988) conventions for *d*, values of 0.20, 0.50 and 0.80 represent a small, moderate and large effect size, respectively.

(63) = 4.47, *p* = < 0.001. With respect to the group means, students of the Glossomatheia group tended to have significantly higher scores than students of the Koios group, *t* (63) = 4.02, *p* < 0.001, but no significant difference was observed in the mean scores between the MicroworldsPro and the Koios groups, *t* (63) = 1.17, *p* > 0.05.

According to these results, hypothesis H1 should be rejected. However, the first step of mediation was established, because the independent variable programming tool significantly affected the dependent variable student's procedural knowledge.

5.2.1.3 Second procedural-knowledge measurement.

The second procedural-knowledge test took place in the beginning of the sixth week of lessons and just before the second declarative-knowledge measurement. The nine items of the second test were found to be highly reliable, Cronbach's *alpha* = 0.78. This test measured the concepts of *integer, arithmetic expression, string, output-statement, input-statement, variable* and *assignment-statement* and took approximately 15-20 minutes to administer. The reliability of marking this measurement's procedural-knowledge tests was high, *r* (15) = 0.82. Table 5.3 presents descriptive statistics for the scores of the second procedural-knowledge measurement.

Table 5.3
Descriptive Statistics for the Scores of the Second Procedural-Knowledge Measurement (First Round)

Group	*N*	*Mean*	*SD*	*SE*	*Cohen's d*
Koios	23	9.89	5.56	1.16	
Glossomatheia	21	13.90	4.20	0.92	-0.81
MicroworldsPro	26	10.07	4.82	0.94	-0.03
Total	70	11.16	5.16	0.62	

Note. Because not all pupils attended the test session, sample size was reduced in Glossomatheia group.

As Table 5.3 presents, the mean scores of the three groups were approximately from 10 to 14. Given that the minimum and maximum values of the test were 0 and 20, respectively, these means are middle-range values. Specifically, the *mean* and *SD* for the Koios, the Glossomatheia and the MicroworldsPro groups were 9.89 and 5.56, 13.90 and 4.20 and, 10.07 and 4.82, respectively. Based on Cohen's (1988) recommendations, a large effect size was detected for the difference in the mean scores of the Koios and the Glossomatheia groups. A small effect size was detected for the mean difference of the Koios and the MicroworldsPro groups. The results of HMR analysis of this test's scores, with *Participant's GPA without CS* and programming tool as predictors, are shown in Table 5.4.

Table 5.4
Results of Regression Analysis for the Scores of the Second Procedural-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-7.46	3.26		*-2.29
	Participant's GPA without CS	1.19	0.21	0.57	***5.78
Model 2					
	Constant	-7.38	3.28		*-2.25
	Participant's GPA without CS	1.10	0.20	0.53	***5.45
	Glossomatheia vs Koios	3.22	1.25	0.29	*2.58
	MicroworldsPro vs Koios	0.76	1.18	0.07	0.65

Note. $R^2 = 0.33$ for Model 1; $\Delta R^2 = 0.07$ for Model 2 ($p < 0.05$). *$p < 0.05$, *** $p < 0.001$

The results of the first model indicated that *Participant's GPA without CS* accounted for a statistically significant proportion of variance in the scores of the second test. Particularly, 33% of variance in the scores was explained by *GPA without CS*, $R^2 = 0.33$, $F\,(1, 68) = 33.43$, $p < 0.001$ and this predictor was significant. The addition of programming tool to the regression model explained an extra 7% of variance in the scores; this additional variance was statistically significant as well, $\Delta R^2 = 0.07$, $F_{change}\,(2, 66) = 3.58$, $p < 0.05$. Thus, programming tool was a significant predictor of the second test's scores, as was *Participant's GPA without CS*, *t* (66) = 5.45, *p* < 0.001. The comparison of the mean scores showed that the Glossomatheia group students achieved significantly higher scores than students of the Koios group, *t* (66) = 2.58, *p* < 0.05, but the mean scores of the MicroworldsPro and the Koios groups were not significantly different, |*t*| < 1.

Based on the results, hypothesis H1 had to be rejected for this measurement. However, the independent variable programming tool had a significant effect on the dependent variable level of procedural knowledge; thus the first step of mediation was established for this measurement.

5.2.1.4 Third procedural-knowledge measurement.

The final measurement of procedural knowledge occurred after nine weeks of lessons – at the end of the study – and just before the final measurement of declarative knowledge. This test measured all 13 programming concepts, namely *integer, arithmetic expression, string, output-statement, input-statement, variable, assignment-statement, iteration-statement, procedure, procedure call, input parameter, logical condition* and *conditional statement*. The test took approximately 20-25 minutes to administer. The 10 items of the third test, presented high reliability, Cronbach's *alpha* = 0.81. The reliability of marking the procedural-knowledge tests of the third measurement was high, *r* (15) = 0.83. The data of this measurement violated both the assumptions of normality and homogeneity of variance and therefore the significance level for this variable's analysis was reduced to 0.025. Table 5.5 shows descriptive statistics for the scores of this measurement.

Table 5.5
Descriptive Statistics for the Scores of the Third Procedural-Knowledge Measurement (First round)

Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Koios	22	9.30	5.52	1.18	
Glossomatheia	22	12.99	3.03	0.64	-0.83
MicroworldsPro	26	7.68	5.77	1.13	0.28
Total	70	9.86	5.40	0.65	

The mean scores of the three groups were ranging approximately from 8 to 13. These values were relatively low to medium, because the minimum and maximum values of the scores were 0 and 20 respectively. In particular, the *mean* for the Koios, the Glossomatheia and the MicroworldsPro groups was 9.30, 12.99 and 7.68, respectively, while the *SD* was 5.52, 3.03 and 5.77, respectively. Values of Cohen's *d* (1988) indicated a large effect size for the difference between the mean scores of the Koios and the Glossomatheia groups and a small effect size for the mean-score difference between the Koios and the MicroworldsPro groups. Table 5.6 presents the HMR results for the scores of the final procedural test, using *Participant's GPA without CS* and programming tool as predictors.

Table 5.6
Results of Regression Analysis for the Scores of the Third Procedural-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-10.33	3.35		**-3.09
	Participant's GPA without CS	1.29	0.21	0.60	***6.11
Model 2					
	Constant	-9.26	3.29		**-2.82
	Participant's GPA without CS	1.17	0.20	0.54	***5.85
	Glossomatheia vs Koios	3.25	1.23	0.28	*#2.64
	MicroworldsPro vs Koios	-0.74	1.19	-0.07	-0.63

Note. $R^2 = 0.35$ for Model 1; $\Delta R^2 = 0.10$ for Model 2 ($p < 0.01$). *#$p < 0.025$, ** $p < 0.01$, *** $p < 0.001$

The results of the first regression model showed that 35% of the variance in the scores was explained by *GPA without CS*, $R^2 = 0.35$, $F\ (1, 68) = 37.31$, $p < 0.001$. The percentage of the explained variance was significant and therefore, *GPA without CS* was a significant predictor of the final scores. Results of the second model showed that an additional 10% of variance in the scores was explained by the inclusion of programming tool in the regression model, $\Delta R^2 = 0.10$, $F_{change}\ (2, 66) = 6.08$, $p < 0.01$.

This additional (10%) percentage was statistically significant, so programming tool was a statistically significant predictor. *Participant's GPA without CS* was a significant predictor of procedural knowledge in the second model as well, $t\ (66) = 5.85$, $p < 0.001$. The mean-score difference between the Glossomatheia and the Koios groups revealed that students of the Glossomatheia group tended to perform better in this test than the students of the Koios group, $t\ (66) = 2.64$, $p < 0.025$. Nonetheless, no significant difference was reported between the mean performance of the MicroworldsPro and the Koios groups, $|t| < 1$.

These results supported the rejection of hypothesis H1 for this final measurement. The significant effect of programming tool (IV) on the scores of the final procedural-knowledge measurement (DV) yielded that the first step of mediation hypothesis was true for the final measurement.

5.2.2 Second Step of Mediation Analysis – Declarative Knowledge

The second step of mediation analysis is to test whether an experimental effect of the independent variable (IV) on the mediator (M) exists. In this case, this step investigated whether programming tool (IV) used by students affected their declarative knowledge, represented by their mental models (M). Furthermore, in this step hypothesis H2 was tested using inferential statistics. The aim was to test if the students, who use Koios as their programming tool, construct richer mental models than the users of Glossomatheia and MicroworldsPro.

The five variables selected as the main representatives of declarative knowledge were: *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information*. They were measured in the beginning of the study, after five and after nine weeks of lessons since the study commenced. The final measurement was taken at the end of the study. These variables were measures of similarity between a desired baseline knowledge-network and the networks students produced during declarative-knowledge measurements. For a detailed discussion of the five variables and the selection procedure, and the design of this experiment see Section 4.4.1.2 and Section 4.2.2.3, respectively.

5.2.2.1 Descriptive statistics for declarative-knowledge measurements.

The mean scores of *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference between Observed and Expected Similarity by Chance* and *Information* for each of the three groups over the three measurements are presented in Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5 and Figure 5.6, respectively. In each of the following figures, the y-axis ranges from the minimum to the maximum value of the variable that is analysed. In order to provide more information for examining these figures the maximum and minimum value of each declarative-knowledge variable were calculated.

The maximum values of the five variables were calculated using the baseline knowledge-network. More precisely, the value for each variable when the baseline network was compared with itself was considered as maximum value for the specific variable. The maximum values for *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference between Observed and Expected Similarity by Chance and Information* were 19, 14.37, 1.00, 0.86 and 59.22, respectively.

The variable *Links in Common* measured the common links between two networks. If two networks have no links in common, then the value of *Links in Common* is zero. The values of this variable cannot be negative, because negative values do not make sense for measuring common links. Therefore, the minimum value for *Links in Common* was zero. *Similarity* measured the closeness of the two networks as a proportion of *Links in Common*, while *Information* a probability associated with *Links in Common.* For more details on the three measures, see Chapter 4, Section 4.4.1.2. Because the calculation of each of the latter two variables was based on a multiplication with *Links in Common*, the minimum value for each of the two variables was zero as well.

The estimation of minimum values for *Difference between Observed and Expected Number of Links in Common* and *Difference between Observed and Expected Similarity by Chance* required the expected values of *Links in Common* and *Similarity*, respectively. For any two networks, *N1* and *N2*, these expected values are computed based on (a) all possible networks that have as many links and nodes as *N1* has and (b) all possible networks that have as many links and nodes as *N2* has. Then, each network of (a) is combined with all the networks of (b). For every combination, the value of *Links in Common* is calculated and the expected value of

Links in Common is the average of the calculated values. The same procedure is used to estimate the expected value of *Similarity*. Given that (a) all networks of this study had 13 nodes, (b) the baseline network had 19 links, (c) the number of links of a student's network could range from 1 to 78 and (d) the calculation of all possible networks with 13 nodes and *n* number of links is based on combinations, the number of all possible different networks is

$$\frac{78!}{n!\ 78-n\ !}\ . \tag{1}$$

Therefore, it is obvious that cumbersome calculations were involved in the estimation of the minimum values of *Difference between Observed and Expected Number of Links in Common* and *Difference between Observed and Expected Similarity by Chance.* In addition, the exact value of these minima would add little information to the illustrative purposes of the figures. Therefore, the minimum values of *Difference between Observed and Expected Number of Links in Common* and *Difference between Observed and Expected Similarity by Chance* observed in this round of data collection were used as minimum values in the figures; they were -3.63 and -0.11, respectively.

The examination of these figures revealed that in the first declarative-knowledge measurement, the mean scores of all variables for the three groups were very low. In the second measurement, the mean score of each group seemed higher than its score in the first measurement for all variables. From the second to the final measurement, an apparent slight decline was noticed in the performance's level of each group for all variables. Overall, the mean scores of all variables over the three measurements were very low, possibly an indication of a floor effect.

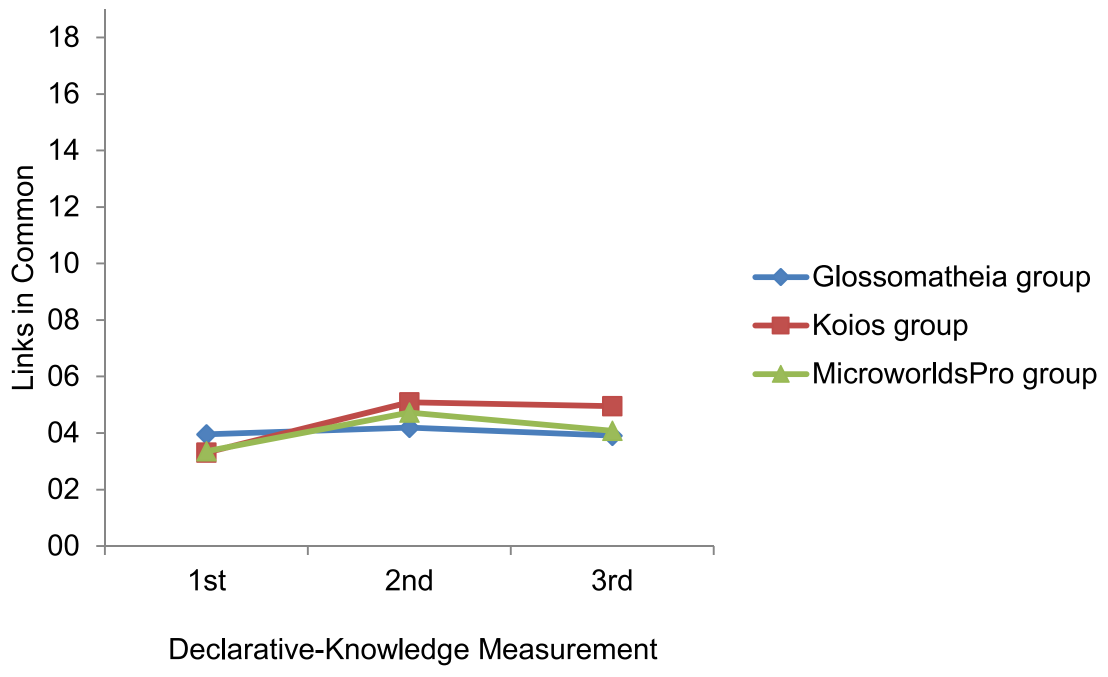


Figure 5.2. Mean scores of Links in Common for each group over the three measurements (first round).

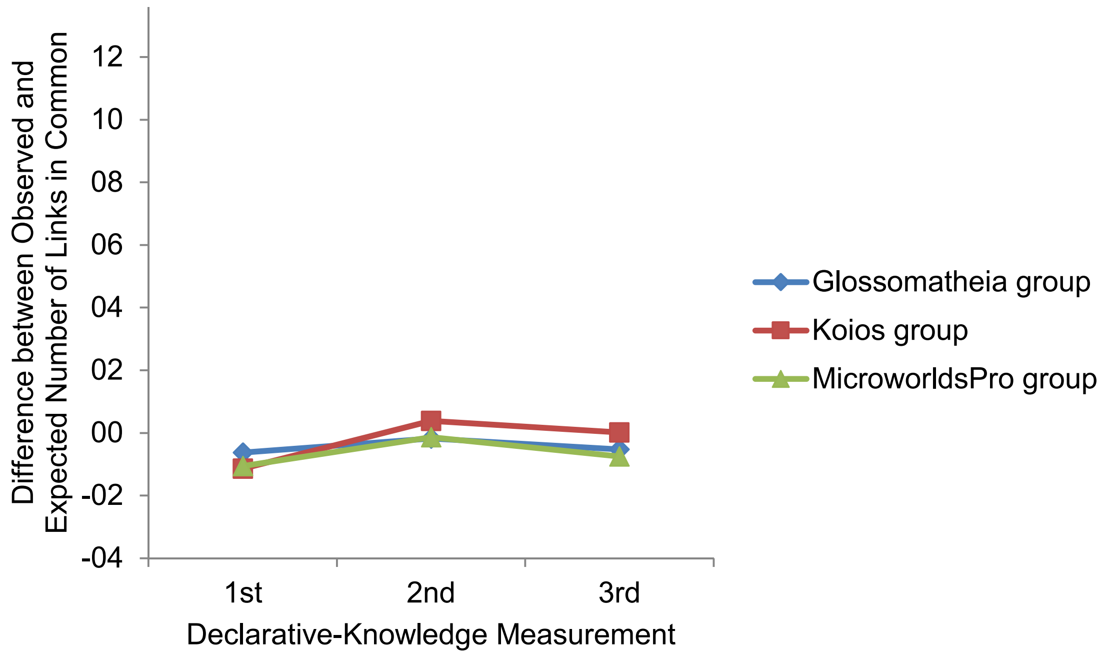


Figure 5.3. Mean scores of Difference between Observed and Expected Number of Links in Common for each group over the three measurements (first round).

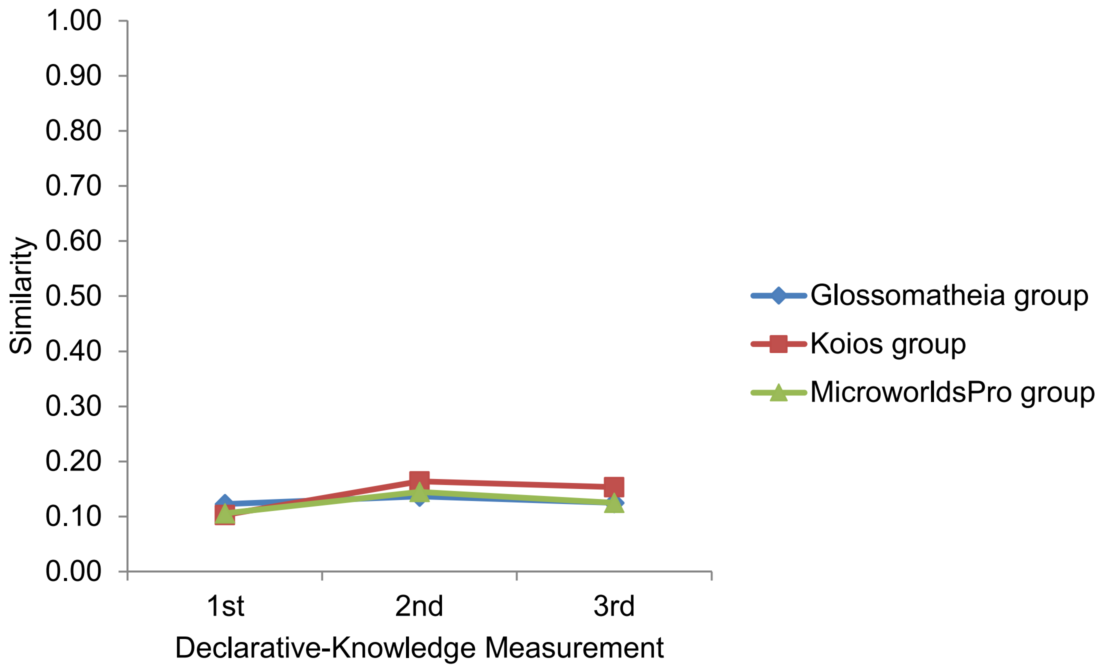


Figure 5.4. Mean scores of Similarity for each group over the three measurements (first round).

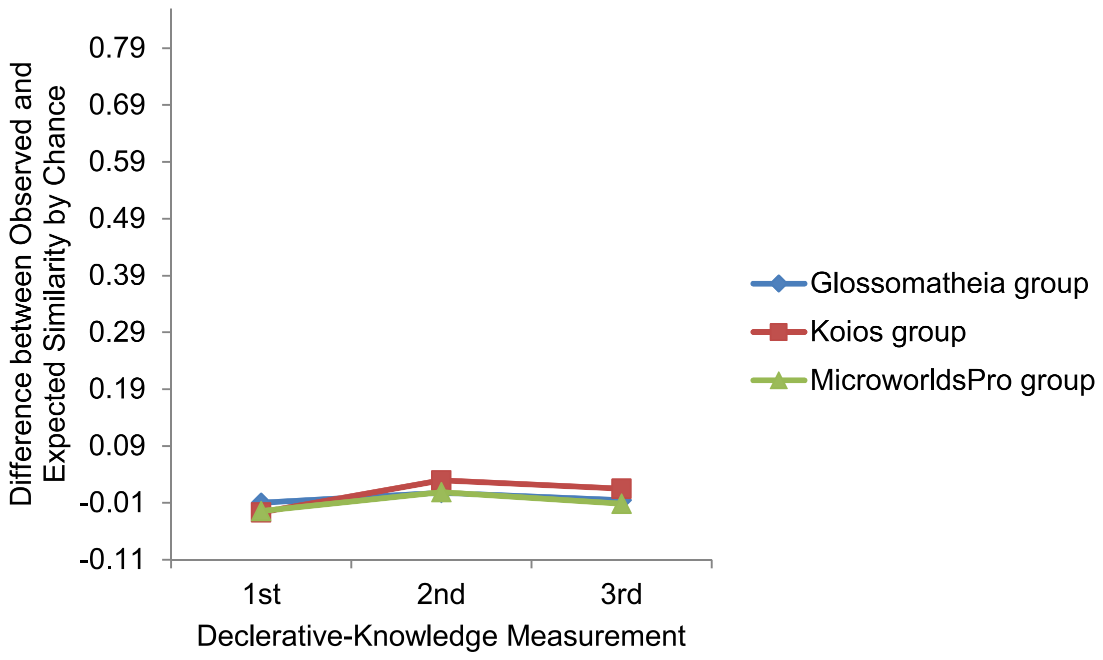


Figure 5.5. Mean scores of Difference between Observed and Expected Similarity by Chance for each group over the three measurements (first round).

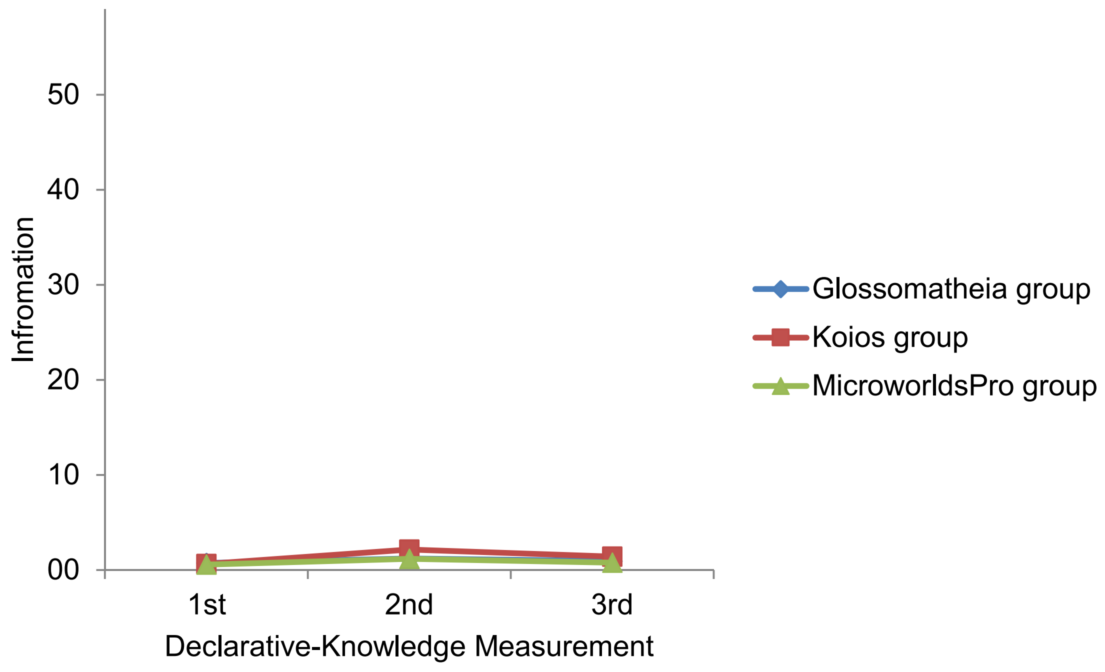


Figure 5.6. Mean scores of Information for each group over the three measurements (first round).

These five measures were used in inferential statistical analysis as variables of declarative knowledge. Each of the following three sections describes the results for each declarative-knowledge measurement. These results of the second step of mediation analysis were used to test hypothesis H2.

5.2.2.2 First declarative-knowledge measurement.

The first declarative-knowledge measurement took place at the beginning of this round of data collection. Descriptive statistics for the variables, included in the analysis, are presented in Table 5.7. The mean values of all five variables, shown in Table 5.7, were very low. More particularly, the *means* for the Koios, the Glossomatheia and the MicroworldsPro groups for *Links In Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference* *between Observed and Expected* Similarity and *Information* were 3.30, 3.95 and 3.96; -0.74, -0.22 and -0.65; 0.10, 0.12 and 0.11; -0.03, -0.01 and -0.02; and 0.63, 0.73 and 0.59, respectively. The *SD* for the same groups and variables were 1.74, 1.43 and 1.35; 1.46, 1.01 and 1.23; 0.06, 0.04 and 0.04; 0.05, 0.04 and 0.04; and 0.82, 0.67 and 0.65, respectively. Cohen's *d* (1988) indicated small to medium effect sizes for the differences between the means of the Koios and the Glossomatheia groups for the five variables.

Table 5.7
Descriptive Statistics for the Outcome Variables of the First Declarative-Knowledge Measurement (First Round)

Variable	Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Links in common	Koios	23	3.30	1.74	0.36	
	Glossomatheia	21	3.95	1.43	0.31	-0.40
	MicroworldsPro	25	3.36	1.35	0.27	-0.04
	Total	69	3.52	1.52	0.18	
Difference between observed and expected number of links in common	Koios	23	-0.74	1.46	0.30	
	Glossomatheia	21	-0.22	1.01	0.22	-0.41
	MicroworldsPro	25	-0.65	1.23	0.25	-0.06
	Total	69	-0.55	1.25	0.15	
Similarity	Koios	23	0.10	0.06	0.01	
	Glossomatheia	21	0.12	0.04	0.01	-0.39
	MicroworldsPro	25	0.11	0.04	0.01	-0.05
	Total	69	0.11	0.05	0.01	
Difference between observed and expected similarity	Koios	23	-0.03	0.05	0.01	
	Glossomatheia	21	-0.01	0.04	0.01	-0.38
	MicroworldsPro	25	-0.02	0.04	0.01	-0.05
	Total	69	-0.02	0.04	0.01	
Information	Koios	23	0.63	0.82	0.17	
	Glossomatheia	21	0.73	0.67	0.15	-0.14
	MicroworldsPro	25	0.59	0.65	0.13	0.06
	Total	69	0.65	0.71	0.09	

Note. Because not all pupils attended the test session, sample size was reduced in Glossomatheia group.

The effect sizes represented by the differences between the means of the Koios and the MicroworldsPro groups for the five measures were very small. As explained in Section 4.4.2.2.1, because this measurement occurred before any experimental manipulation, the covariates that were considered for inclusion in the analysis were *Gender* and *Participant's GPA of previous year*. However, *Gender* was not a significant predictor of any of the five variables and therefore, *Gender* was not included in the regression models. The results of HMR analysis for the five declarative-knowledge variables with *Participant's GPA of previous year* in the first model and programming tool in the second model follow. In Table 5.8 are reported the results for *Links In Common*.

Table 5.8
Results of Regression Analysis for Links in Common of the First Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	6.26	1.25		***5.00
	Participant's GPA of previous year	-0.16	0.08	-.027	*-2.17
Model 2					
	Constant	6.10	1.31		***4.64
	Participant's GPA of previous year	-0.18	0.08	-0.29	*-2.31
	Glossomatheia vs Koios	0.82	0.45	0.27	1.84
	MicroworldsPro vs Koios	0.23	0.45	0.08	0.51

Note. $R^2 = 0.07$ for Model 1; $\Delta R^2 = 0.06$ for Model 2 ($p > 0.05$). * $p < 0.05$, *** $p < 0.001$

The first model of regression showed that 7% of variance in this outcome variable was explained by *Participant's GPA of previous year,* $R^2 = 0.07$, $F(1, 59) = 4.70$, $p < 0.05$. This percentage was significant and *Participant's GPA of previous year* was a significant predictor. The results of the second model revealed that an additional percentage of 6% was explained by programming tool, which was not significant $\Delta R^2 = 0.06$, $F_{change}(2, 57) = 1.82$, $p > 0.05$. Therefore, programming tool was not a significant predictor in the second model, but *Participant's GPA of previous year* was, *t* (57) = -2.31, $p < 0.05$. However, no significant differences were observed between the Glossomatheia and the Koios groups, *t* (57) = 1.84, $p > 0.05$ and between the MicroworldsPro and the Koios groups, $|t| < 1$. The results of HMR analysis *for Difference between Observed and Expected Number of Links in Common* are reported in Table 5.9.

Table 5.9
Results of Regression Analysis for Difference between Observed and Expected Number of Links in Common of the First Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	2.43	0.97		*2.50
	Participant's GPA of previous year	-0.18	0.06	-0.37	**-3.07
Model 2					
	Constant	2.23	1.01		*2.20
	Participant's GPA of previous year	-0.19	0.06	-.039	**-3.19
	Glossomatheia vs Koios	0.71	0.35	0.29	*2.07
	MicroworldsPro vs Koios	0.26	0.35	0.11	0.74

Note. $R^2 = 0.14$ for Model 1; $\Delta R^2 = 0.06$ for Model 2 ($p > 0.05$). * $p < 0.05$, ** $p < 0.01$

The percentage of variance in this outcome variable explained by *Participant's GPA of previous year* was 14%, $R^2 = 0.14$, *F* (1, 59) = 9.42, $p < 0.01$. This proportion was significant and thus, this predictor was significant as well. In the second model, the addition of programming tool explained an extra 6% of variance, $\Delta R^2 = 0.06$, F_{change} (2, 57) = 2.20, $p > 0.05$. The extra variance (6%) explained by programming tool was not significant and therefore, programming tool was a non-significant predictor. However, in the second model, *Participant's GPA of previous year* was a significant predictor, *t* (57) = -3.19, $p < 0.01$. A significant difference was observed between the mean scores of the Glossomatheia and the Koios groups, *t* (57) = 2.07, $p < 0.05$. The Glossomatheia users seem to have performed better in this variable than the Koios users. Nonetheless, no significant differences were observed between the

MicroworldsPro and the Koios users, |*t*| < 1. The results of HMR analysis for *Similarity* are reported in Table 5.10.

Table 5.10
Results of Regression Analysis for Similarity of the First Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.21	0.04		***5.53
	Participant's GPA of previous year	-0.01	0.00	-0.32	*-2.63
Model 2					
	Constant	0.20	0.04		***5.10
	Participant's GPA of previous year	-0.01	0.00	-0.34	**-2.74
	Glossomatheia vs Koios	0.03	0.01	0.27	1.93
	MicroworldsPro vs Koios	0.01	0.01	0.10	0.66

Note. R^2 = 0.11 for Model 1; ΔR^2 = 0.06 for Model 2 ($p > 0.05$). * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

The results of the first model showed that 11% of variance in *Similarity* was explained by *Participant's GPA of previous year,* $R^2 = 0.11$, $F(1, 59) = 6.90$, $p < 0.05$. This predictor was significant, as was the variance it explained. The inclusion of the programming tool in the second model explained an additional 6% of variance in *Similarity*, $\Delta R^2 = 0.06$, $F_{change}(2, 57) = 1.93$, $p > 0.05$. Neither this percentage (6%) nor programming tool as predictor was significant. However, *Participant's GPA of previous year* was a significant predictor in this model, $t(57) = -2.74$, $p < 0.01$. No significant differences in mean scores were observed between the Glossomatheia and the Koios groups, $t(57) = 1.93$, $p > 0.05$, or between the MicroworldsPro and the Koios groups, |*t*| < 1. Table 5.11 reports the results of HMR analysis for *Difference between Observed and Expected Similarity by Chance*.

Table 5.11
Results of Regression Analysis for Difference between Observed and Expected Similarity by Chance of the First Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.08	0.03		*2.41
	Participant's GPA of previous year	-0.01	0.00	-0.36	**-3.01
Model 2					
	Constant	0.08	0.04		*2.11
	Participant's GPA of previous year	-0.01	0.00	-0.38	**-3.12
	Glossomatheia vs Koios	0.02	0.01	0.28	1.98
	MicroworldsPro vs Koios	0.01	0.01	0.10	0.71

Note. R^2 = 0.13 for Model 1; ΔR^2 = 0.06 for Model 2 ($p > 0.05$). * $p < 0.05$, ** $p < 0.01$

The first model showed that 13% of the variance in this variable was explained by *Participant's GPA of previous year,* $R^2 = 0.13$, F (1, 59) = 9.05, $p < 0.01$. This explained proportion as well as this model's predictor was significant. The results of the second model revealed that programming tool explained an extra 6% of variance in this variable, which was not significant, $\Delta R^2 = 0.06$, F_{change} (2, 57) = 2.02, $p > 0.05$. Therefore, programming tool was not a significant predictor in this model. However, *Participant's GPA of previous year* was a significant predictor in the second model as well, t (57) = -3.12, $p < 0.01$. The differences of the mean score between the Glossomatheia and the Koios groups and between the MicroworldsPro and the Koios groups were non-significant, t (57) = 1.98, $p < 0.05$ and $|t| < 1$, respectively. The results of HMR analysis for *Information* are presented in Table 5.12.

The proportion of variance in the final declarative-knowledge variable of the first measurement that was explained by *Participant's GPA of previous year* was 11%, R^2 = 0.11, F (1, 59) = 7.38, $p < 0.01$. The explained proportion was significant and thus, *Participant's GPA of previous year* was a significant predictor of *Information*.

Table 5.12
Results of Regression Analysis for Information of the First Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	2.16	0.57		**3.81
	Participant's GPA of previous year	-0.09	0.03	-0.33	*-2.71
Model 2					
	Constant	2.13	0.61		*3.52
	Participant's GPA of previous year	-0.10	0.04	-0.35	*-2.75
	Glossomatheia vs Koios	0.23	0.21	0.16	1.13
	MicroworldsPro vs Koios	0.05	0.21	0.03	0.23

Note. R^2 = .11 for Model 1; ΔR^2 = .02 for Model 2 (p > .05). *p < .01, ** p < .001

The inclusion of programming tool in the second model explained an additional percentage of 2% in *Information*'s variance, $\Delta R^2 = 0.02$, $F_{change} < 1$. This additional proportion (2%) was non-significant and thus, programming tool was a non-significant predictor. In the second model, *Participant's GPA of previous year* was a significant predictor, t (57) = -2.75, p < 0.01. The difference in the mean scores for *Information* between the Glossomatheia and the Koios groups was not significant, t (57) = 1.13, p > 0.05. This was the case for the difference in the mean scores for *Information* between the MicroworldsPro and the Koios groups, $|t| < 1$.

The analysis of the data revealed that programming tool had no effect on students' mental models. This result is not surprising, because no experimental manipulation had been applied to the three groups when this measurement was taken. Therefore, it was meaningless to test the validity of hypothesis H2 and the second step of mediation.

Furthermore, no significant differences between the mean scores of the Koios group and each of the other two groups were observed for the five variables, with one exception. The mean score of the Glossomatheia group was significantly higher than the mean score of the Koios group for *Difference between Observed and*

Expected Number of Links in Common. It is important to note that, because this variable was measured before any experimental manipulation, this difference indicated that for this variable, the participants of the Koios and the Glossomatheia groups came from different populations. However, the results suggest that for the specific variable, the participants of the Koios and the MicroworldsPro groups came from the same population. Furthermore, the results also confirmed that for the remaining four variables, the participants of the three groups came from the same population.

5.2.2.3 Second declarative-knowledge measurement.

Declarative knowledge was measured for a second time, in the beginning of the sixth week of lessons and just before the second procedural-knowledge measurement. Table 5.13 presents a set of descriptive statistics for the five variables measured on this occasion.

The mean values in Table 5.13 were rather low for all variables. The *means* for the Koios, the Glossomatheia and the MicroworldsPro groups for *Links In Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference between Observed and Expected* Similarity and *Information* were 5.09, 4.19 and 4.73; 0.79, 0.22 and 0.26; 0.16, 0.16 and 0.14; 0.03, 0.01 and 0.01; and 2.15, 1.23 and 1.20, respectively. The *SD* for the same groups and variables were 2.94, 1.81 and 1.07; 2.44, 1.56 and 1.50; 0.10, 0.06 and 0.06; 0.09, 0.06 and 0.05; and 2.46, 1.31 and 1.03, respectively.

Table 5.13
Descriptive Statistics for the Outcome Variables of the Second Declarative-Knowledge Measurement (First Round)

Variable	Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Links in common	Koios	23	5.09	2.94	0.61	
	Glossomatheia	21	4.19	1.81	0.39	0.36
	MicroworldsPro	26	4.73	2.07	0.41	0.14
	Total	70	4.69	2.32	0.28	
Difference between observed and expected number of links in common	Koios	23	0.79	2.44	0.51	
	Glossomatheia	21	0.22	1.56	0.34	0.27
	MicroworldsPro	26	0.26	1.50	0.29	0.26
	Total	70	0.42	1.87	0.22	
Similarity	Koios	23	0.16	0.10	0.02	
	Glossomatheia	21	0.14	0.06	0.01	0.34
	MicroworldsPro	26	0.14	0.06	0.01	0.25
	Total	70	0.15	0.07	0.01	
Difference between observed and expected similarity	Koios	23	0.03	0.09	0.02	
	Glossomatheia	21	0.01	0.06	0.01	0.29
	MicroworldsPro	26	0.01	0.05	0.01	0.29
	Total	70	0.02	0.07	0.01	
Information	Koios	23	2.15	2.46	0.51	
	Glossomatheia	21	1.23	1.31	0.29	0.46
	MicroworldsPro	26	1.20	1.03	0.20	0.52
	Total	70	1.52	1.74	0.21	

Note. Because not all pupils attended the test session, sample size was reduced in Glossomatheia group.

According to Cohen's (1988) suggestions, small to medium effect sizes were detected between the mean differences of the Koios and the Glossomatheia groups and of the Koios and the MicroworldsPro groups for the five measures. None of the variables of this measurement met the assumption of homogeneity of variance and therefore the significance level for their analyses was reduced to 0.025. The analysis of this measurement's variables followed the plan presented in Section 4.4.2.2.1.

In the following analyses, in order to code the categorical covariates *Day of Week* and *Gaps between Lessons* the method of criterion scaling (Pedhazur, 1997) was used. For more details on the reason this technique was used, see Section 4.4.2.4 The results of ANOVA for *Links in Common* and *Similarity* were not significant, $F < 1$ and $F < 1$, respectively. Furthermore, planned contrasts revealed no significant differences between the Koios and each of the other two groups for these two variables. The results of planned contrasts between the Koios and the Glossomatheia groups for *Links in Common* and *Similarity* were t (67) = 1.28, $p > 0.05$ and t (67) = 1.26, $p > 0.05$, respectively. For the same variables the outcome of contrasts between the Koios and the MicroworldsPro groups were $|t| < 1$ and $|t| < 1$, respectively. Table 5.14 reports the results of HMR analysis with *Difference between Observed and Expected Number of Links in Common* as the dependent variable.

In the first model the predicted value of *Day of Week* and *Difference between Observed and Expected Number of Links in Common* of the first measurement were entered, while the dummy variables of programming tool were included in the second. Data analysis showed that scores of *Difference between Observed and Expected Number of Links in Common* of the first measurement and *Day of Week* explained 17% of variance in *Difference between Observed and Expected Number of Links in Common* of the second measurement, $R^2 = 0.17$, F (5,61) = 2.53, $p > 0.025$. *Difference between Observed and Expected Number of Links in Common* of the first measurement was not a significant predictor, $|t| < 1$.

Table 5.14
Results of Regression Analysis for Difference between Observed and Expected Number of Links in Common of the Second Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-0.01	0.25		-0.05
	Difference Between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	0.03	0.19	0.02	0.18
	Predicted Value of Day of Week	1.05	0.31	0.42	**3.38
Model 2					
	Constant	-0.40	0.50		-0.80
	Difference Between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	0.07	0.20	0.05	0.37
	Predicted Value of Day Of Week	1.25	0.38	0.50	**3.30
	Glossomatheia vs Koios	0.37	0.60	0.09	0.62
	MicroworldsPro vs Koios	0.58	0.62	0.15	0.94

Note. $R^2 = 0.17$ for Model 1; $\Delta R^2 = 0.01$ for Model 2 ($p > 0.05$). ** $p < 0.01$

However, *Day of Week* was significant predictor of this variable in the first model, *t* (61) = 3.38, $p < 0.01$. The results of the second model revealed that programming tool alone explained a non-significant proportion of 1% of variance in this variable, $\Delta R^2 = 0.01$, $F_{change} < 1$. Therefore, the programming tool students were using was a non-significant predictor of this variable in the second model, as was *Difference between Observed and Expected Number of Links in Common* of the first measurement, $|t| < 1$. However, *Day of Week* was a significant predictor in the second model as well, *t* (59) = 3.30, $p < 0.01$. No significant differences in the mean score of this variable between the Glossomatheia and the Koios users were found, $|t| < 1$. This was the case for the Koios and the MicroworldsPro users as well, $|t| < 1$. The results of HMR analysis for *Difference between Observed and Expected Similarity by Chance* are shown in Table 5.15.

Table 5.15
Results of Regression Analysis for Difference between Observed and Expected Similarity by Chance of the Second Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.00	0.01		0.00
	Predicted Value of Gaps Between Lessons	1.00	0.26	0.42	***3.86
Model 2					
	Constant	0.00	0.02		-0.02
	Predicted Value of Gaps Between Lessons	1.01	0.28	0.43	**3.59
	Glossomatheia vs Koios	0.00	0.02	-0.03	-0.20
	MicroworldsPro vs Koios	0.00	0.02	0.03	0.18

Note. $R^2 = 0.18$ for Model 1; $\Delta R^2 = 0.00$ for Model 2 ($p > 0.05$). ** $p < 0.01$, *** $p < 0.001$.

The first model of regression analysis showed that 18% of variance in *Difference between Observed and Expected Similarity by Chance* was explained by *Gaps between Lessons*, $R^2 = 0.18$, F (5,64) = 2.79 , $p < 0.025$. It also showed that *Gaps between Lessons* was a significant predictor of this variable, t (64) = 3.86, $p < 0.001$. The inclusion of programming tool did not explain any additional percent of variance in this variable, $\Delta R^2 = .00$, $F_{change} < 1$. Consequently, programming tool was a non-significant predictor of this variable. Nonetheless, *Gaps between Lessons* was a significant predictor in the second model, t (62) = 3.59, $p < 0.01$. Students of the Glossomatheia and the Koios groups had mean scores with no significant differences for this variable, $|t| < 1$. Similarly, the mean scores of the MicroworldsPro and the Koios users did not differ significantly, $|t| < 1$.

HMR analysis with *Information* as the dependent variable was conducted with *Day of Week* in the first model and the categorical variables of programming tool in the second model. The outcome can be seen in Table 5.16.

Table 5.16
Results of Regression Analysis for Information of the Second Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*T*
Model 1					
	Constant	0.00	0.42		0.00
	Predicted Value of Day of Week	1.00	0.25	0.44	***4.04
Model 2					
	Constant	-0.19	0.79		-0.24
	Predicted Value of Day of Week	1.06	0.33	0.47	**3.26
	Glossomatheia vs Koios	0.07	0.57	0.02	0.11
	MicroworldsPro vs Koios	0.20	0.58	0.06	0.35

Note. $R^2 = 0.19$ for Model 1; $\Delta R^2 = 0.00$ for Model 2 ($p > 0.05$). ** $p < 0.01$, *** $p < 0.001$

Analysis of the data revealed that *Day of Week* explained 19% of the variance in *Information*, $R^2 = 0.19$, $F\ (4, 65) = 3.90$, $p < 0.001$ and that it was a significant predictor, $t\ (65) = 4.04$, $p < 0.001$. The results of the second model showed that no additional variance in *Information* was explained by programming tool, $\Delta R^2 = 0.00$, $F_{change} < 1$. Thus, in the second model, programming tool was not a significant predictor, but *Day of Week* was, $t\ (63) = 3.26$, $p < 0.01$. No significant differences were observed between the means of the Glossomatheia and the Koios groups, $|t| < 1$ and of the MicroworldsPro and the Koios groups, $|t| < 1$.

After five weeks of lessons, the main effect of programming tool on all five variables was not significant. Planned contrasts and *t*-test values revealed that the programming tool used by students, was non-significant in differentiating the quality of mental models, constructed in the context of introductory programming. Nonetheless, *Day of Week* was a significant predictor of *Difference between Observed and Expected Number of Links in Common* and *Information*. Moreover, *Gaps between Lessons* was a significant predictor of *Difference between Observed and Expected Similarity by Chance.*

Evidence was found for neither hypothesis H2 nor the second step of mediation analysis. In order to further investigate whether a mediation effect could be detected the correlations between each of the five variables (M) and the scores of the second procedural-knowledge test (DV) were calculated. However, no significant correlations (all < 0.30) were found and therefore, the mediator seemed to have no significant effect on the dependent variable. Thus, programming tool (IV) could not affect procedural knowledge (DV) via mediation of the mental models.

5.2.2.4 Third declarative-knowledge measurement.

The final declarative-knowledge measurement took place at the end of the first round of data collection. Table 5.17 presents descriptive statistics for the measured variables.

The mean scores of the three groups were relatively low for the five variables. More particularly, the *means* for the Koios, the Glossomatheia and the MicroworldsPro groups for *Links In Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference between Observed and Expected* Similarity and *Information* were 4.95, 3.91 and 4.08; 0.41, -0.12 and -0.35; 0.45, 0.12 and 0.12; 0.01, -0.01 and -0.01; and 1.41, 0.94 and 0.78, respectively. The *SD* for the same groups and variables were 1.94, 1.66 and 1.38; 1.69, 1.42 and 1.39; 0.06, 0.06 and 0.04; 0.06, 0.05 and 0.05; and 1.57, 1.04 and 0.72, respectively. Compared with Cohen’s (1988) standards, the difference of the group means between Koios and Glossomatheia indicated a medium to large effect size for *Links in Common* and small to medium effect sizes for the remaining variables. The effect sizes represented by the mean differences of the Koios and the MicroworldsPro groups were medium to large for the five variables.

Table 5.17
Descriptive Statistics for the Outcome Variables of the Third Declarative-Knowledge Measurement (First Round)

Variable	Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Links in common	Koios	22	4.95	1.94	0.41	
	Glossomatheia	22	3.91	1.66	0.35	0.58
	MicroworldsPro	26	4.08	1.38	0.27	0.53
	Total	70	4.30	1.70	0.20	
Difference between observed and expected number of links in common	Koios	22	0.41	1.69	0.36	
	Glossomatheia	22	-0.12	1.42	0.30	0.34
	MicroworldsPro	26	-0.35	1.39	0.27	0.50
	Total	70	-0.04	1.51	0.18	
Similarity	Koios	22	0.15	0.06	0.01	
	Glossomatheia	22	0.12	0.06	0.01	0.48
	MicroworldsPro	26	0.12	0.04	0.01	0.54
	Total	70	0.13	0.05	0.01	
Difference between observed and expected similarity	Koios	22	0.01	0.06	0.01	
	Glossomatheia	22	-0.01	0.05	0.01	0.35
	MicroworldsPro	26	-0.01	0.05	0.01	0.49
	Total	70	0.00	0.05	0.01	
Information	Koios	22	1.41	1.57	0.34	
	Glossomatheia	22	0.94	1.04	0.22	0.35
	MicroworldsPro	26	0.78	0.72	0.14	0.53
	Total	70	1.03	1.16	0.14	

All variables of this measurement, except for Information, met the assumptions of normality and homogeneity of variance. Therefore, the significance level was reduced to 0.025 only for the analysis of Information. The plan for analysing the data of this measurement was discussed in Section 4.4.2.2.1. For the coding of the categorical covariate Day of Week the criterion scaling method was used (Pedhazur, 1997). For more details about criterion scaling and the reasons for using it, see Section 4.4.2.4. Table 5.18 presents the results of HMR analysis for *Links in*

Common, with *Day of Week* in the first model and the dummy variables of programming tool in the second.

Table 5.18
Results of Regression Analysis for Links in Common of the Third Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.00	1.29		0.00
	Predicted Value of Day of Week	1.00	0.30	0.38	**3.38
Model 2					
	Constant	0.30	1.93		0.15
	Predicted Value of Day of Week	0.95	0.39	0.36	*2.45
	Glossomatheia vs Koios	-0.27	0.58	-0.07	-0.46
	MicroworldsPro vs Koios	-0.01	0.58	0.00	-0.02

Note. $R^2 = 0.14$ for Model 1; $\Delta R^2 = 0.01$ for Model 2 ($p > 0.05$). * $p < 0.05$, ** $p < 0.01$

The results of the first model of regression analysis revealed that *Day of Week* accounted for 14% of variance in the scores of *Links in Common,* $R^2 = 0.14$, F (4,65) = 2.72 , $p < 0.05$ and that it was a significant predictor t (65) = 3.38, $p < 0.01$. The results of the second model showed that programming tool was a non-significant predictor of this variable's scores, as it explained 1% of variance, $\Delta R^2 = 0.01$, $F_{change} < 1$. In the second model, *Day of Week* was a significant predictor, t (63) = 2.45, $p < 0.05$. Nonetheless, the difference between the means of the Glossomatheia and the Koios groups for this variable was not significant, $|t| < 1$, as was the difference between the means of the MicroworldsPro and the Koios groups, $|t| < 1$. The results of the HMR analysis of *Difference between Observed and Expected Number of Links in Common*, with the scores of the specific variable of the first measurement and programming tool as predictors are reported in Table 5.19.

Table 5.19

Results of Regression Analysis for Difference between Observed and Expected Number of Links in Common of the Third Declarative-Knowledge Measurement (First Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-0.14	0.20		-0.72
	Difference Between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	-0.19	0.15	-0.16	-1.32
Model 2					
	Constant	0.28	0.34		0.83
	Difference Between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	-0.19	0.15	-.016	-1.31
	Glossomatheia vs Koios	-0.38	0.46	-0.12	-0.83
	MicroworldsPro vs Koios	-0.84	0.44	-0.27	-1.91

Note. $R^2 = 0.03$ for Model 1; $\Delta R^2 = 0.05$ for Model 2 ($p > 0.05$).

The results of the first model showed that *Difference between Observed and Expected Number of Links in Common* of the first measurement explained 3% of the variance in the scores of the same variable, when it was measured in the third measurement, $R^2 = 0.03$, $F\,(1,65) = 1.74$, $p > 0.05$. The addition of programming tool in the second model explained an extra 5% in the variance of the dependent variable, which was not significant, $\Delta R^2 = 0.05$, $F_{change}\,(2, 63) = 1.84$, $p > 0.05$. Therefore, programming tool was not a significant predictor of the dependent variable. *Difference between Observed and Expected Number of Links in Common* of the first measurement was not a significant predictor either, $t\,(63) = -1.31$, $p > 0.05$. Moreover, no significant differences were observed in the mean scores between the Glossomatheia and the Koios groups, $|t| < 1$. This was the case for the difference in the mean scores between the MicroworldsPro and the Koios groups, $t\,(63) = -1.91$, $p > 0.05$.

The results of ANOVA for *Similarity*, *Difference between Observed and Expected Similarity by Chance* and *Information* were *F* (2, 67) = 2.10, *p* > 0.05; *F* (2, 67) = 1.54, *p* > 0.05; and *F* (2, 67) = 1.89, *p* > 0.05, respectively. Hence, in the final measurement, the main effect of programming tool on students' mental models was not significant.

The planned comparisons of the Koios and the Glossomatheia groups for *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* were *t* (67) = 1.75, *p* > 0.05; *t* (67) = 1.25, *p* > 0.05; and *t* (67) = 1.36, *p* > 0.05, respectively. The planned contrasts for these three variables between the Koios and the MicroworldsPro groups were *t* (67) = 1.82, *p* > 0.05; *t* (67) = 1.70, *p* > 0.05; and *t* (67) = 1.90, *p* > 0.05, respectively. The results of the planned comparison revealed that the differences in the mean scores for *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* were not significant neither between the Koios and the Glossomatheia groups nor between the Koios and the MicroworldsPro groups.

However, some of these results varied when the one-tailed probability was considered. With respect to *Difference between Observed and Expected Number of Links in Common*, Koios users seem to have performed better than MicroworldsPro users in this variable, when the one-tailed probability of the *t*-test was considered, *t* (63) = -1.91, *p* < 0.05 one-tailed only, *d* = 0.50 (see Table 5.19). The group difference between Koios and Glossomatheia for this variable was not-significant, even when one-tailed probability was considered, |*t*| < 1. The one-tailed comparisons of the Koios and the Glossomatheia groups for *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* were *t* (67) = 1.75, *p* < 0.05 one-tailed only, *d* = 0.48; *t* (67) = 1.25, *p* > 0.05; and *t* (67) = 1.36, *p* >

0.05, respectively. The planned contrasts for the three variables between the Koios and the MicroworldsPro groups were *t* (67) = 1.82, *p* < 0.05 one-tailed only, *d* = 0.54; *t* (67) = 1.70, *p* < 0.05 one-tailed only, *d* = 0.49; and *t* (67) = 1.90, *p* > 0.025 one-tailed only, respectively. According to hypothesis H2, students who learn to program using Koios construct richer mental models than those constructed by students who use tools that do not highly comply with the set of seven principles. The group means in Table 5.17 were consistent with this one-tailed hypothesis. The results of the planned contrasts in the expected direction were significant for *Difference between Observed and Expected Number of Links in Common, Similarity,* and *Difference between Observed and Expected Similarity by Chance*. In particular, students in the Koios group constructed significantly richer mental models than students in the MicroworldsPro group, when the mental models were measured based on *Difference between Observed and Expected Number of Links in Common*, and *Difference between Expected and Observed Similarity by Chance*. On these measures, the mental models of the Koios and the Glossomatheia users showed no significant difference. When the mental models were measured by *Similarity*, the Koios users seemed to have created significantly richer mental models than those formed by both the Glossomatheia and the MicroworldsPro users. However, the one-tailed results cannot be accepted and used to test the research hypotheses because hypotheses H1 and H2 were not proposed as directional and thus, one-tailed testing was not planned for analysing the data of this declarative-knowledge measurement.

Hence – based on the (two-tailed) results – hypothesis H2 was not confirmed for this measurement. Furthermore, the effect of programming tool (IV) on pupils' mental models (M) was not significant overall. To further investigate the possibility of an existing mediation effect, the correlations between each of the five variables (M) and the scores of the third declarative-knowledge measurement (DV) were computed. All correlations were non- significant and below 0.20, which means that the dependent variable appeared not to be affected by the mediator. Hence, no mediation effect could be established between programming tool (IV) and procedural knowledge (DV) through mental models.

5.2.3 Check of Multiple Regression Assumptions

In order to test whether the regression models used in the analysis of the procedural- and declarative-knowledge data were biased, further checks had to be made. These checks examined the existence of any outliers or influential cases, the validity of homoscedasticity and assumptions of linear relationship between outcome and predictors, and independence of errors. For each multiple regression analysis conducted, residual statistics, histogram, p-p plot of the standardised residual, plot of the standardised residual against standardised predicted values and the Durbin-Watson statistic test were employed to that end (Field, 2009).

Residual statistics revealed four (Case 29, Case 33, Case 37 and Case 64), three (Case 29, Case 57 and Case 63) and three (Case 59,Case 70 and Case 72) extreme cases with standardised residual outside the limits of minus two and two for the first, second and third procedural-knowledge measurement, respectively. In the first declarative-knowledge measurement, one (Case 17), three (Case 16, Case 69 and Case 72), one (Case 17), four (Case 16, Case 38, Case 69 and Case 72) and four (Case 16, Case 38, Case 69 and Case 72) extreme cases were found in the regression analysis of *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and*

Expected Similarity by Chance and *Information*, respectively. In the second declarative-knowledge measurement, two (Case 32 and Case 42), three (Case 24, Case 32 and Case 42) and three (Case 24, Case 29 and Case 42) extreme cases were observed in the regression analysis of *Difference between Observed and Expected Number of Links in Common*, *Difference between Expected and Observed Similarity by Chance* and *Information*, respectively. Finally, two (Case 27 and Case 34) and three (Case 27, Case 29 and Case 33) extreme cases were found when *Links In Common* and *Difference between Observed and Expected Number of Links in Common* of the third declarative-knowledge measurement were analysed, respectively. To determine if these cases were potential outliers or influential, Cook's distance, three and two times the average leverage, Mahalanobis distance, covariance ratio and DFBeta statistics were calculated for each case (Field, 2009). The investigation of these statistics revealed that these cases were neither influential nor outlying.

Histograms and the p-p plots of the standardised residual against standardised predicted values were plotted for each of the seven HMR analyses. Their examination indicated that assumptions of homoscedasticity and linear relationship between outcome and predictors were tenable for all the cases.

The values of Durbin-Watson statistic calculated during analysis of the first, second and third procedural-knowledge data set were 2.10, 1.61 and 2.10, respectively. For the models used in the analysis of *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* of the first declarative-knowledge measurement, the values of the Durbin–Watson statistic were 1.32, 1.41, 1.30, 1.41 and 1.51, respectively. For the regression models used in the analysis of the declarative-knowledge data of the second measurement, the Durbin-Watson statistic was 2.53, 2.60 and 2.59 for *Difference between Observed and Expected Number of Links in Common*, *Difference between Expected and Observed Similarity by Chance*, and *Information*, respectively. Finally, for *Links In Common* and *Difference between Observed and Expected Number of Links in Common* of the third measurement, the values of the Durbin–Watson statistic were 2.37 and 2.39, respectively. These values suggested that the residuals of each data set were independent. Additionally, VIF and tolerance values provided no evidence of multicollinearity in any of the 13 analyses.

As explained in Section 4.4.2, HMR analysis was used to perform ANCOVA and in order to obtain valid results from ANCOVA, the assumption of homogeneity of regression slopes must be met. Therefore, for every HMR analysis performed in this round of data collection that included at least one covariate, an additional ANCOVA was conducted that included the main effects of the independent variable and the covariate(s) as well as the interaction term between them (Field, 2009). The results revealed that none of the interaction terms explained a significant percentage of variance. Therefore, the assumption of homogeneity of regression slopes was not violated in any of the performed HMR analyses.

5.2.4 Summary of the Results of the First Round of Data Collection

The first measurement of students' declarative knowledge took place in the beginning of the first round of data collection. The results of this measurement showed that there was no main effect of programming tool on all five variables. Planned contrasts revealed no significant differences between the means of the three groups for all five variables, with one exception. Students of the

Glossomatheia group performed better than the students of the Koios group for *Difference between Observed and Expected Number of Links in Common.* Furthermore, *Participant's GPA of previous year* was a significant predictor for all five variables.

The first procedural-knowledge measurement was conducted after three weeks since the beginning of the study. In this measurement, the main effect of programming tool on the programming skills developed for the concepts of *integer, arithmetic expression, string, output command, input command* and *variable* was significant. *Participant's GPA without CS* was determined to be significant predictor of students' programming skills for this set of concepts as well. The Glossomatheia users achieved significantly higher procedural-knowledge scores for these concepts than the Koios users, but the users of MicroworldsPro did not.

After five weeks of lessons from the beginning of the study, the second procedural- and declarative-knowledge measurements took place. The main effect of programming tool on procedural knowledge of the following concepts: *integer, arithmetic expression, string, output-statement, input-statement, variable and assignment-statement* was significant. Again, *Participant's GPA without CS* was a significant predictor of the programming skills acquired until this measurement. Moreover, the Glossomatheia group's mean score of procedural knowledge was significantly higher than the mean score of the Koios group. No significant difference between the mean scores of the MicroworldsPro and the Koios groups was noticed. Thus, the first step of mediation analysis was successful for this measurement. The results of the second declarative measurement showed that the main effect of programming tool on participants' mental models was non-significant. Likewise, the differences between the mean score of the Koios group and each of the other two groups were not significant for the five variables. The second step of mediation analysis was not verified for this measurement, because students' declarative knowledge did not seem to have a significant effect on their procedural knowledge. Hence, the mediator seemed to have no significant effect on the dependent variable and therefore, no mediation effect could be established. In this measurement, *Day of Week* was a significant predictor of *Difference between Observed and Expected Number of Links in Common* and *Information*. The data of *Difference between Observed and Expected Similarity by Chance* were significantly predicted by *Gaps between Lessons.*

The procedural and declarative knowledge of participant was measured for a last time at the end of the study, after nine weeks of lessons. The main effect of programming tool on the programming skills – regarding the 13 concepts – that were developed throughout this study was significant. *Participant's GPA without CS* was found to be a significant predictor. Students using Glossomatheia seem to have performed better in this final procedural-knowledge test than students using Koios. However, no significant differences in the levels of procedural knowledge between the users of Koios and MicroworldsPro were observed. Hence, the first step of mediation analysis was successful for this measurement. The analysis of the declarative-knowledge measurement showed that the main effect of programming tool on participants' mental models was not significant. Consequently, the effect of the independent variable on the mediator could not be confirmed and thus, no mediation analysis could take place. Moreover, the planned comparisons between the Koios and each of the other two groups on the third declarative-knowledge test revealed that the Koios users did not seem to construct significantly richer mental models than the users of Glossomatheia and MicroworldsPro. Finally, in this third measurement, the covariate *Day of Week* was a significant predictor of the variable *Links in Common*.

5.3 Second Round of Data Collection

This section reports the results for the second round of data collection, using the same structure as that used to report the results of the first round (see Section 5.2). The results for procedural knowledge are presented first. The outcomes for declarative knowledge follow. These results were used to test hypotheses H1 and H2 as well as the mediation hypothesis H3. Finally, the results of the practical test are presented.

A significance level of 0.05 was used in the following analyses. In the case of data violating the assumption of homogeneity of variances (see Table 4.5), analysis was carried out with an adjusted significance level of 0.025 (Keppel et al., 1992). No adjustment of the significance level was considered when the normality assumption was broken, because the inferential statistical methods used in the following analyses (ANOVA and ANCOVA run as multiple regression) are robust against violation of normality (Lorenzen & Anderson, 1993).

5.3.1 First Step of Mediation Analysis – Procedural knowledge

As with the case of the first round of data collection, the results for procedural knowledge were used to test the validity of hypothesis H1 and the first step of mediation analysis. Hypothesis H1 claims that users of Koios develop a higher level of procedural knowledge than users of Glossomatheia and MicroworldsPro. The first step of mediation analysis checks if programming tool (IV) significantly affects participants’ procedural knowledge (DV).

The tests used for the three procedural-knowledge measurements in the second round of data collection were revised versions of the ones used in the first round. The revisions were based on students’ performance in each item of the tests and changes were made so that they would become more ‘sensitive’ in discriminating the level of procedural knowledge between quasi-experimental groups. For more details on these revisions see Section 4.3.2.1. The minimum and maximum mark of each test was 0 and 20, respectively. The three procedural-knowledge measurements took place after three, five and nine weeks of lessons – just like the measurements of the first round. For a detailed discussion of the data-collection procedure of the second round see Section 4.3.2. In order to assess how reliable the marking of the procedural-knowledge tests was, a reliability analysis (Field, 2009) of the marking was conducted. More details on how these tests were marked and how the reliability of marking was measured are reported in Section 4.4.1.1

Finally, in the analysis of this round’s procedural-knowledge data, *Participant’s GPA without CS* was the only covariate that was used as a predictor. The reasons for this choice were discussed in Section 4.4.2.1.2. Thus, *Participant’s GPA without CS* was entered in the first HMR model and the categorical variables of programming tool were included in the second model. The plan used for analysing procedural-knowledge data was presented in Section 4.4.2.1.2.

5.3.1.1 Descriptive statistics for procedural-knowledge measurements.

In Figure 5.7 the mean scores of procedural knowledge for the three groups over the three measurements are presented. Values of y-axis range from 0 (minimum) to 20 (maximum).

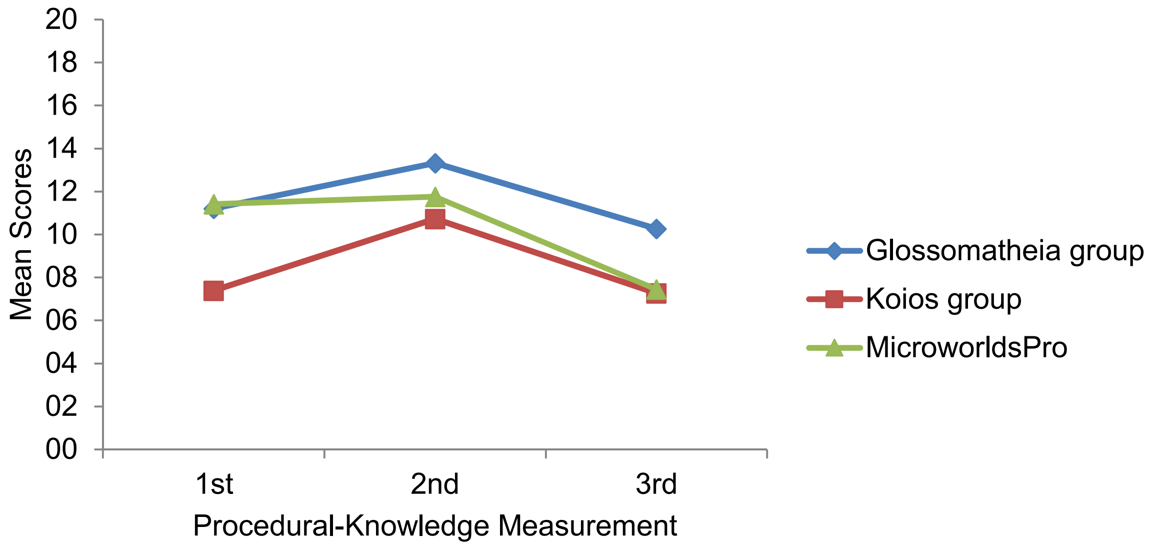


Figure 5.7. Mean scores for each group over the three procedural-knowledge measurements (second round).

As can be seen in Figure 5.7, the scores of the three measurements for the Koios, the Glossomatheia and the MicroworldsPro groups were from relatively low to medium. The mean score of each group in the second measurement seems higher than its score in the first measurement. Nevertheless, the score of each group in the third measurement seems lower than its score in the second. The results of inferential statistical analysis for each one of the three measurements follow.

5.3.1.2 First procedural-knowledge measurement.

The first procedural-knowledge measurement took place after three weeks of lessons and took approximately 15 minutes to complete. This test measured the procedural knowledge of the following programming concepts: *integer, arithmetic expression, string, output command, input command* and *variable*. The nine items of this test were found to be reliable, Cronbach's *alpha* = 0.79. The marking of the first procedural-knowledge test was reliable as well, *r* (15) = 0.78. Descriptive statistics for the test's scores are reported in Table 5.20.

Table 5.20
Descriptive Statistics for the Scores of the First Procedural-Knowledge Measurement (Second Round)

Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Koios	21	7.40	5.33	1.16	
Glossomatheia	25	11.21	5.37	1.07	-0.71
MicroworldsPro	19	11.42	6.15	1.41	-0.70
Total	65	10.04	5.81	0.72	

Note. Because not all pupils attended the test session, sample size was reduced in Koios and MicroworldsPro groups.

As can be seen in Table 5.20, the mean scores of the three groups were relatively low and ranged approximately from 7 to 11. More specifically, the *mean* and *SD* for the Koios, the Glossomatheia and the MicroworldsPro groups were 7.40 and 5.33, 11.21 and 5.37 and, 11.42 and 6.15, respectively. The difference between the means of the Koios and the Glossomatheia groups indicated a medium to large effect size, when compared with Cohen's (1988) recommendation. This was also the case for the difference of group means between the Koios and the MicroworldsPro groups. The results of HMR analysis with *Participants GPA without*

CS in the first model, programming tool in the second and procedural-knowledge scores as the dependent variable are presented in Table 5.21.

Table 5.21
Results of Regression Analysis for the Scores of the First Procedural-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-14.38	4.39		**-3.28
	Participant's GPA without CS	1.44	0.26	0.58	***5.62
Model 2					
	Constant	-15.50	4.24		**-3.65
	Participant's GPA without CS	1.38	0.25	0.55	***5.55
	Glossomatheia vs Koios	2.99	1.37	0.25	*2.18
	MicroworldsPro vs Koios	3.50	1.46	0.28	*2.40

Note. R^2 = 0.33 for Model 1; ΔR^2 = 0.07 for Model 2 ($p < 0.05$). * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

Analysis of data showed that *Participant's GPA without CS* alone explained 33% of variance in the scores of the first procedural-knowledge measurement, $R^2 = 0.33$, F (1, 63) = 31.57, $p < 0.001$. *Participant's GPA without CS*, as a predictor, was statistically significant. The results of the second model of regression showed that programming tool accounted for an extra 7% of variance in the scores, $\Delta R^2 = 0.07$, F_{change} (2, 61) = 3.50, $p < 0.05$. This extra proportion (7%) was statistically significant and thus, programming tool was a significant predictor. *GPA without CS* was a significant predictor in this model as well, t (61) = 5.55, $p < 0.001$. Both Glossomatheia and MicroworldsPro users seem to have performed significantly better in this test than Koios users, t (61) = 2.18, $p < 0.05$ and t (61) = 2.40, $p < 0.05$, respectively. Therefore, hypothesis H1 was rejected. The experiment plan did not include a declarative-knowledge measurement at the same time with this first procedural-knowledge measurement and thus, it was not feasible to investigate any mediation effect.

5.3.1.3 Second procedural-knowledge measurement.

After five weeks of lessons since the beginning of the study, procedural knowledge was measured for a second time. This test took approximately 15-20 minutes to complete and measured the procedural knowledge of *integer, arithmetic expression, string, output-statement, input-statement, variable and assignment-statement.* The ten items of this test were found to be highly reliable, Cronbach's *alpha* = 0.85. The reliability of marking this test was high, *r* (15) = 0.84. Table 5.22 presents descriptive statistics for this test's scores.

Table 5.22
Descriptive Statistics for the Scores of the Second Procedural-Knowledge Measurement (Second Round)

Group	*N*	*Mean*	*SD*	*SE*	*Cohen's d*
Koios	21	10.73	5.53	1.21	
Glossomatheia	26	13.32	5.35	1.05	-0.48
MicroworldsPro	19	11.76	4.62	1.06	-0.20
Total	66	12.05	5.25	0.65	

Note. Because not all pupils attended the test session, sample size was reduced in Glossomatheia and MicroworldsPro groups.

From Table 5.22, it is shown that the mean scores for the three groups ranged approximately from 11 to 13. In particular, the *mean* for the Koios, the Glossomatheia and the MicroworldsPro groups was 10.73, 13.32 and 11.76, respectively, while the *SD* was 5.52, 5.35 and 4.62, respectively. The effect size represented by the comparison of means between the Koios and the Glossomatheia groups was medium, based on Cohen's (1988) standards. However, the difference in the group means between Koios and MicroworldsPro indicated a small effect size. Table 5.23 presents the results of HMR analysis with *Participant's GPA without CS* and the dummy variables of programming tool as predictors and this measurement's scores as the dependent variable.

Table 5.23
Results of Regression Analysis for the Scores of the Second Procedural-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-12.77	3.86		**-3.31
	Participant's GPA without CS	1.46	0.23	0.63	***6.48
Model 2					
	Constant	-12.77	3.87		**-3.30
	Participant's GPA without CS	1.44	0.23	0.62	***6.25
	Glossomatheia vs Koios	1.30	1.23	0.12	1.06
	MicroworldsPro vs Koios	-0.23	1.32	-0.02	-0.17

Note. $R^2 = 0.40$ for Model 1; $\Delta R^2 = 0.02$ for Model 2 ($p > 0.05$). ** $p < 0.01$, *** $p < 0.001$

The results of Table 5.23 shows that *Participant's GPA without CS* was a significant predictor in the first model and explained 40% of variance in the scores, $R^2 = 0.40$, F (1, 64) = 41.96, $p < 0.001$. In the second model, the additional proportion of variance explained by programming tool was 2% and it was non-significant, $\Delta R^2 = 0.02$, $F_{change} < 1$. Hence, programming tool had no significant effect on this test's scores. However, *GPA without CS* was a significant predictor in the second model, t (62) = 6.25, $p < 0.001$. No significant difference in the mean performance between the Glossomatheia and the Koios groups were observed, t (62) = 1.06, $p > 0.05$. This was also the case for the difference in the mean scores between the MicroworldsPro and the Koios groups, $|t| < 1$. Hence, hypothesis H1 had to be rejected. Moreover, the first step of mediation analysis was not successful, because no significant effect of programming tool (IV) on the test scores (DV) was detected.

5.3.1.4 Third procedural-knowledge measurement.

The third and final measurement of procedural knowledge was conducted at the end of the study, namely after nine weeks of lessons. It took approximately 20-25 minutes to administer and measured all the 13 programming concepts. Reliability analysis showed that the third procedural-knowledge test was reliable, Cronbach's *alpha* = 0.78. However, an improvement in this test's reliability, Cronbach's *alpha* = 0.82, would be achieved if the last item were deleted. Because the pattern of the results was identical, regardless of whether the last item was removed, it was decided to report the results of the analysis of the scores with highest reliability. Hence, in this one and following analyses, the scores of the third procedural-knowledge test were computed based on all the items of the test except for the last one. The marking procedure of the third measurement's procedural-knowledge tests was highly reliable, *r* (15) = 0.81. Table 5.24 presents descriptive statistics for the grades of this measurement.

Table 5.24
Descriptive Statistics for the Scores of the Third Procedural-Knowledge Measurement (Second Round)

Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Koios	22	7.28	5.11	1.09	
Glossomatheia	24	10.27	4.63	0.94	-0.62
MicroworldsPro	24	7.45	5.44	1.11	-0.03
Total	70	8.36	5.18	0.62	

The data in Table 5.24 show that the scores of the final measurement were low to medium and ranged approximately from 7 to 10. More specifically, the *mean* and *SD* for the Koios, the Glossomatheia and the MicroworldsPro groups were 7.28 and 5.11, 10.27 and 4.63 and, 7.45 and 5.44, respectively. The difference in the mean score between the Koios and the Glossomatheia groups presented a medium to large effect size, according to Cohen's (1988) suggestions. However, the effect size indicated by the mean difference between the Koios and the MicroworldsPro groups was small. In Table 5.25, the results of HMR analysis with this test's scores as the dependent variable, *Participant's GPA* as the predictor in the first model and the dummy variables of programming tool as the predictor in the second model, are reported.

Table 5.25
Results of Regression Analysis for the Scores of the Third Procedural-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-15.51	3.75		***-4.13
	Participant's GPA without CS	1.42	0.22	0.61	***6.42
Model 2					
	Constant	-15.44	3.67		***-4.21
	Participant's GPA without CS	1.39	0.22	0.60	***6.37
	Glossomatheia vs Koios	2.03	1.20	0.19	1.69
	MicroworldsPro vs Koios	-0.58	1.19	-0.05	-0.49

Note. $R^2 = 0.38$ for Model 1; $\Delta R^2 = 0.05$ for Model 2 ($p > 0.05$). *** $p < 0.001$

The results of the first model showed that *Participant's GPA without CS* accounted for the 38% of variance in the scores, $R^2 = 0.38$, F (1, 68) = 41.21, $p < 0.001$. *GPA without CS* was a significant predictor, t (68) = 6.42, $p < 0.001$. Programming tool explained an additional 5% of variance in the scores, $\Delta R^2 = 0.05$, $F_{change} < 1$. This percentage was non-significant and thus, programming tool had no significant effect on the scores. Nonetheless, *GPA without CS* was a significant predictor in the second model as well, t (66) = 6.37, $p < 0.001$. The difference in the mean performance between the Glossomatheia and the Koios groups was non-significant t (66) = 1.69, $p > 0.05$, as was the difference in the mean performance between the MicroworldsPro and the Koios groups, $|t| < 1$. Based on the latter results, hypothesis H1 was rejected. Because no significant effect of programming tool (IV) on procedural-knowledge scores (DV) was detected, the first step of mediation analysis could not be supported.

5.3.2 Second Step of Mediation Analysis – Declarative Knowledge

As with the case of the first round of data collection, the selected five variables to represent the measured declarative knowledge were: *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information*. For a more detailed discussion on these variables and the selection process, see Section 4.4.1.2.

Based on the results of the three declarative-knowledge measurements, the validity of hypothesis H2 and the second step of mediation analysis were tested. According to hypothesis H2, Koios users construct richer mental models (declarative knowledge) in the domain of introductory programming than Glossomatheia- and MicroworldsPro users. The second step of mediation analysis determines if programming tool (IV) has a significant effect on mental models (M).

5.3.2.1 Descriptive statistics for declarative-knowledge measurement.

The descriptive statistics for the three groups over the three declarative-knowledge measurements for *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* are presented in Figure 5.8, Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12, respectively. The maximum values for *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* were 19.00, 14.37, 1.00, 0.86 and 59.22, respectively.

The minimum value for *Links in Common*, *Similarity* and *Information* was zero. The minimum values for *Difference between Observed and Expected Number of Links in Common* and *Difference between Observed and Expected Similarity by Chance* were -5.12 and -0.15, respectively. These two values were the minimum observed for each variable in this round of data collection. For a short discussion on the computation of minimum and maximum values, see Section 5.2.2.1. The minimum and maximum value of the y-axis in each of the following figures is set to the minimum and maximum value of the variable that is analysed.

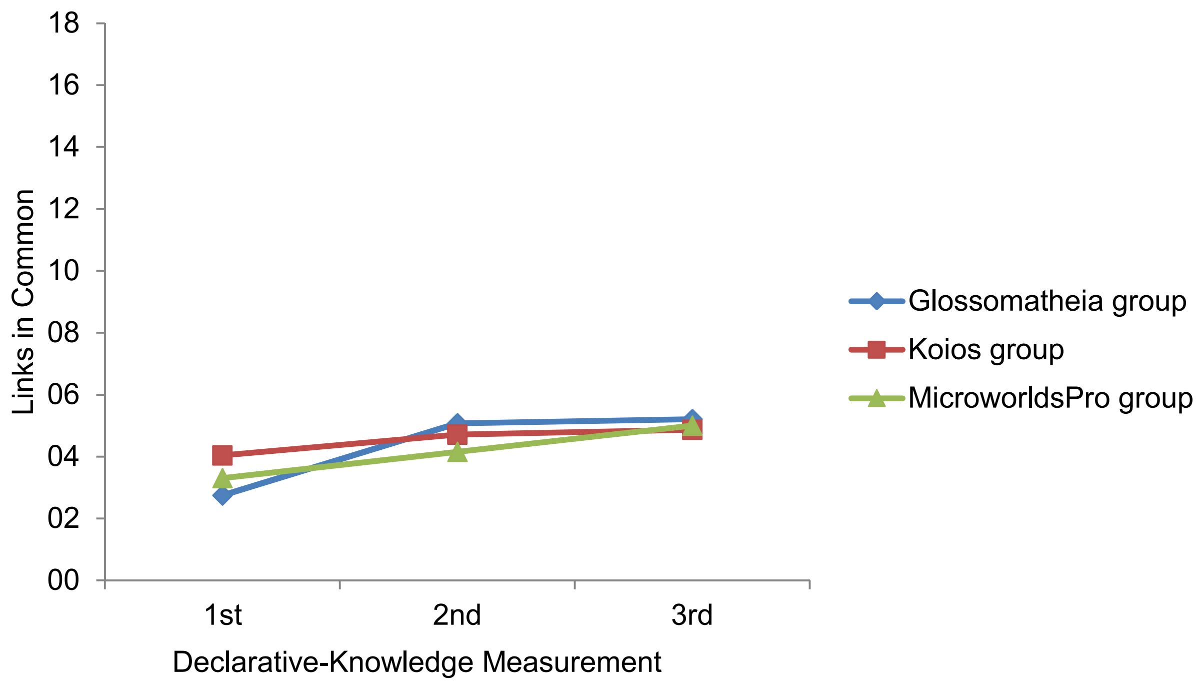


Figure 5.8. Mean scores of Links in Common for each group over the three measurements (second round).

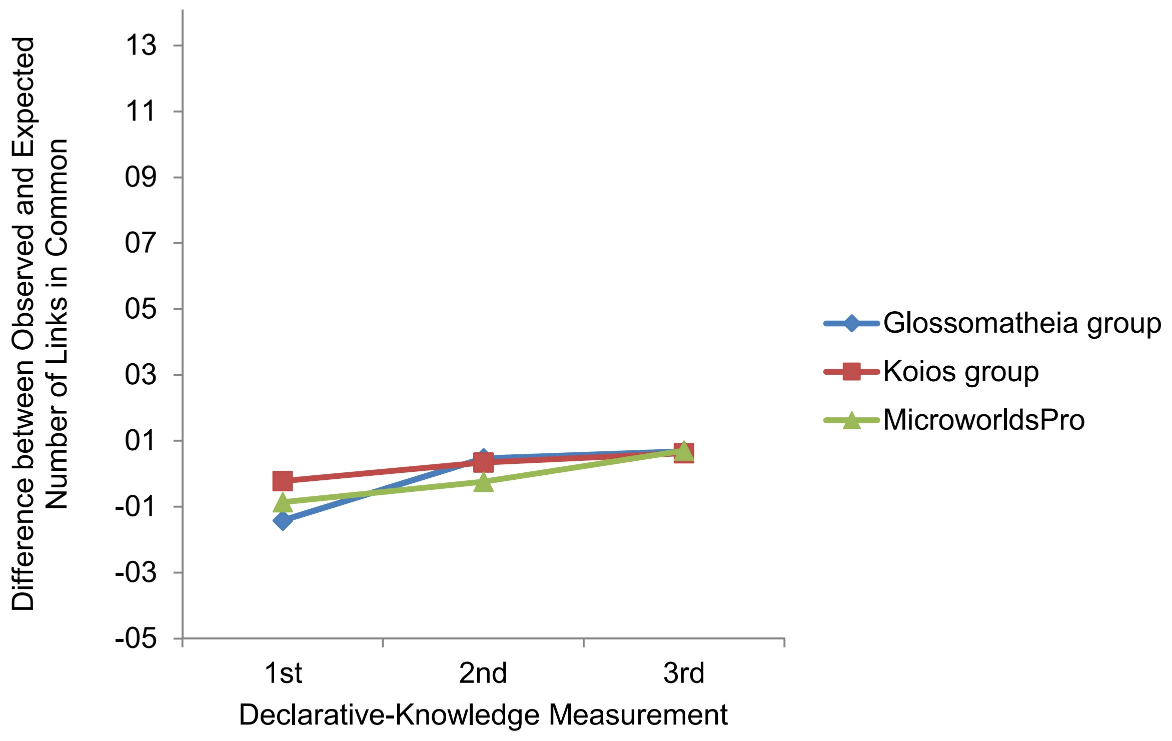


Figure 5.9. Mean scores of Difference between Observed and Expected Number of Links in Common for each group over the three measurements (second round).

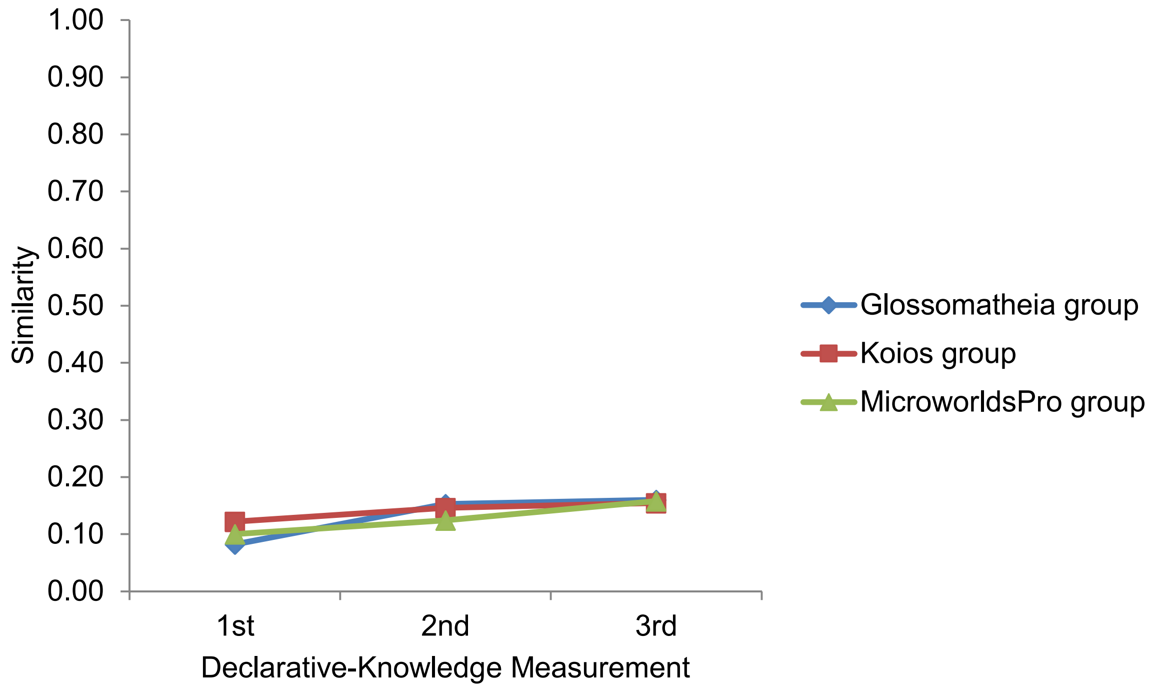


Figure 5.10. Mean scores of Similarity for each group over the three measurements (second round).

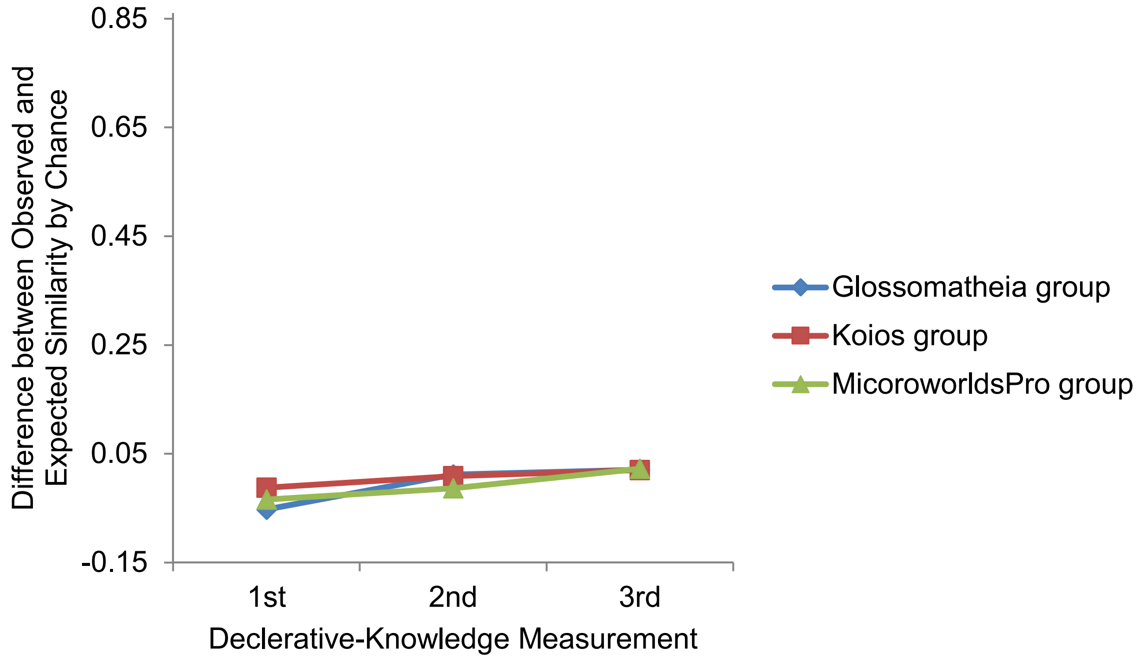


Figure 5.11. Mean scores of Difference between Observed and Expected Similarity by Chance for each group over the three measurements (second round).

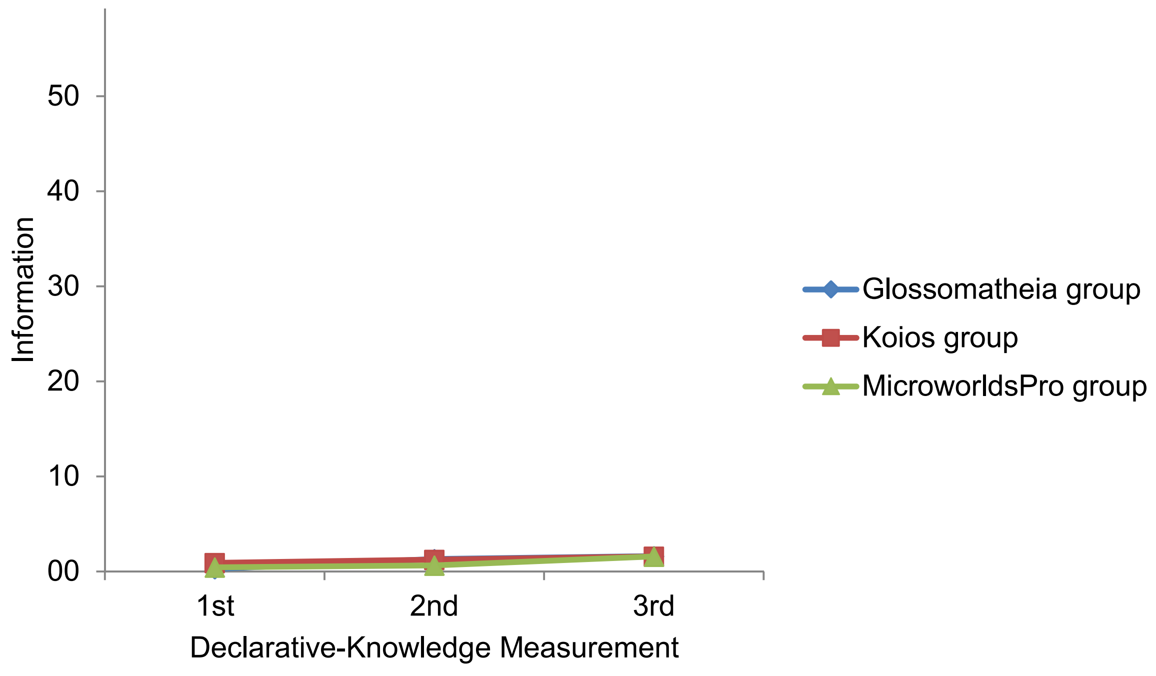


Figure 5.12. Mean scores of Information for each group over the three measurements (second round).

As can be seen from the five figures, the scores of the three groups over the three measurements were overall low for each of the variables, possibly indicating a floor effect. The mean score of each group in the second measurement seems relatively higher than its mean score in the first measurement for all five variables. An apparent relative improvement in the mean score of each group was observed from the second to the third measurement for all variables, as well.

The five variables of declarative knowledge were used in inferential statistical analysis. In the following three sections the results of analysing the data collected in the first, second and third declarative-knowledge measurement are presented.

5.3.2.2 First declarative-knowledge measurement.

The first declarative-knowledge measurement took place at the beginning of the second round of data collection. Table 5.26 presents descriptive statistics for the five variables.

The mean scores of the three groups for the five variables were overall low. According to Cohen's (1988) recommendations, medium to large effect sizes seem to have been observed based on the differences of the mean scores between the Koios and the Glossomatheia groups for all variables. More particularly, the *means* for the Koios, the Glossomatheia and the MicroworldsPro groups for *Links In Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference between Observed and Expected* Similarity and *Information* were 4.04, 2.75 and 3.30; -0.32, -1.51 and -0.96; 0.12, 0.08 and 0.10; -0.01, -0.05 and -0.03; and 0.90, 0.27 and 0.43, respectively. The *SD* for the same groups and variables were 2.29, 1.16 and 1.22, 1.63, 1.08 and 1.13; 0.06, 0.04 and 0.04; 0.06, 0.04 and 0.04; and 1.12, 0.40 and 0.49, respectively. Moreover, small to medium effect sizes were observed in the mean difference between the Koios and the MicroworldsPro groups for the five variables. The analysis of this measurement's data followed the analysis plan presented in Section 4.4.2.2.2.

Table 5.26
Descriptive Statistics for the Outcome Variables of the First Declarative-Knowledge Measurement (Second Round)

Variable	Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Links in common	Koios	24	4.04	2.29	0.47	
	Glossomatheia	20	2.75	1.16	0.26	0.69
	MicroworldsPro	23	3.30	1.22	0.25	0.40
	Total	67	3.40	1.73	0.21	
Difference between observed and expected number of links in common	Koios	24	-0.32	1.63	0.33	
	Glossomatheia	20	-1.51	1.08	0.24	0.85
	MicroworldsPro	23	-0.96	1.13	0.24	0.46
	Total	67	-0.90	1.39	0.17	
Similarity	Koios	24	0.12	0.06	0.01	
	Glossomatheia	20	0.08	0.04	0.01	0.75
	MicroworldsPro	23	0.10	0.04	0.01	0.41
	Total	67	0.10	0.05	0.01	
Difference between observed and expected similarity	Koios	24	-0.01	0.06	0.01	
	Glossomatheia	20	-0.05	0.04	0.01	0.82
	MicroworldsPro	23	-0.03	0.04	0.01	0.45
	Total	67	-0.03	0.05	0.01	
Information	Koios	24	0.90	1.12	0.23	
	Glossomatheia	20	0.27	0.40	0.09	0.72
	MicroworldsPro	23	0.43	0.49	0.10	0.53
	Total	67	0.55	0.80	0.10	

Note. Because not all pupils attended the test session, sample size was reduced in Glossomatheia group.

For more details about using *Participant's GPA of previous year* instead of *Participant's GPA without CS* as covariate in the analysis of the declarative-knowledge data of the first measurement, see Section 4.4.2.1.1. The significance level used for *Difference between Observed and Expected Number of Links in*

Common was 0.05. For the remaining four variables, it was adjusted to 0.025, due to violation of the assumption of homogeneity of variance.

The results of ANOVA for *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* were F (2, 64) = 3.30, $p > 0.025$; F (2, 64) = 4.45, $p < 0.05$, $\varepsilon^2 = 0.11$; F (2, 64) = 3.72, $p > 0.025$; F (2, 64) = 4.29, $p < 0.025$, $\varepsilon^2 = 0.11$; and F (2, 64) = 4.11, $p < 0.025$, $\varepsilon^2 = 0.10$, respectively. These results seem to indicate a significant difference between means of the three groups for *Difference between Observed and Expected Number of Links in Common*, *Difference between Observed and Expected Similarity by Chance* and *Information*.

The results of planned contrasts between the Koios and the Glossomatheia groups for the five variables were t (64) = 2.57, $p < 0.025$, $d = 0.69$; t (64) = 2.99, $p < 0.01$, $d = 0.85$; t (64) = 2.71, $p < 0.01$, $d = 0.75$; t (64) = 2.91, $p < 0.01$, $d = 0.82$; and t (64) = 2.73, $p < 0.01$, $d = 0.72$, respectively. The outcomes of planned comparisons between the Koios and the MicroworldsPro groups for the five measures were t (64) = 1.51, $p > 0.05$; t (64) = 1.67, $p > 0.05$; t (64) = 1.54, $p > 0.05$; t (64) = 1.65, $p > 0.05$; and t (64) = 2.08, $p > 0.025$, respectively.

As this measurement occurred before any experimental manipulation, these results revealed that the participants of this study did not come from the same population. Specifically, students allocated to the Koios group performed significantly better than the students of the Glossomatheia group in all variables, while no significant differences were found between students allocated to the Koios and the MicroworldsPro groups. Participants – of the Koios and the Glossomatheia groups – coming from different populations could also explain the medium to large effect sizes that were observed for the differences in the outcome variables between these two groups. Moreover, because this measurement was taken before any experimental manipulation, it was meaningless to check the validity of the second step of mediation.

5.3.2.3 Second declarative-knowledge measurement.

The second measurement of declarative knowledge was taken at the sixth week of lessons and just before the second procedural-knowledge test. The descriptive statistics for the variables, presented in Table 5.27, revealed an overall low achievement in the five measures by the three groups. More specifically, the *means* for the Koios, the Glossomatheia and the MicroworldsPro groups for *Links In Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference between Observed and Expected* Similarity and *Information* were 4.71, 5.08 and 4.16; 0.25, 0.37 and -0.34; 0.15, 0.15 and 0.12; 0.01, 0.01 and -0.01; and 1.25, 1.30 and 0.64, respectively. The *SD* for the same groups and variables were 1.93, 2.04 and 1.38; 1.67, 1.51 and 0.98; 0.06, 0.06 and 0.04; 0.06, 0.06 and 0.03; 1.11, 1.41 and 0.41, respectively.

Following Cohen's (1988) suggestions, the differences in the mean value between the Koios and the Glossomatheia groups for the five variables represented small effect sizes. The effect sizes indicated by the mean differences between the Koios and the MicroworldsPro groups were low to medium for the first four variables. The effect size of the mean difference in *Information* between the Koios and the MicroworldsPro groups was medium to large.

Table 5.27
Descriptive Statistics for the Outcome Variables of the Second Declarative-Knowledge Measurement (Second Round)

Variable	Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Links in common	Koios	21	4.71	1.93	0.42	
	Glossomatheia	26	5.08	2.04	0.40	-0.18
	MicroworldsPro	19	4.16	1.38	0.32	0.33
	Total	66	4.70	1.85	0.23	
Difference between observed and expected number of links in common	Koios	21	0.25	1.67	0.36	
	Glossomatheia	26	0.37	1.51	0.30	-0.07
	MicroworldsPro	19	-0.34	0.98	0.23	0.43
	Total	66	0.12	1.45	0.18	
Similarity	Koios	21	0.15	0.06	0.01	
	Glossomatheia	26	0.15	0.06	0.01	-0.11
	MicroworldsPro	19	0.12	0.04	0.01	0.43
	Total	66	0.14	0.05	0.01	
Difference between observed and expected similarity	Koios	21	0.01	0.06	0.01	
	Glossomatheia	26	0.01	0.06	0.01	-0.05
	MicroworldsPro	19	-0.01	0.03	0.01	0.47
	Total	66	0.00	0.05	0.01	
Information	Koios	21	1.25	1.11	0.24	
	Glossomatheia	26	1.30	1.41	0.28	-0.04
	MicroworldsPro	19	0.64	0.41	0.09	0.71
	Total	66	1.09	1.13	0.14	

Note. Because not all pupils attended the test session, sample size was reduced in Koios and MicroworldsPro groups.

For the analysis of *Links in Common* a significance level of 0.05 was used. Due to the violation of homogeneity of variance by the remaining four variables, a significance level of 0.025 was used for their analysis. Based on the analysis plan described in Section 4.4.2.2.2, the results of analysing the data of this measurement follow. The HMR results for *Links in Common* are presented in Table 5.28.

Table 5.28
Results of Regression Analysis for Links in Common of the Second Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	3.89	0.51		***7.59
	Links in Common(First Declarative-Knowledge Measurement)	0.22	0.13	0.21	1.65
Model 2					
	Constant	3.66	0.69		***5.30
	Links in Common(First Declarative-Knowledge Measurement)	0.25	0.14	0.25	1.82
	Glossomatheia vs Koios	0.60	0.58	0.16	1.03
	MicroworldsPro vs Koios	-0.53	0.58	-0.14	-0.91

Note. $R^2 = 0.05$ for Model 1; $\Delta R^2 = 0.06$ for Model 2 ($p > 0.05$). *** $p < 0.001$

The data of *Links in Common* collected in the first measurement accounted for 5% of variance in the data of the second measurement, $R^2 = 0.05$, $F\ (1, 56) = 2.71$, $p > 0.05$; this predictor was non-significant. When programming tool was added in the model, it explained an extra 6% of variance in *Links in Common* of the second measurement, $\Delta R^2 = 0.06$, $F_{change}\ (2, 54) = 1.91$, $p > 0.05$, which was a non-significant proportion, as well. Thus, programming tool was not a significant predictor; neither was *Links in Common* of the first measurement, $t\ (54) = 1.82$, $p > 0.05$. The mean difference between the Glossomatheia and the Koios groups was non-significant, $t\ (54) = 1.03$, $p > 0.05$, as was the mean difference between the MicroworldsPro and the Koios groups, $|t| < 1$. The outcome of HMR analysis for *Difference between Observed and Expected Number of Links in Common* of the second measurement is shown in Table 5.29.

Table 5.29
Results of Regression Analysis for Difference between Observed and Expected Number of Links in Common of the Second Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.04	0.21		0.18
	Difference between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	0.07	0.13	0.07	0.53
Model 2					
	Constant	0.25	0.30		0.85
	Difference between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	0.04	0.15	0.04	0.27
	Glossomatheia vs Koios	-0.10	0.47	-0.04	-0.22
	MicroworldsPro vs Koios	-0.70	0.46	-0.24	-1.51

Note. R^2 = 0.01 for Model 1; ΔR^2 = 0.05 for Model 2 (p > 0.05).

The data of *Difference between Observed and Expected Number of Links in Common* measured the first time explained 1% of variance in this measurement's data, R^2 = 0.01, F < 1. Consequently, this was a non-significant predictor. In the second model, an additional 5% of variance was explained by programming tool, ΔR^2 = 0.05, F_{change} (2, 54) = 1.38, p > 0.05, but this percentage was not significant either. Therefore, programming tool was not a significant predictor. In the second model, the *Difference between Observed and Expected Number of Links in Common* of the first measurement was not a significant predictor, as well, $|t|$ < 1. No significant differences in the mean performance was observed between the Glossomatheia and the Koios groups, $|t|$ < 1, or between the MicroworldsPro and the Koios groups, t (54) = -1.51, p > 0.05. The results of HMR analysis of *Similarity* scores are reported in Table 5.30.

Table 5.30
Results of Regression Analysis for Similarity of the Second Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.12	0.01		***8.09
	Similarity (First Declarative-Knowledge Measurement)	0.16	0.13	0.16	1.25
Model 2					
	Constant	0.13	0.02		***6.06
	Similarity (First Declarative-Knowledge Measurement)	0.16	0.14	0.17	1.17
	Glossomatheia vs Koios	0.01	0.02	0.05	0.32
	MicroworldsPro vs Koios	-0.02	0.02	-0.21	-1.38

Note. R^2 = 0.03 for Mode 1; ΔR^2 = 0.06 for Model 2 (p > 0.05). *** p < 0.001

The first regression model showed that *Similarity* of the first measurement was not a significant predictor; it accounted for 3% of the variance in *Similarity* of the second measurement, R^2 = 0.03, F (1, 56) = 1.56, p > 0.05. The addition of programming tool in the second model, explained an extra 6% of the variance in *Similarity*, ΔR^2 = 0.06, F_{change} (2, 54) = 1.68, p > 0.05. This percentage was non-significant. Hence, in the second model, neither programming tool, nor *Similarity* of the first measurement, t (54) = 1.17, p > 0.05, were significant predictors. The mean score of the Koios group was not significantly different than the mean score of the Glossomatheia group, $|t|$ < 1 and the mean score of the MicroworldsPro group, t (54) = -1.38, p > 0.05.

The next HMR analysis was conducted with *Difference between Expected and Observed Similarity by Chance* as the dependent variable with this variable's data obtained from first measurement as the predictor in the first model and programming tool in the second. The results are presented in Table 5.31.

Table 5.31
Results of Regression Analysis for Difference between Expected and Observed Similarity by Chance of the Second Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.00	0.01		0.09
	Difference between Expected and Observed Similarity by Chance (First Declarative-Knowledge Measurement)	0.09	0.13	0.09	0.68
Model 2					
	Constant	0.01	0.01		0.88
	Difference between Expected and Observed Similarity by Chance (First Declarative-Knowledge Measurement)	0.05	0.14	0.05	0.35
	Glossomatheia vs Koios	-0.01	0.02	-0.06	-0.35
	MicroworldsPro vs Koios	-0.03	0.02	-0.25	-1.64

Note. $R^2 = 0.01$ for Model 1; $\Delta R^2 = 0.05$ for Model 2 ($p > 0.05$).

The results revealed that *Difference between Expected and Observed Similarity by Chance* of the first measurement was a non-significant predictor and it explained 1% of the variance in the second measurement's data, $R^2 = 0.01$, $F < 1$. Programming tool accounted for an additional 5% of this variable's variance, $\Delta R^2 = 0.05$, F_{change} (2, 54) = 1.52, $p > 0.05$. The extra proportion (5%) explained by programming tool was not significant either. Thus, in the second model, programming tool was a non-significant predictor and so was *Difference between Expected and Observed Similarity by Chance* of the first measurement, $|t| < 1$. The mean difference between the Glossomatheia and the Koios groups and between the MicroworldsPro and the Koios groups were both non-significant, $|t| < 1$ and t (54) = -1.64, $p > 0.05$, respectively.

The final variable measured in the second declarative-knowledge measurement was *Information.* A HMR was used for the analysis of *Information* and its results are reported in Table 5.32.

Table 5.32
Results of Regression Analysis for Information of the Second Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.91	0.14		***6.46
	Information (First Declarative-Knowledge Measurement)	0.08	0.14	0.07	0.54
Model 2					
	Constant	1.28	0.24		***5.35
	Information (First Declarative-Knowledge Measurement)	-0.03	0.15	-0.03	-0.21
	Glossomatheia vs Koios	-0.30	0.29	-0.17	-1.05
	MicroworldsPro vs Koios	-0.71	0.30	-0.37	*#-2.40

Note. R^2 = 0.01 for Model 1; ΔR^2 = 0.10 for Model 2 ($p > 0.05$). *# $p < 0.025$, *** $p < 0.001$

The results showed that *Information* of the first measurement explained 1% of the variance in this measurement's *Information* and that it was a non-significant predictor, $R^2 = 0.01$, $F < 1$. In the second model, programming tool accounted for an extra 10% of variance in *Information*, $\Delta R^2 = 0.10$, F_{change} (2, 54) = 2.92, $p > 0.05$, but this percentage was not significant either. Thus, programming tool was not a significant predictor. Neither was *Information* of the first measurement in the second model, $|t| < 1$. No significant difference was observed in the mean scores between the Glossomatheia and the Koios groups, *t* (54) = -1.05, $p > 0.05$. However, based on the scores of *Information* measured in this occasion, Koios users seemed to have constructed significantly richer mental models than MicroworldsPro users, *t* (54) = -2.40, $p < 0.25$.

Because the difference between the means of the Koios and the MicroworldsPro groups represented a medium effect size for *Difference between Observed and Expected Number of Links in Common*, *Similarity* and *Difference between Expected and Observed Similarity by Chance*, but the analysis of these variables yielded no significant results, a retrospective power analysis was conducted to ascertain that the statistical tests used in these analyses had enough statistical power to detect an effect. In particular, the power of the *t*-test between the Koios and the MicroworldsPro groups for each of these three variables was investigated using G*Power (Faul, Erdfelder, Buchner & Lang, 2009; Faul, Erdfelder, Lang & Buchner, 2007). This experiment was designed to be able to detect a large effect size, namely for the *t*-test the intended effect size was *d* = 0.80. The power for the intended effect sizes of the *t*-tests that were used to compare the Koios and the MicroworldsPro groups for *Difference between Observed and Expected Number of Links in Common*, *Similarity* and *Difference between Expected and Observed Similarity by Chance* was 0.58. This value suggested that these *t*-tests had medium to high statistical power, but less than the conventional power level of 0.80.

The results of the second declarative-knowledge test revealed that the declarative-knowledge variables of the first measurement did not significantly affect the variables measured in this occasion. The mean performance of the Glossomatheia and the Koios groups was not significantly different for each of the five variables. This was the case for the mean performance of the MicroworldsPro and the Koios groups for the first four variables. Nonetheless, the mental models of Koios users were significantly better than those of MicroworldsPro users, when measured by *Information*. Overall, with the exception of the latter case, Koios users did not seem to have constructed better mental models than Glossomatheia- and MicroworldsPro users, and therefore, hypothesis H2 had to be rejected. The findings also revealed that programming tool (IV) was not a significant predictor of the five variables and consequently, of participants' declarative knowledge (M). Thus, programming tool did not have a significant effect on declarative knowledge and therefore, the conditions described in the second step of mediation were not met. To further investigate the relationship between this measurement's declarative-knowledge data (M) and procedural-knowledge scores (DV), the correlation of each variable with the procedural-knowledge scores was calculated. However, all correlations (< 0.20) were non-significant and therefore the existence of a mediation effect could not be supported.

5.3.2.4 Third declarative-knowledge measurement.

The final declarative-knowledge measurement was taken at the end of this round, after nine weeks of lessons and just before the final procedural-knowledge test. Descriptive statistics for this measurement' variables are presented in Table 5.33. From Table 5.33, it can be seen that participants' performance was rather low in all variables, regardless of the group they belonged to. In particular, the *means* for the Koios, the Glossomatheia and the MicroworldsPro groups for *Links In Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity*, *Difference between Observed and Expected* Similarity and *Information* were 4.86, 5.21 and 5.00; 0.52, 0.58 and 0.61; 0.15, 0.16 and 0.16; 0.02, 0.02 and 0.02; and 1.56, 1.61 and 1.58, respectively. The *SD* for the same groups and variables were 2.14, 2.23 and 1.84; 1.93, 1.80 and 1.58; 0.07, 0.07 and 0.06; 0.07, 0.07 and 0.06; and 1.44, 1.92 and 2.06, respectively. Small effect sizes were observed for the differences in the mean values between the Koios and the Glossomatheia groups for all variables, based on Cohen's *d* (1988). This was the case for the mean differences between the Koios and the MicroworldsPro groups for all variables. A significance level of 0.05 was used for this measurement's analyses, because only normality was violated for all variables.

Table 5.33
Descriptive Statistics for the Outcome Variables of the Third Declarative-Knowledge Measurement (Second Round)

Variable	Group	*n*	*Mean*	*SD*	*SE*	*Cohen's d*
Links in common	Koios	22	4.86	2.14	0.46	
	Glossomatheia	24	5.21	2.23	0.45	-0.16
	MicroworldsPro	24	5.00	1.84	0.38	-0.07
	Total	70	5.03	2.05	0.25	
Difference between observed and expected number of links in common	Koios	22	0.52	1.93	0.41	
	Glossomatheia	24	0.58	1.80	0.37	-0.03
	MicroworldsPro	24	0.61	1.58	0.32	-0.05
	Total	70	0.57	1.75	0.21	
Similarity	Koios	22	0.15	0.07	0.01	
	Glossomatheia	24	0.16	0.07	0.01	-0.08
	MicroworldsPro	24	0.16	0.06	0.01	-0.05
	Total	70	0.16	0.07	0.01	
Difference between observed and expected similarity	Koios	22	0.02	0.07	0.01	
	Glossomatheia	24	0.02	0.07	0.01	-0.01
	MicroworldsPro	24	0.02	0.06	0.01	-0.04
	Total	70	0.02	0.07	0.01	
Information	Koios	22	1.56	1.44	0.31	
	Glossomatheia	24	1.61	1.92	0.39	-0.03
	MicroworldsPro	24	1.58	2.06	0.42	-0.01
	Total	70	1.58	1.81	0.22	

This analysis followed the plan presented in Section 4.4.2.2.2. The HMR results for *Links in Common* of the third measurement are presented in Table 5.34. The data of *Links in Common* of the first measurement explained 1% of the variance in this measurement's *Links in Common* and was not a significant predictor, $R^2 = 0.01$, $F < 1$. The inclusion of programming tool in the model explained an additional 2% of the variance, which was not significant, $\Delta R^2 = 0.02$, $F_{change} < 1$.

Table 5.34
Results of Regression Analysis for Links in Common of the Third Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	4.66	0.57		***8.18
	Links in Common (First Declarative-Knowledge Measurement)	0.09	0.15	0.08	0.61
Model 2					
	Constant	4.30	0.79		***5.42
	Links in Common (First Declarative-Knowledge Measurement)	0.13	0.16	0.12	0.86
	Glossomatheia vs Koios	0.65	0.68	0.15	0.96
	MicroworldsPro vs Koios	0.03	0.63	0.01	0.05

Note. $R^2 = 0.01$ for Model 1; $\Delta R^2 = 0.02$ for Model 2 ($p > 0.05$). *** $p < 0.001$

Therefore, programming tool was not a significant predictor. This was also the case for *Links in Common* of the first measurement, $|t| < 1$. The mean scores of the Glossomatheia and the Koios groups, as well as of the MicroworldsPro and the Koios groups, were not significantly different, $|t| < 1$ and $|t| < 1$, respectively. The HMR results of *Difference between Observed and Expected Number of Links in Common* are reported in Table 5.35.

The results from Model 1 show that the data of this variable's first measurement explained none of the variance in the data of the third measurement and it was a non-significant predictor, $R^2 = 0.00$, $F < 1$. The second model revealed that 1% of the variance in *Difference between Observed and Expected Number of Links in Common* of the third measurement was explained by programming tool, $\Delta R^2 = 0.01$, $F_{change} < 1$. This percentage was not significant and therefore programming tool was not a significant predictor. The *Difference between Observed and Expected Number of Links in Common* from the first measurement was not a significant predictor in the second model either, $|t| < 1$.

Table 5.35
Results of Regression Analysis for Difference between Observed and Expected Number of Links in Common of the Third Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.61	0.26		*2.32
	Difference between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	0.05	0.16	0.04	0.29
Model 2					
	Constant	0.54	0.39		1.37
	Difference between Observed and Expected Number of Links in Common (First Declarative-Knowledge Measurement)	0.08	0.18	0.06	0.43
	Glossomatheia vs Koios	0.29	0.62	0.08	0.47
	MicroworldsPro vs Koios	0.04	0.57	0.01	0.07

Note. R^2 = 0.00 for Model 1; ΔR^2 = 0.01 for Model 2 (p > 0.05). * p < 0.05

The mean difference between the Glossomatheia and the Koios groups and between the MicroworldsPro and the Koios groups were both non-significant, $|t| < 1$ and $|t| < 1$, respectively.

According to the analysis plan presented in Section 4.4.2.2.2, in the next three analyses, the covariate *Homework Frequency until the third procedural-knowledge measurement* was also included in the first regression model. The HMR results of *Similarity*, obtained from the third measurement, are presented in Table 5.36. The first model of data analysis showed that the two predictors explained 10% of variance in *Similarity* of the third measurement, $R^2 = .10$, F (2,60) = 3.37, $p < 0.05$. The predictor *Homework Frequency until the third procedural-knowledge measurement* was a significant one, t (60) = 2.56, $p < 0.05$, but *Similarity* of the first measurement was not, $|t| < 1$. The addition of programming tool accounted for an extra 1% of variance in this variable, $\Delta R^2 = 0.01$, $F_{change} < 1$.

Table 5.36
Results of Regression Analysis for Similarity of the Third Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.14	0.02		***7.26
	Homework Frequency until the Third Procedural-Knowledge Measurement	0.01	0.00	0.31	*2.56
	Similarity (First Declarative-Knowledge Measurement)	0.03	0.16	0.02	0.18
Model 2					
	Constant	0.15	0.03		***5.46
	Homework Frequency until the Third Procedural-Knowledge Measurement	0.01	0.00	0.33	*2.54
	Similarity (First Declarative-Knowledge Measurement)	0.02	0.18	0.02	0.11
	Glossomatheia vs Koios	0.00	0.02	0.01	0.04
	MicroworldsPro vs Koios	-0.02	0.02	-0.10	-0.68

Note. R^2 = 0.10 for Model 1; ΔR^2 = 0.01 for Model 2 ($p > 0.05$). * $p < 0.05$, *** $p < 0.001$

This extra proportion was not significant, which means that programming tool was not a significant predictor. This was also the case for *Similarity* of the first measurement, $|t| < 1$, but not for *Homework Frequency until the third procedural-knowledge measurement*, which was a significant predictor in this model, t (58) = 2.54, $p < 0.05$. No significant differences were observed between the means of neither the Glossomatheia and the Koios groups, $|t| < 1$, nor the MicroworldsPro and the Koios groups, $|t| < 1$. Next, *Difference between Expected and Observed Similarity by Chance* was analysed using HMR and the results are presented in Table 5.37. Data analysis revealed that the two predictors of the first model explained 9% of variance in *Difference between Expected and Observed Similarity by Chance* of the third measurement, $R^2 = 0.09$, F (2,60) = 2.81, $p > 0.05$. This variable's data, at the first measurement, was not a significant predictor, $|t| < 1$, while the covariate *Homework Frequency until the third procedural-knowledge measurement* was, t (60) = 2.36, $p < 0.05$.

Table 5.37

Results of Regression Analysis for Difference between Expected and Observed Similarity by Chance of the Third Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	0.01	0.01		0.71
	Homework Frequency until the Third Procedural-Knowledge Measurement	0.01	0.00	0.29	*2.36
	Difference between Expected and Observed Similarity by Chance (First Declarative-Knowledge Measurement)	-0.01	0.17	0.00	-0.04
Model 2					
	Constant	0.01	0.01		0.88
	Homework Frequency until the Third Procedural-Knowledge Measurement	0.01	0.00	0.32	*2.39
	Difference between Expected and Observed Similarity by Chance (First Declarative-Knowledge Measurement)	-0.03	0.19	-0.02	-0.17
	Glossomatheia vs Koios	-0.01	0.02	-0.04	-0.23
	MicroworldsPro vs Koios	-0.01	0.02	-0.10	-0.66

Note. R^2 = 0.09 for Model 1; ΔR^2 = 0.01 for Model 2 ($p > 0.05$). * $p < 0.05$

The inclusion of programming tool in the model explained an additional 1% of variance, which was not significant, $\Delta R^2 = 0.01$, $F_{change} < 1$. *Homework Frequency* was a significant predictor in this model, t (58) = 2.39, $p < 0.05$. However, *Difference between Expected and Observed Similarity by Chance* of the first measurement was not a significant predictor, $|t| < 1$. This was also the case for programming tool, because the additional percent of variance explained by it was non-significant. The mean values of the Glossomatheia and the Koios groups, as well as of the MicroworldsPro and the Koios groups, were not significantly different, $|t| < 1$ and $|t| < 1$, respectively. Finally, the results of HMR analysis for *Information* are reported in Table 5.38.

Table 5.38
Results of Regression Analysis for Information of the Third Declarative-Knowledge Measurement (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	1.33	0.32		***4.17
	Homework Frequency until the Third Procedural-Knowledge Measurement	0.26	0.11	0.29	*2.34
	Information (First Declarative-Knowledge Measurement)	-0.15	0.29	-0.07	-0.54
Model 2					
	Constant	1.58	0.50		**3.18
	Homework Frequency until the Third Procedural-Knowledge Measurement	0.29	0.12	0.32	*2.43
	Information (First Declarative-Knowledge Measurement)	-0.23	0.31	-0.10	-0.73
	Glossomatheia vs Koios	-0.24	0.64	-0.06	-0.37
	MicroworldsPro vs Koios	-0.51	0.61	-0.13	-0.84

Note. R^2 = 0.09 for Model 1; ΔR^2 = 0.01 for Model 2 (p > 0.05). * p < 0.05, ** p< 0.01, *** p < 0.001

The results revealed that the predictors of the first model accounted for 9% of variance in *Information* of the last measurement, R^2 = 0.09, F (2,60) = 2.80, p > 0.05. The covariate *Homework Frequency until the third procedural-knowledge measurement* was a significant predictor, t (60) = 2.34, p < 0.05, whilst *Information* of the first measurement was not, $|t|$ < 1. In the second model, programming tool explained an extra 1% of variance in *Information* of the third measurement, ΔR^2 = 0.01, F_{change} < 1. The additional explained proportion of variance was not significant and thus, neither was programming tool as a predictor. The variable *Information* of the first measurement, was not a significant predictor for this model either, $|t|$ < 1. Nonetheless, *Homework Frequency until the third procedural-knowledge measurement* was a significant predictor, for this model, t (58) = 2.43, p < 0.05. The difference in the mean scores between the Glossomatheia and the Koios groups were non-significant, $|t| < 1$. This was also the case for the difference between the MicroworldsPro and the Koios groups, $|t| < 1$.

The analysis outcome of the final declarative-knowledge measurement revealed that the group mean of Koios was not significantly different from the group means of Glossomatheia and of MicroworldsPro for all variables. Consequently, hypothesis H2 had to be rejected. Furthermore, programming tool (IV) had no significant effect on all declarative-knowledge variables (M). A further investigation of the mediation hypothesis included the estimation of the correlation between each declarative-knowledge variable and procedural-knowledge scores (DV) of this measurement. The extremely low correlations (< 0.06) were not significant, which suggested that procedural knowledge was presumably not affected by declarative knowledge. Therefore, no mediation effect was detected and the second step of mediation could not be established for the final measurement.

5.3.3 Practical Test

The practical test took place the week following the final procedural- and declarative-knowledge measurement. Students had to create three programs using only the programming tool which they had been using during the second round of data collection. For more details about the practical test and its design, see Section 4.3.2.2.3. The programs that students produced were evaluated using an assessment scheme (see Section 4.4.1.3) and the minimum and maximum score of the practical test was 0 and 20, respectively. The reliability of the practical test's items was high, Cronbach's alpha = 0.77. The reliability of marking the programs of the practical test was high as well, r (15) = 0.79. For more details about the procedure used to assess the reliability of marking the practical test's programs, see

Section 4.4.1.3. The scores of the test violated the assumption of normality, but not that of homogeneity of variances. Hence, a significance level of 0.05 was used for the analysis. Descriptive statistics for the scores of the practical test are presented in Table 5.39.

Table 5.39
Descriptive Statistics for Practical-Test Scores

Group	n	Mean	SD	SE	Cohen's d
Koios	23	14.22	9.70	2.02	
Glossomatheia	25	8.38	7.28	1.46	0.69
MicroworldsPro	22	7.34	8.29	1.77	0.76
Total	70	9.97	8.86	1.06	

As can be seen from Table 5.39, the mean values of the Glossomatheia and the MicroworldsPro groups were approximately eight and seven, respectively. The mean score of the Koios group was approximately 14; almost twice the mean score of each of the other two groups. The SD for the Koios, the Glossomatheia and the MicroworldsPro groups was 9.70, 7.28 and 8.29, respectively. The effect size observed in the mean difference between the Koios and the Glossomatheia groups, as well as between the Koios and the MicroworldsPro groups, was, in both cases, medium to large, according to Cohen's (1988) recommendations. The analysis of practical test was based on the plan described in Section 4.4.2.3. The results of HMR of the practical-test scores are presented in Table 5.40.

In the first regression model, *Participant's GPA without CS* was the only predictor. It was a significant predictor and explained 26% of variance in the scores, R^2 =0 .26, F (1, 64) = 21.87, p < 0.001. In the second model, *Information* of the third measurement was added, which accounted for an extra 5% of variance, R^2 = 0.31, F (2, 63) = 13.98, p < 0.001.

Table 5.40
Results of Regression Analysis for the Practical-Test Scores (Second Round)

	Predictor	*B*	*SE*	*β*	*t*
Model 1					
	Constant	-23.07	7.18		**-3.21
	Participant's GPA without CS	1.98	0.42	0.50	***4.68
Model 2					
	Constant	-22.28	6.99		**-3.19
	Participant's GPA without CS	1.82	0.42	0.47	***4.37
	Information (Third Declarative-Knowledge Measurement)	1.13	0.51	0.23	*2.19
Model 3					
	Constant	-14.55	7.49		-1.94
	Participant's GPA without CS	1.10	0.50	0.28	*2.18
	Information (Third Declarative-Knowledge Measurement)	1.19	0.50	0.25	*2.39
	Scores of the Third Procedural-Knowledge Measurement	0.52	0.22	0.30	*2.38
Model 4					
	Constant	-11.78	6.31		-1.87
	Participant's GPA without CS	1.24	0.43	0.32	**2.92
	Information (Third Declarative-Knowledge Measurement)	1.19	0.41	0.25	**2.88
	Scores of the Third Procedural-Knowledge Measurement	0.59	0.19	0.34	**3.12
	Glossomatheia vs Koios	-8.35	1.86	-0.45	***-4.49
	MicroworldsPro vs Koios	-8.98	1.87	-0.47	***-4.81

Note. R^2 = 0.26 for Model 1; ΔR^2 = 0.05 for Model 2 (p < 0.05); ΔR^2 = 0.06 for Model 3 (p < 0.05); ΔR^2 = 0.21 for Model 4 (p < 0.001). * p < 0.05, ** p <0 .01, *** p < 0.001

Both *Participant's GPA without CS* and *Information* were significant predictors, *t* (63) = 4.37, *p* < 0.001 and *t* (63) = 2.19, *p* < 0.05, respectively. In Model 3, the scores of the third procedural-knowledge test were included. The additional variance explained was 6%, R^2 = 0.37, *F* (3, 62) = 11.89, *p* < 0.001. In this model, *Participants GPA without CS*, *Information* and the procedural-knowledge scores of the final measurement were all significant predictors, *t* (62) = 2.18, *p* < 0.05; *t* (62) = 2.39, *p* < 0.05; *t* (62) = 2.38, *p* < 0.05, respectively. Finally, the dummy variables of programming tool were introduced in the regression model. They accounted for an additional 21% of variance in the scores, $\Delta R^2 = 0.21$, F_{change} (2, 60) = 14.62, $p < 0.001$ and a total 57% of variance was explained by all predictors, $R^2 = 0.57$, F (5,60) = 16.12, $p < 0.001$. Programming tool explained a significant proportion of variance and therefore it was a significant predictor. This was also the case for *Participants GPA without CS*, *Information* and the procedural-knowledge scores of the third measurement, t (60) = 2.92, $p < 0.01$, t (60) = 2.88, $p < 0.01$ and t (60) = 3.12, $p < 0.01$, respectively. Finally, after controlling for these predictors, it was revealed that Koios users performed significantly better than Glossomatheia users, t (60) = 4.49, $p < 0.001$ and significantly better than MicroworldsPro users, t (60) = 4.81, $p < 0.001$. To further investigate the aspects in which Koios users achieved higher scores than users of MicroworldsPro and Glossomatheia, an in-depth analysis of the practical-test scores was performed. Figure 5.13 presents each group's scores with respect to aspects of creating and executing a program.

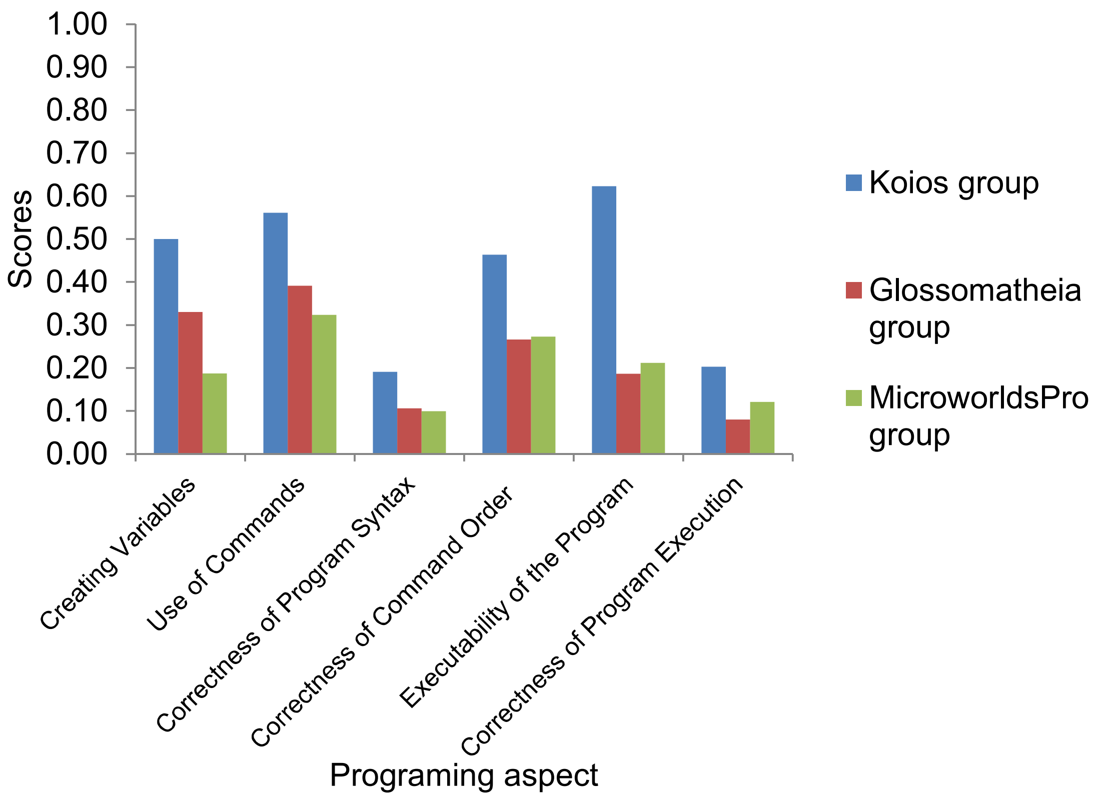


Figure 5.13. Mean scores for each programming aspect per group.

The aspects considered were: creating variables, overall use of commands (input-statement, output-statement, assignment-statement, iteration-statement, conditional statement and creating and calling a procedure), correctness of program syntax, correctness of command order, executability of the program and correctness of program execution. The maximum mark for each aspect was one. The grades achieved by the students of each group were averaged over the three programs for each aspect. Hence, the scores for each aspect ranged from zero (minimum) to one (maximum). From Figure 5.13 it can be seen that students who used Koios to create the programs of the practical test, seemed to have performed better than the users of the other two programming tools on all aspects. The values of mean, standard deviation and Cohen's *d* (1988) of the three groups for the six programming aspects are presented in Table 5.41.

Table 5.41
Descriptive Statistics for the Six Programming Aspects

	Group							
	Koios		Glossomatheia			MicroworldsPro		
Programming Aspect	*Mean*	*SD*	*Mean*	*SD*	Cohen's *d*	*Mean*	*SD*	Cohen's *d*
Creating Variables	0.50	0.35	0.33	0.35	0.49	0.19	0.30	0.96
Use of Commands	0.56	0.57	0.39	0.48	0.32	0.32	0.46	0.46
Correctness of Program Syntax	0.19	0.18	0.11	0.14	0.53	0.10	0.15	0.56
Correctness of Command Order	0.46	0.50	0.27	0.33	0.47	0.27	0.39	0.43
Executability of the Program	0.62	0.48	0.19	0.30	1.09	0.21	0.38	0.95
Correctness of Program Execution	0.20	0.39	0.08	0.15	0.43	0.12	0.32	0.23

More specifically, the mean for the Koios, the Glossomatheia and the MicroworldsPro groups for Creating Variables, Use of Commands, Correctness of Program Syntax, Correctness of Command Order, Executablity of the Program and Correctness of Program Execution were 0.50, 0.33 and 0.19; 0.56, 0.39 and 0.32; 0.19, 0.11 and 0.10; 0.46, 0.27 and 0.27; 0.62, 019 and 0.21; 0.20, 0.08 and 0.12, respectively. Likewise, the SD for the three groups and the six programming aspects were 0.35, 0.35 and 0.30; 0.57, 0.48 and 0.46; 0.18, 0.14 and 0.15; 0.50, 0.33 and 0.39; 0.48, 0.30 and 0.38; and 0.39, 0.15 and 0.32, respectively. The sizes of groups are reported in Table 5.39. As can be seen from Table 5.41 and according to Cohen's recommendations (1988), the effect sizes observed for the difference of the mean scores between the Koios and the Glossomatheia groups for (a) creating variables, (b) overall use of commands, (c) correctness of program syntax, (d) correctness of command order, (e) executability of the program and (f) correctness of program execution were (a) medium, (b) relatively low to medium, (c) medium, (d) medium, (e) large and (f) medium, respectively. The effect sizes represented by the differences of the mean scores for the six programming aspects between the Koios and the MicroworldsPro groups were (a) large, (b) medium, (c) medium, (d) medium, (e) large and (f) small, respectively.

In order to identify which differences in mental-model quality seemed to be responsible for the practical-test performance, the following analysis was conducted. From the produced mental models of the final declarative-knowledge measurement, the five ones with the lowest similarity and the five ones with the highest similarity with the baseline mental model were selected. One set of concept links was determined by identifying the common links between the five mental models with the lowest similarity and the baseline mental model. Similarly, the common links between the top-five mental models and the baseline mental model formed a second set of concept links. Comparing these two set of concept links the differences in the quality between the mental models with highest and lowest similarity were identified. In particular, the links that were different between the two sets were connecting the following concepts:

- integer and input command
- integer and iteration
- variable and input parameter
- variable and assignment
- variable and input command, and
- variable and logical condition

The links between these concepts seem to have differentiate the performance in the practical test. More specifically, students, whose mental models included these links, seemed to perform better in the practical test than students, whose mental models did not include them.

5.3.4 Check of Multiple Regression Assumptions

In the second round of data collection, three regression models were used to analyse procedural-knowledge data, ten regression models for the data analysis of the second and third declarative-knowledge measurement and one regression model to analyse the practical-test scores. The bias of each of these regression models was tested with the use of the following checks. For each model, the existence of outliers or influential cases, the assumptions of homoscedasticity and linear relationship between outcome and predictors and the independence of errors were

tested. In order to perform these checks, residual statistics, histogram, p-p plot of the standardised residual, plot of the standardised residual against standardised predicted values and the Durbin-Watson statistic test were used (Field, 2009).
Three (Case 22, Case 39 and Case 53), two (Case 5 and Case 74) and two (Case 19 and Case 56) extreme cases, with standardised residual less than minus two or greater than two, were detected from the residual statistics in the case of the first, second and third measurement, respectively. The residual statistics revealed two (Case 9 and Case 35), one (Case 9), two (Case 9 and Case 35), two (Case 9 and Case 12) and two (Case 12 and Case 35) extreme cases for *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance and Information* of the second declarative-knowledge measurement, respectively. In the case of the third declarative-knowledge measurement, the extreme cases observed for the five latter variables were three (Case 3, Case 10 and Case 11), four (Case 3, Case 10, Case 41 and Case 60), four (Case 3, Case 10, Case 41 and Case 60), three (Case 10, Case 41 and Case 60) and three (Case 3, Case 41 and Case 60), respectively. Finally, two extreme cases (Case 6 and Case 15) were detected in the practical-test scores. For each one of these cases, Cook's distance, three and two times the average leverage, Mahalanobis distance, covariance ratio and DFBeta statistics were computed (Field, 2009). The examinations of these statistics revealed that none of these cases were influential or outlying.
For each regression analysis of practical-test score, procedural- and declarative-knowledge data, histograms and the p-p plots of the standardised residual against standardised predicted values were plotted. Their investigation confirmed that

assumptions of homoscedasticity and linear relationship between outcome and predictors were met for all the cases.

The Durbin-Watson statistic was calculated for the three procedural-knowledge data analyses. Its value was 1.69, 1.87 and 1.83 for the first, second and third analysis, respectively. The value of Durbin-Watson statistic for *Links in Common*, *Difference between Observed and Expected Number of Links in Common*, *Similarity, Difference between Observed and Expected Similarity by Chance* and *Information* of the second declarative-knowledge measurement were 1.56, 1.53, 1.47, 1.51 and 1.40, respectively. For the five latter variables measured in the final measurement were 2.22, 2.17, 2.20, 2.16 and 2.14, respectively. Finally the Durbin-Watson statistic for the regression analysis of the practical-test scores was 1.90. The specific values indicated that the residuals in each analysis were independent. In addition, VIF and tolerance values revealed that no multicollinearity existed in any of the analysed data sets.

Finally, because HMR analysis was used to perform ANCOVA (see Section 4.4.2), the validity of the assumption of homogeneity of regression slopes should also be checked. Therefore, for every HMR analysis performed that included at least one covariate, this assumption was checked by conducting an additional ANCOVA that included the main effects of the independent variable and the covariate(s) as well as the interaction term between them (Field, 2009). More specifically, in the analysis of the practical-test scores, in which three covariates – *Participants GPA without CS*, *Information* and the final procedural-knowledge scores – were used, the terms that were included in the model of the ANCOVA were (a) the main effect of the independent variable and each of the covariates, (b) the interaction term between the independent variable and each of the covariates, (c) the interaction terms between *Participants GPA without CS* and *Information* as well as between *Participants GPA without CS* and the final procedural-knowledge scores, (d) the interaction terms between (i) the independent variable, *Participants GPA without CS* and *Information*, (ii) the independent variable, *Participants GPA without CS* and the final procedural-knowledge scores, and (iii) the independent variable, *Participants GPA without CS*, *Information* and the final procedural-knowledge scores. In all cases the variance explained by the interaction terms was not significant and thus, the assumption of homogeneity of regression slopes was met for all HMR analyses.

5.3.5 Summary of the Results of the Second Round of Data Collection

Declarative knowledge was initially measured in the beginning of the second round of data collection and before any experimental manipulation. The analysis of the data showed that programming tool was a significant predictor of three of the five declarative-knowledge variables. Planned contrasts revealed that students allocated to the Koios and the MicroworldsPro groups came from the same population, but students allocated to Glossomatheia group did not.

The first procedural-knowledge measurement was taken in the beginning of the fourth week of lessons after the second round had started. The results showed that both programming tool and *Participant's GPA without CS* were significant predictors. Moreover, the mean scores of the MicroworldsPro and the Glossomatheia groups were significantly higher than the mean score of the Koios group and therefore hypothesis H1 was rejected for this measurement.

The second procedural- and declarative-knowledge measurements took place consecutively and after five weeks of lessons. The results for the procedural-knowledge data revealed that *Participant's GPA without CS* was the only significant predictor of this' measurements scores. No significant differences were found between the mean score of the Koios and each of the other two groups. Hence, hypothesis H1 was not accepted and the first step of mediation analysis was not valid. The analysis outcome of the declarative-knowledge data showed that no significant differences between the mean scores of the three groups were detected, with one exception: the users of Koios scored significantly better in *Information* than the users of MicroworldsPro. Results also revealed that there were no significant predictors and students' declarative knowledge was not affected by programming tool. Therefore, hypothesis H2 was rejected and the second step of mediation was not successful.

The final procedural- and declarative-knowledge measurements took place at the end of the second round of data collection, namely after nine lessons since the beginning of the study. The results of procedural knowledge showed that, in the final test, *Participant's GPA without CS* was the sole predictor of the scores. Again no significant differences were observed between the mean performance of the Koios group and each of the other two groups. Thus, hypothesis H1 was rejected and the first step of mediation was not successful for this measurement. The analysis of the declarative-knowledge data revealed that no significant differences were observed between the means of the three groups and that programming tool had no effect on participants' declarative knowledge. Therefore, hypothesis H2 was rejected and the second step of mediation was not successful. The covariate *Homework Frequency until the third procedural-knowledge measurement* was a significant predictor of *Similarity, Difference between Expected and Observed Similarity by Chance* and *Information* of the third measurement.

Finally, the results of analysing the scores in the practical test revealed that *Participants GPA without CS*, *Information* and the procedural-knowledge scores of the third measurement were significant predictors of the practical-test scores. Furthermore, students of the Koios group achieved significantly higher scores in the practical test than students of both the Glossomatheia and the MicroworldsPro groups.

5.4 Conclusions

In this chapter, the results of the two rounds of data collection were reported. First the results for the procedural-knowledge data of the first round were presented, followed by the outcome of the declarative-knowledge data. The results of the second round were reported following the same presentation order and followed by results of the practical test. Using the outcome for procedural- and declarative-knowledge data, the validity of hypothesis H1, hypothesis H2 and mediation hypothesis H3 for each measurement was examined. Tables 5.42 and 5.43 present the results of both rounds synoptically. In the next chapter, the discussion of these findings follows.

Table 5.42
Summary of Procedural-Knowledge Results for Both Rounds of Data Collection

	Validity of		Significant Predictors
	Hypothesis H1	First Step of Mediation Hypothesis	
First Round of Data Collection			
First Procedural-Knowledge Scores	x	√	Programming tool Participant's GPA without CS
Second Procedural-Knowledge Scores	x	√	Programming tool Participant's GPA without CS
Third Procedural-Knowledge Scores	x	√	Programming tool Participant's GPA without CS
Second Round of Data Collection			
First Procedural-Knowledge Scores	x	√	Programming tool Participant's GPA without CS
Second Procedural-Knowledge Scores	x	x	Participant's GPA without CS
Third Procedural-Knowledge Scores	x	x	Participant's GPA without CS
Practical-Test Scores	Koios group performed significantly better than Glossomatheia and MicroworldsPro groups		Participant's GPA without CS Programming tool Information of Third Declarative-Knowledge Measurement Procedural-Knowledge Scores of the Third Measurement

Table 5.43
Summary of Declarative-Knowledge Results for Both Rounds of Data Collection

Variable	Validity of Hypothesis H2	Validity of Second Step of Mediation Hypothesis	Significant Predictors
First Round of Data Collection			
First Declarative-Knowledge Measurement			
Links in Common	x	x	Participant's GPA of previous year
Difference between Observed and Expected Links in Common	x	x	Participant's GPA of previous year
Similarity	x	x	Participant's GPA of previous year
Difference between Observed and Expected Similarity by Chance	x	x	Participant's GPA of previous year
Information	x	x	Participant's GPA of previous year
Second Declarative-Knowledge Measurement			
Links in Common	x	x	
Difference between Observed and Expected Links in Common	x	x	Day of Week
Similarity	x	x	
difference between Observed and Expected Similarity by Chance	x	x	Gaps Between Lessons
Information	x	x	Day of Week
Third Declarative-Knowledge Measurement			
Links in Common	x	x	Day of Week
Difference between Observed and Expected Links in Common	x	x	
Similarity	x	x	
Difference between Observed and Expected Similarity by Chance	x	x	
Information	x	x	
Second Round of Data Collection			
First Declarative-Knowledge Measurement			
Links in Common	x	x	
Difference between Observed and Expected Links in Common	x	√	
Similarity	x	x	

Table 5.43 (continued)

Variable	Validity of Hypothesis H2	Validity of Second Step of Mediation Hypothesis	Significant Predictors
Difference between Observed and Expected Similarity by Chance	x	√	
Information	x	√	
Second Declarative-Knowledge Measurement			
Links in Common	x	x	
Difference between Observed and Expected Links in Common	x	x	
Similarity	x	x	
Difference between Observed and Expected Similarity by Chance	x	x	
Information	x	x	
Third Declarative-Knowledge Measurement			
Links in Common	x	x	
Difference between Observed and Expected Links in Common	x	x	
Similarity	x	x	Homework Frequency until the Third Procedural-Knowledge Measurement
Difference between Observed and Expected Similarity by Chance	x	x	Homework Frequency until the Third Procedural-Knowledge Measurement
Information	x	x	Homework Frequency until the Third Procedural-Knowledge Measurement

Chapter Six

Discussion, Conclusions and Future Work

Contents

6.1 Recap of Thesis and Chapter Overview

This is the sixth and final chapter of this thesis. Chapter One introduced the reader to the research project reported in this thesis. The aim of this project is to improve the teaching and learning of programming for novices from a pedagogical and psychological point of view. Therefore, a set of seven principles for designing educational programming software was identified and discussed in the second chapter. Chapter Two also reviewed the relevant literature. Furthermore, a new programming tool, Koios, was designed and developed as an embodiment of the combined set of design principles. The third chapter explained the design and development process of Koios. An empirical study was designed and conducted in order to determine the importance of compliance with this set of principles. This was accomplished through the evaluation of Koios in Greek secondary educational institutes. The method, data collection and data analysis used in this study were described in Chapter Four. The results of analysing the collected data were reported in the fifth chapter of this thesis. This chapter discusses the reported results, outlines the conclusions that can be drawn from them, identifies this study's contribution to knowledge and proposes possible directions for future work.

6.2 Discussion

This section discusses the results reported in Chapter Five. First, a summary of the main findings is outlined, followed by their interpretations and explanations. Potential sources of bias are reported next and finally the limitations of this research are presented.

6.2.1 Summary of the Main Findings

The results of analysing declarative- and procedural-knowledge data of both rounds of data collection (see Table 5.42 and Table 5.43) revealed that, at the end of the study, novice programmers who used Koios – a programming tool with a high compliance with the set of the seven design principles – seem to have developed neither a higher level of programming skills nor richer mental models in the domain of introductory programming than novice programmers who used Glossomatheia and MicroworldsPro – programming software with partial compliance with this set. In other words, no significant effect of the compliance with the set of design principles on students' development of procedural and declarative programming knowledge was observed. Furthermore, examination of the results suggested that students' mental models (declarative programming knowledge) could not have mediated any effect of the programming environment on students' programming performance (procedural programming knowledge). Therefore, overall, not enough evidence was found to support the three research hypotheses under investigation by this research project (see Section 4.2.1) and thus the null hypothesis for each of the three research hypotheses could not be rejected. Nevertheless, according to the results, the variable *Participant's GPA without CS* significantly predicted the procedural-knowledge scores in all measurements of both rounds of data collection.
Moreover, the results of analysing the practical-test scores indicated that Koios users performed significantly better in the specific test than users of Glossomatheia and MicroworldsPro. *Participant's GPA without CS*, the scores on the third procedural-knowledge test and the scores on *Information*[2] in the third declarative-knowledge measurement were significant predictors of students' performance in the practical

[2] *Information* was a declarative-knowledge variable and associated with the probability of two PFNETs having at least a certain number of links in common by chance.

test. Moreover, an in-depth analysis of the practical-test scores revealed that Koios users appeared to perform better than users of Glossomatheia and MicroworldsPro in the following six programming aspects: creation of variables, overall use of commands, correctness of syntax, correctness of command order, executability of the program and correctness of program execution. It is important to note that the effect sizes of the differences in the mean scores of the practical test between the Koios and the Glossomatheia groups as well as between the Koios and the MicroworldsPro groups were large according to Cohen's (1988) conventions for the effect-size measure *d*.

Finally, a minor observation is that Glossomatheia users seem to have developed a significantly higher level of procedural knowledge than both Koios and MicroworldsPro users within the first three weeks of the study. However, this effect decayed in the long run and after the fifth lesson no strong evidence to support its existence was found.

6.2.2 Explanation and Interpretation of the Main Findings

The scores of both types of programming knowledge were overall low for all groups in both rounds of data collection according to the scales used to measure them (see Section 5.2.1.1 and Section 5.2.2.1). This finding indicated that participants found the declarative- and procedural-knowledge tests rather difficult, which, in turn, resulted, in my interpretation, in observing a floor effect (Pedhazur & Pedhazur Schmelkin, 1991). Moreover, regardless of the overall mean levels of acquired knowledge, no significant differences between the three groups in the mean level of procedural knowledge (dependent variable) and declarative knowledge (mediator) were observed. Hence, a significant effect of *compliance with the set of seven* *principles* (independent variable) on the dependent variable and mediator could not be detected in both rounds. Consequently, a mediation effect could not be detected as well. Possible reasons for these results follow. First, the possibility that the independent variable *compliance with the set of seven principles* could not have a significant effect on dependent variable and mediator is examined. Second, other factors that could have prevented the independent variable from having a significant effect are explored.

6.2.2.1 Examination of the effect of the independent variable.

The explanation for the lack of mediation that should be considered first is the presumed effectiveness of a programming tool with a high level of compliance with the set of principles on students' development of programming knowledge. This means that, perhaps, the use of this type of programming tool may not significantly affect the programming performance of novices. Or, in other words, compliance with the identified set of the seven design principles may not play an important role in affecting the development of procedural and declarative programming knowledge in an introductory programming course.

However, a significant effect of Koios on students' performance in the practical test was observed. This fact demonstrates that even if a programming tool that embodies the set of principles may not significantly influence the development of procedural and declarative programming knowledge, it can be more helpful for novices and facilitate the application of their procedural programming knowledge during program creation. The results of the practical test seem to support the idea that programming tool, declarative and procedural knowledge are independent components because no significant correlations between them were observed (see

Section 4.4.2.3). Nevertheless, the notion that this type of programming tool together with a certain level of procedural and declarative programming knowledge are all required for a successful programming performance is also supported by the results. More specifically, the results of the practical test revealed that procedural- and declarative-knowledge scores as well as programming tool – a programming tool with high compliance with the set of principles (Koios) versus tools without high compliance (MicroworldsPro and Glossomatheia) – were significant predictors of the practical-test scores.

The seven design principles were identified using not only novices' difficulties that are reported in research literature, but also a framework that was defined by combining the theories of Constructivism and Cognitive Load Theory. The pedagogical justification for the selection of the seven design principles based on this theoretical framework was briefly discussed in Section 2.5.8. More particularly, by *adopting the linguistic attributes of users' native language for the PL and IDE,* unnecessary mappings between foreign and natural language are avoided and thus, extraneous cognitive load is reduced. This is also achieved by *easily comprehensible error messages*, because such messages can help users understand and locate errors without having to refer to other sources and to split their attention. An appropriate *level of abstraction of programming commands*, which hides irrelevant information about them, while presenting only their necessary attributes could also reduce extraneous cognitive load as well as properly manage intrinsic cognitive load. This seems to be the case for a *small set of instructions*, which introduces the appropriate element interactivity for effective learning without increasing extraneous cognitive load through the introduction of unnecessary programming commands. *More natural syntax* and *understandable semantics* appear to facilitate the understating of programming concepts and produce fewer misconceptions and, thus, the formation of better schemata about them. This seems to be supported by *Visualisation* and *a high level of interaction with the IDE* as well. The latter two principles seem to actively engage users in assimilating and accommodating new knowledge at their own 'learning pace' by visually conveying implicit information, facilitating the understanding of programming concepts and providing a more user-friendly and interactive way of program creation and execution, during which guidance and on-time information are always available. This latter feature is very important from a constructivist viewpoint because it can support exploratory learning. Moreover, these seven principles were also identified as potential ways of addressing issues such as syntax and errors, which are often-cited in the relevant research literature as problematic for novices. This further supports the selection of these principles from a pedagogical viewpoint. However, particular interpretations regarding the effectiveness of each design principle separately cannot be valid because these principles were treated and tested as a combined set, not as individual principles.

Based on these characteristics, the combined set of seven principles seems to provide very important pedagogical guidelines for the design of PLs and IDEs. This is because theoretically together they can (a) reduce extraneous cognitive load, (b) provide a proper management of intrinsic cognitive load, (c) allocate more working-memory resources to the processing of new information, (d) facilitate the understanding of programming concepts and reduce possible misconceptions, (e) actively engage users in program creation and execution, (f) provide an easier and more comprehensible way of program creation and (g) present a more clear view and support a better understanding of program execution. Therefore, the combined set of the seven principles appears to be able to define a promising learning environment for novice programmers.

As discussed in Section 2.5.8, additional principles could be included in this set, however, as their number increases, so does the complexity of and the trade-off between their application. Nevertheless, a number of other principles were also taken into account during the design and implementation of Koios, for example *closeness of mapping* or *viscosity* (Pane & Myers, 1996). Closeness of mapping suggests that mental distance between the task and the programming domain should be as close as possible. However, closeness of mapping is a principle that can be more effectively applied in task-specific PLs, for example PLs for electronics (Green & Petre, 1996) than non-task-specific PLs. Viscosity describes the amount of effort that is required for a small change in programs. The application of viscosity, although important, is rather not expected to yield a large educational effect. Therefore, the selection process of the principles was aiming in including principles, the application of which could be as wide as possible and have a potential large pedagogical effect in the context of designing educational programming software. Thus, within the time limitations of this research project, the principles that were identified as such, were the ones that were included in the set.

Moreover, the theoretical framework, which contributed to identifying the seven design principles, can also be used to interpret the findings of this research. First, taking this framework into account, the finding that compliance with the principles had no significant effect on the development of declarative and procedural programming knowledge seems surprising. This is because Koios was designed with a high level of compliance with the set of design principles. The theoretical framework was used to identify a set of design principles in such manner that it would be possible for a programming tool, which embodies this set, to have an important pedagogical impact, in terms of learning to program, on its users. Therefore, a programming tool with a high level of compliance was supposed to assist its users in acquiring a substantial level of programming knowledge. However, this framework also highlights the influence of other factors, such as time or teaching method and material, in the learning process. If these factors are not properly considered or provided in a learning environment, then, instead of promoting the learning process, they may hinder it. Whether these factors could have affected the outcome of this study is examined in Section 6.2.2.2. Second, the observed effect of Koios on students' performance during the practical test seems to be the result of its high compliance with the seven design principles. Because Koios was designed as a manifestation of the combined set, it was expected to facilitate its users in acquiring a sufficient level of declarative and procedural programming knowledge. This effect was not detected; however, Koios seems to have facilitated its users in the application of their procedural knowledge during program creation. The particular effect, although not anticipated, positively influenced students' performance with respect to procedural knowledge. Thus, this outcome could be considered as a pedagogical improvement generated by the compliance with set of principles, which, to a certain extent, derived from the theoretical framework defined in this study.

Therefore, although a high compliance with the set of design principles was not found to play the anticipated pedagogical role, that is to support students in acquiring declarative and procedural programming knowledge, the idea that a high compliance with this set could play this role under different conditions could, perhaps, be further investigated. This is because there is evidence to suggest that high compliance with this set seems to play a different pedagogical role regarding procedural programming knowledge and more specifically, that of facilitating the application of procedural programming knowledge.

6.2.2.2 Alternative explanations regarding the effect of the independent variable.

Examining the case that compliance with this set could indeed significantly affect the development of participants' programming knowledge, the following factors could be identified as responsible for preventing this from taking place during this study: (a) the assessment of programming tools' compliance with the set of principles, (b) the duration of lessons as well as that of each of the two studies as a whole, (c) the number of programming concepts in the CS curriculum, (d) the teaching method and material used in the two studies and (e) the potential imprecision of measurement. A discussion of these factors follows.

First, perhaps the compliance of the three programming tools used in this research project with the set of design principles was assessed inaccurately. This could result in assessing programming tools with similar compliance with the set of design principles as if they had different levels of compliance. In turn, mapping these tools with different levels of the independent variable, according to the basic research design (see section 4.2.2.1), would not yield any significant changes in the dependent variable between the performances of the quasi-experimental groups. In order to avoid this research pitfall, an assessment protocol was produced (see Appendix C.1). This protocol was used to assess the compliance of each programming tool with the set of design principles as objectively as possible. The use of this protocol resulted in a comparison between the programming tools regarding their levels of compliance with the design principles (see Table 4.1), which indicated that there was a difference in their compliance with set of design principles. More specifically, according to the assessment, Koios highly complied with the principles, while Glossomatheia and MicroworldsPro only partially complied. However, the development and/or application of the protocol could be subjected to bias, because I was the person who produced the protocol, developed Koios and conducted the research. This and other sources of bias that could threat the internal validity of this study are further discussed in Section 6.2.3. In addition, alternative methods for assessing compliance between this set and programming tools could also be proposed. A first method could involve the assessment of compliance between the set of principles and programming tools by CS teachers. Different teachers could assess the level of compliance between the seven principles and a specific programming tool using a protocol (for instance, the one produced for this study, see Appendix C.1) and predefined levels – for example, low, medium and high. The assessments of teachers could then be averaged per programming tool and an overall assessment per programming tool could be produced. In a second method, one particular teacher could use two or more different programming tools in order to decide whether the tools differ in their compliance with a specific principle. In this case, further assessments regarding the level of compliance between the specific principle and each tool could be provided by the teacher.

The second factor that could have prevented compliance with the set of design principles from having a significant effect on participants' programming knowledge was time. From a theoretical point of view as well as based on findings from empirical programming studies, time is an important factor for learning to program. Both constructivism and CLT as well as mental-model theory argue that different individuals develop their knowledge at a different pace, the influence of declarative on procedural knowledge is a repetitive process that requires time to take place and that modification of mental models (declarative and procedural knowledge) is a slow process (see Sections 2.2.1, 2.2.2, 2.7.1 and 2.7.2.1). Additionally, according to Winslow's (1996) suggestion, ten years are approximately required for a novice programmer to become an expert. This suggestion indicates the importance of time and exposure to programming concepts in the advancement of programming performance and experience. The importance of duration and exposure in an introductory programming course has been highlighted by the work of Palumbo (1990) as well. Furthermore, McGill and Volet (1997) reported that even after a 12-week introductory course participants had acquired little programming knowledge. Taking these findings into consideration, a course of nine lessons, each approximately 40 minutes long, during which participants were exposed to programming concepts and practices reported in this thesis, was rather very short. Students' involvement in programming lessons, exercises and homework during the study is estimated to have been an average of 10 hours, which seems relatively inadequate for a successful course of learning programming.

The number and complexity of programming concepts that were supposed to be taught during the study is the third factor that could have hindered compliance with the set of design principles from significantly affecting participants' performance. The overall difficulty of programming concepts for novices as well as the misconceptions regarding these concepts that can arise were extensively presented in Section 2.4 and particularly in Section 2.4.2. This fact, in combination with the small amount of time that participants had to elaborate on and work with each new programming concept, could have rendered the development of a high level of relevant declarative knowledge an even more difficult task. This idea could provide a plausible explanation for the overall low levels of participants' programming knowledge observed during both rounds of data collection.

Despite the possible obstacles to knowledge development that the number of programming concepts and the duration of lessons and quasi-experiments might have posed in the empirical study, these two factors could not be controlled in order for the quasi-experiments and evaluation of Koios to take place in a 'real' learning environment of a secondary-education institute. However, these two factors are basic components of the quasi-experimental design and the limitations that this design may have introduced to this research are described in Section 6.2.4.

The fourth factor that could have influenced the effect of compliance with the design principles on the dependent variable is the teaching method and material adopted in this study. The teaching method was presented in Section 4.2.2.3 and part of the teaching material can be found translated in English in Appendix C. Apparently, this is a factor with serious implications in any educational setting and its importance is pointed out by CLT (see Section 2.2.2). Inappropriate teaching material and/or method could result in ineffective knowledge development, for example by increasing extraneous cognitive load or mismanaging intrinsic cognitive load, and thus, hampering a possible significant effect of programming tool. Therefore, an attempt to follow the guidelines of CLT was made not only for the design and development of Koios, but also for the teaching method and material used in this study. Although following these guidelines does not guarantee successful learning, it provides a framework within which suitable material for effective learning can be produced.

The fifth possible cause for why the expected effect did not occur is the accuracy of the knowledge measurements. From my experience, the rating task used to measure declarative programming knowledge could have been quite demanding. This was particularly pointed out by the fact that only two of the ten PFNETs submitted by teachers were usable to form the baseline PFNET (see Section 4.4.1.2). This was due to low values of coherence presented by the ratings of eight teachers. According to the PCKNOT manual (Sitze, 1992), low coherence values in ratings is indicative of participants who could not or did not take the rating task seriously. Thus, if teachers who have experience with programming concepts and their application produced ratings with low coherence values, then this could mean that some features of the rating task may be relatively difficult. Additionally, if more PFNETs could be included in the construction of the baseline PFNET, then it could, perhaps, be more accurate and representative of teachers' declarative knowledge. This finding suggests that the low level of declarative knowledge that students may develop during an introductory programming course may not be sufficient for an effective performance in the rating task. Thus, an additional and, perhaps, simpler measurement of declarative knowledge could be introduced in the data collection scheme. This measurement could include items that would require students to use their declarative knowledge to address them. Taking into consideration that declarative knowledge is supposed to be easy to communicate verbally, these items could be simple questions regarding the declarative knowledge of programming concepts. For instance, "why in the development of programs do we use an output statement?" (McGill & Volet, 1997). However, the scores on *Information* – a variable produced by PCKNOT – in the final measurement of the second round was a significant predictor of the practical test scores. This indicates that using pair ratings and PCKNOT could produce meaningful measurements of declarative knowledge. Therefore, a delicate combination of the two methods could be used for a more precise declarative-knowledge measurement. For example, declarative-knowledge questions could be used for initial measurements or when a low level of declarative knowledge is expected, while ratings of paired concepts could replace them or be additionally employed when more complex declarative knowledge is expected to have been developed. Moreover, correlations between the scores of the two declarative-knowledge measurements with one or more dependent variables, such as procedural-knowledge scores, could be used to determine which one of the two declarative-knowledge measurements seems to be more sensitive per occasion. Nevertheless, it seems that there were not similar problems regarding the accuracy of procedural-knowledge measurements. Although the items of the three procedural-knowledge tests of the first round were created from scratch, they turned out to be overall good in discriminating students' procedural knowledge among the three quasi-experimental groups. This was determined by cross-tabulation analysis on the scores of these items (see Appendix C.11). The results revealed that students' responses to the majority of tests' items presented, in my interpretation, neither a ceiling nor a floor effect, in other words, tests' items were neither too easy nor too difficult. Nevertheless, items that were not very sensitive were refined for the second round of data collection, thus making the items of procedural-knowledge tests of the second round, presumably, more sensitive in discriminating procedural knowledge between groups. Possible bias regarding the assessment of students' responses to these items is discussed in Section 6.2.3.

Although, the items used for assessing students programming skills seem to have provided good measurements, qualitative techniques for measuring programming skills could be included in the data collection scheme as well, for example interviews with students and thinkaloud protocols. During these interviews students could be given a number of procedural-knowledge items, like the ones used in the procedural-knowledge tests, and asked to explain their response to particular items. Additionally, students could be presented with a piece of code and asked to create and explain the control flow of this piece of code or walk the interviewer through the execution of the code. This way, students' understanding with respect to procedural knowledge of programming concepts could be tested. Moreover, declarative knowledge could also be measured by employing this kind of technique. In particular, the procedural-knowledge items of the interviews could be replaced by declarative-knowledge items.

Finally, the measurements of declarative and procedural programming knowledge did not involve the use of any programming tool. This was corrected by the addition of a practical test in the data-collection scheme (see Sections 4.3.2.1 and 4.3.2.2.3).

6.2.2.3 Explanation and interpretation of remaining findings.

One interesting but not surprising finding was that the variable *Participant's GPA without CS* was a significant predictor of procedural-knowledge and practical-test scores. A number of variables have been reported as significant predictors of programming performance, for example students' mark in maths (Bennedssen & Caspersen, 2008; Pillay & Jugoo, 2005). Participants' math score was a significant predictor of programming performance in this study as well. However, participants' math score presented a high correlation with *Participant's GPA without CS* and, of the two variables, the latter presented a higher correlation with procedural-knowledge scores (see Section 4.4.1.4). Therefore, *Participant's GPA without CS* was used in the place of participant's math score. This finding is not surprising when one considers that *Participant's GPA without CS* was selected to represent participants' overall learning performance at school. Thus, a person with a high overall learning performance is, presumably, expected to present this level of performance in individual subjects as well, such as learning to program. Furthermore, this study corroborated the finding from previous studies that gender does not affect programming performance (Kahler, 2002; Lau & Yuen, 2010, 2011; Linn, 1985; Pillay & Jugoo, 2005).

Overall, a low level of procedural and declarative programming knowledge was acquired by participants during the study. This is not a surprising outcome for a – rather short – first programming course, as similar findings were reported by Garner et al. (2005), Kurland, Pea, Clement and Mawby (1989), Linn and Dalbey (1989), Winslow (1996) and McGill and Volet (1997). More specifically and with respect to the practical-test results, the correlation between the scores of the practical test and the third procedural-knowledge measurement appeared to be higher than the correlation between the scores of the practical test and *Information* on the third knowledge measurement (see Section 4.4.2.3). This might support the idea that participants relied more on their procedural knowledge than on their declarative knowledge to deal with the practical test. This was expected because the application of procedural programming knowledge was supposed to be more important than the use of declarative programming knowledge in order to efficiently respond to the items of the practical test. Moreover, this finding could also be explained by the postulation of McGill and Volet (1997) that a certain level of procedural knowledge can be acquired and applied by simple observation and before one develops a high level of declarative knowledge. The same assumption was discussed and supported by Schnotz and Kürschner (2007) as well. Additionally, based on this postulation and the observed levels of programming knowledge acquired by participants, it could be assumed that overall participants acquired a higher level of procedural than declarative programming knowledge. This assumption could also be supported by the hypothesis of Cañas et al. (1994) that mental models, after their initial formation, do not change substantially before a significant improvement in procedural knowledge and that mental models seem not to affect performance, at least at the early stages of learning. Nonetheless, the hypothesis of Cañas et al (1994) together with the supposedly slow modification of mental models (Ma, 2007) could also explain the observed low levels of declarative knowledge acquired by students. The overall low level of programming knowledge acquired by students could also indicate that they did not develop more complex forms of knowledge than those of declarative and procedural programming knowledge. This indication could also support the choice of measuring only declarative and procedural programming knowledge. Although various types of knowledge have been acknowledged (de Jong & Ferguson-Hessler, 1996), the distinction of programming knowledge between declarative and procedural as the only relevant forms of knowledge seems justified, in the context of this research project, for the following two reasons. First, the majority of research in psychology of programming mostly uses only two types of knowledge: declarative and procedural, whilst only one additional type is identified: strategic or conditional knowledge (Anderson, 1982; Anderson, Conrad & Corbett, 1989; McGill & Volet, 1997; Palumbo, 1990). This is also suggested by the relevant literature review reported in Section 2.7. Moreover, strategic or conditional knowledge describes knowing when and where to apply declarative and procedural knowledge and is supposed to be hierarchically higher and more complex than them. The acquisition of a certain level of declarative and procedural knowledge is assumed before the development of strategic or conditional knowledge. However, this level was not observed in this study. Second, McGill and Volet (1997) argued that “a simple distinction between declarative and procedural knowledge is sufficient for educational computing purposes and there is no need for a complex model.”(McGill & Volet, 1997, p. 292).

6.2.3 Sources of Potential Bias in This Study

Bias can be a threat to the validity of a study and distort its outcomes and conclusions (Pedhazur & Pedhazur Schmelkin, 1991). Therefore, it is important to identify possible sources of bias in research because the acknowledgement and investigation of these sources could potentially minimise bias in future research. Naturally, this research project was, to a certain degree, potentially subject to bias. The main source of bias for this study could be the fact that I was the person who designed and developed Koios, prepared the teaching material, gave the lessons and collected procedural and declarative-knowledge data. The evident conflict of interests and potential bias could genuinely affect the validity of this study. Therefore, three possible causes of as well as the measures taken to reduce bias are analysed.

First, as discussed in Section 6.2.2.2, the assessment of compliance of the programming tools involved in this study with the set of design principles could be biased because I designed Koios and also measured compliance. Moreover, the allocation of the programming tools to the levels of the independent variable was based on this assessment. In order to obtain an objective assessment of compliance a protocol was developed and used. For more details about this protocol, see Appendix C.1.

A second feature that could introduce bias to the quasi-experiments that were conducted was the teaching material and the teaching method used. Again, it is not difficult to imagine the potential threats for objectiveness that could arise from having developed Koios and giving all the lessons to three quasi-experimental groups. An example would be elaborating more on programming concepts with the Koios group than with the Glossomatheia and MicroworldsPro groups. This is used only as an example and, to my best knowledge, it did not take place. This type of bias was perhaps more difficult to detect and control than the first type. The measure taken to minimise this was to produce generic teaching material in terms of theory, examples and homework for each lesson. This material was later accordingly modified for each group to describe the use of each programming tool within the context of each lesson. Thus, the only necessary differences in the teaching material between the three groups were the details of using each programming tool. The teaching process during class was heavily based on this material in order to have a uniform teaching method across experimental groups.

Finally, because I marked the practical and procedural-knowledge tests and also developed Koios, I could possibly be more tolerant to errors made by students of the Koios group than with errors made by students of the MicroworldsPro and the Glossomatheia groups. Making the marking process more objective involved the production of two marking schemes, one for procedural-knowledge tests (see Section 4.4.1.1) and one for practical test (see Appendix C.16), respectively. Moreover, an external marker was recruited to mark a sample of students' responses. External marker's assessment was used to calculate the reliability of marking procedural-knowledge tests as well as the practical test. High reliability values were observed for marking these tests (see Sections 5.2.1, 5.3.1 and 5.3.3), which indicated an accurate initial marking.

6.2.4 Limitations of the Research

The outcomes of a research project are affected not only by potential bias, but also by its inherent limitations. These limitations should be acknowledged as well. A discussion of the identified limitations of the research reported in this thesis follow. The main limitation of this research design was that in order to increase the possibility of this empirical study taking place in 'real' learning environments, such as secondary or tertiary educational institutes, the research design had to be based on an existing curriculum and programme of studies. The details of and the reasons for this choice were discussed in Section 4.2.2. Although this choice facilitated the execution of the empirical study, it also caused a number of restrictions. More specifically, I could not have control over the following features: (a) the selection and the number of programming concepts to be taught, (b) the length of the study, (c) the duration of lessons, (d) any 'gaps' between consecutive lessons and (e) class size. The first two features were constant for all quasi-experimental groups and could not be accounted for. However, the latter three varied across groups and thus, their effect could be controlled for by including these features as covariates in data analysis. In addition, gender was identified as another potential confounding variable and included in data analysis. This, by all means, does not suggest that all potential confounding variables were identified. Presumably, there could be a number of variables that could influence the outcome of this study, but the identification of all these variables is not possible. Consequently, these variables were not included in data analysis and thus, not accounted for.

A second limitation posed by conducting the empirical study at educational institutes was that an experimental design could not be used for the data collection. Instead, a quasi-experimental design was used because a random allocation of participants to experimental groups was not possible. Random allocation is used to eliminate any systematic initial differences between experimental groups. Therefore, the existence of this type of difference between groups could not be ruled out nor its absence claimed. However, initial differences between groups were detected and particularly during the first declarative-knowledge measurement of the second round of data collection (see Sections 4.4.2.2.2 and 5.3.2.2). This difference could not even be explained by students' performance in the school year before the year that data collection took place, because *GPA from previous year* was not a significant predictor of the outcome variables of this round's first declarative-knowledge measurement (see Section 4.3.2.2.1). Because this measurement took place before any experimental manipulation, this difference was treated as an initial difference between the experimental groups and was used as a covariate in the analysis of the data collected in the following measurements. Notably, results did not reveal any significant differences to be existing before the experimental manipulation between groups in both rounds.

The third limitation was with respect to the age of participants. Because data collection in tertiary institutes was not feasible for reasons explained in Sections 4.3.1.1 and 4.3.2.1, all participants were students in secondary schools and particularly of the third grade of high-school. The reasons for this were that (a) participants of this study were required to have no previous school training in programming and (b) the introductory programming course for students in the Greek educational system is included in the CS curriculum of the third grade of high-school.

Thus, whether the effect of compliance with the seven design principles on pupils' programming knowledge is influenced by their age could not be investigated, so effectively the variable *age* was eliminated by making it effectively a constant (Pedhazur & Pedhazur Schmelkin, 1991). As conducting data collection in tertiary institutes was not feasible, not only was access prohibited to participants of different ages but this also posed the following limitations.

The fourth limitation was that a programming tool with high compliance with the set of principles was not evaluated as a programming tool for tertiary level. The evaluation of Koios in tertiary institutes could provide more evidence to examine its effect on the acquisition of programming knowledge by tertiary students. More specifically, the effect of compliance with the seven principles on developing declarative and procedural knowledge of programming concepts for tertiary-level introductory courses (see Section 4.2.2.4) could be studied. Hypotheses H2 and H1 proposed in this project suggest that users of Koios would develop richer mental models and higher level of procedural knowledge, respectively, than users of C during such courses. Moreover, at least one practical test with tertiary students could be scheduled. Based on the findings of the practical test during the quasi-experiment in secondary education, Koios users would be expected to perform better in creating programs on their own than C users.

The fifth limitation had to do with comparing educational programming tools with professional, non-educational programming languages, such as C, widely used as programming tools for introductory programming courses in tertiary education. This comparison was not possible and thus, a conclusion with respect to determining which of the two types of programming tool is more suitable for tertiary education could not be reached. The findings of the quasi-experiment with secondary-education students suggest that Koios would better facilitate its users in the application of their procedural programming knowledge than C during program creation. This is because Koios was superior to Glossomatheia and MicroworldsPro in the practical test, while C had an even poorer compliance with the seven principles than Glossomatheia and MicroworldsPro. In addition, the effectiveness of a programming tool, which was developed as a manifestation of the combined set of the seven principles, as a ‘stepping stone’ to a professional programming language could not be investigated.

Finally, due to low sample size a valid exploratory factor analysis of procedural-knowledge tests was not possible (Field, 2009). Exploratory factor analysis could be used to reveal the factor structure of potential underlying components of procedural programming knowledge measured by items of the relevant tests. Thus, a relation between these potential components could not be identified and used for producing tests that, perhaps, could elicit procedural-knowledge more accurately and be used to measure the effect of programming tool on individual components.

6.3 Conclusions

This section presents the conclusions that can be drawn from the results reported in Chapter Five and after considering the issues discussed in Section 6.2. The main conclusion, based on the results, is that none of the three research hypotheses were supported. This means that novice programmers who use a programming tool with a high level of compliance with the set of the seven design principles do not develop significantly different levels of procedural (programming skills) and declarative (mental models) programming knowledge from novice programmers who use ‘standard’ educational programming software during an introductory programming course. Moreover, it could not be established that the quality of students' mental models is a mediator of the effect of compliance on programming skills. These results suggest that programming tool, procedural and declarative knowledge are independent components in the process of learning to program.

However, their importance is highlighted by a second conclusion. From the results of the practical test, it was concluded that both types of knowledge, programming tool (tools varying in their compliance with the seven design principles) and overall academic performance in subjects other than CS are independent influential predictors of successful performance in an introductory programming course.

The third tentative conclusion can be drawn from the results of the practical test as well. Users of a programming tool with a high compliance with the set of design principles appear to perform better than users of 'standard' software in the following six programming aspects: creation of variables, overall use of commands, correctness of syntax, correctness of command order, executability of the program and correctness of program execution.

A fourth conclusion is that a successful teaching of the 13 programming concepts – *integer, arithmetic expression, string, output-statement, input-statement, variable, assignment-statement, iteration-statement, procedure, procedure call, input parameter, logical condition* and *conditional statement* – in nine lessons of approximately 40 minutes each, is rather unrealistic, regardless of the programming tool used. This could, perhaps, be accomplished by reducing the number of programming concepts or by increasing the number of lessons and/or their duration. Of course, the combination of both suggestions could be even more effective.

The fifth and last conclusion is that students' GPA is a significant independent predictor of students' procedural knowledge. Students' GPA is a significant predictor of students' performance in practical tests as well.

6.4 Summary of Contribution to Knowledge

This section outlines in which respects the research project presented in this thesis makes an original contribution to knowledge. First, the main contribution of this study is to identify and suggest a set of scientific principles for the design of introductory educational programming tools (see Section 2.5). Thus, this set can be used as a framework for designing and developing pedagogical programming tools or even educational software outside the domain of programming, if and where this is applicable. However, this set was not only identified as an important set of principles through literature review, but it was also empirically tested. Contributions to knowledge by the design and execution of the empirical study follow.

Designing and developing a new Greek programming tool, Koios, as an embodiment of this set of design principles (see Chapter Three) is a second contribution to knowledge. Koios was evaluated in Greek secondary educational institutes through quasi-experimental research. Furthermore, its educational impact was compared with the impact of software that had already been used, at the time that this study took place, for teaching and learning programming in these institutes (see Chapter Four). Through this evaluation and comparison, the suggestion of Koios as an effective programming tool for an introductory programming course seems to be well supported.

A third contribution to knowledge of this study is by proposing the method used to evaluate Koios and compare it with other software as a guideline for conducting similar research and/or comparing educational software programs. Particular important are the techniques that were developed to measure students' programming knowledge. Naturally, this is only a methodological suggestion for future studies and there is space for improvement and adaptation of the method according to requirements of each study. For example, a practical test could be administered after each procedural-knowledge measurement or a comparison between programming tools that support different programming paradigms could take place. A number of suggestions regarding the design and execution of future studies of this type follows in Section 6.5.2. Furthermore, the results of this type of study seem to provide a basis for informed choices in terms of software's educational effectiveness.

The empirical study of the effect of compliance with the set of design principles on the development of novices' programming knowledge through the evaluation of Koios is a fourth contribution to knowledge. The results of this study provided no evidence to establish a relationship between compliance with this set of design principles and the acquisition of programming knowledge by novices, in the context of an introductory course for learning to program. However, it was discovered that procedural and declarative programming knowledge are independent components, at least at the early stages of learning programming. This can be interesting from a theoretical viewpoint because it seems to offer a better insight in the relationship between procedural and declarative programming knowledge during the early stages of their development. If this, or better the absence of a strong, relationship between the two types of knowledge is taken into account, then new research ideas can be proposed (see Section 6.5.2) for determining the factors that influence the development of declarative and procedural programming knowledge. Furthermore, their interrelation and the level of their development, at which this relation begins to form, could be investigated. From an educational point of view, being aware of a possible disassociation between the two types of knowledge could be used for a more effective design of teaching methods and/or material regarding introductory programming courses.

A fifth contribution of this research is identifying that compliance with this set of principles can facilitate the *application* of procedural programming knowledge by novices during program creation. In this manner, not only the educational significance of the set of design principles is demonstrated, but also the area where this effect seems to be particularly strong is revealed. Therefore, following this set of principles when designing programming software for novices appears to be an important consideration.

Finally, this study contributes to knowledge by determining that procedural and declarative knowledge as well as a programming tool's compliance with design principles contribute to successful performance in an introductory programming course. Obviously, these factors are only but part of the features that are needed for an effective course. However, recognising even a part of the necessary 'ingredients' for a successful introductory programming course is a valuable knowledge. This knowledge can improve course design and make it more efficient, and improve students' performance. Overall, this knowledge seems to provide useful information for improving the teaching and learning of programming for novices, which is an important aim of the research project reported in this thesis.

6.5 Future Work

I have identified two directions that can be followed with respect to the development of this work: (a) future work regarding tool development, involving the Koios programming environment and (b) future work regarding research. First, suggestions for Koios are presented, followed by suggestions for further research.

6.5.1 Future Work Regarding Tool Development

A number of modifications that should be considered first include some choices that were rejected during the design of the Koios prototype. Thus, such modifications could include inserting comments during the creation of programs and making the states of variables, expressions and conditions visible in every step of programs' execution. The inclusion of comments is considered good programming practice that helps programmers read and understand code. The presentation of values of variables, expressions and conditions increases the visibility of the notional machine and, in turn, facilitates the understanding of programming concepts (du Boulay et al., 1999; Pane & Myers, 2000). However, whether the provision of these features by programming tools is necessary within the context of an introductory programming course was discussed in Sections 3.2.2.2 and 3.2.2.3. Therefore, the provision of these features by programming tools should be optional and teachers or individual users could decide, according to their requirements, whether these features should be enabled or disabled. Furthermore, additional modes could be provided with respect to the execution of programs. Hence, programs could be executed in a step-by-step mode, allow users to return to a previous state in execution and overall provide simple features of professional debuggers. Similarly, these modes could be used to increase the visibility of the notional machine and, more particularly, expose the intermediate states and steps during program execution.

Another major goal of modifications could be to further increase Koios' compliance with the seven design principles. For instance, increasing users' attention during programs' execution. This could engage users in a more focused trace of programs' execution and thus, enable users to better understand programs' execution. Increasing users' attention could be achieved by posing questions regarding programs' state to students during execution, like VILLE does (Rajala et al., 2008). Demonstrated techniques of CLT (see Section 2.2.2.2) could be a considerable source of inspiration in further increasing compliance. For example, based on the *modality effect*, the comments produced by Koios in Program Execution mode could be provided in an oral, instead of a visual, form (Shaffer et al., 2003; Stefik & Gellenbeck, 2011; van der Veer, 1989). Thus, a new functionality could be added to Koios that would use a text-to-speech system to read out loud these comments. Moreover, considering the *completion problem effect*, Koios could support a number of predefined programs that would be missing a number of commands according to users' level of expertise for them to complete.

Other suggestions may involve making visible the graphical view together with the textual view during execution and being able to create programs in a textual manner. The first suggestion could allow users to follow programs' execution in the graphical view, which is the one they use to create programs, while the second one could familiarise users with the 'traditional' way of creating programs. Nevertheless, the latter suggestion could be considered rather impractical given the verbose syntax of Koios' programs, which, however, increases the readability of programs and, presumably, renders them easier to understand.

6.5.2 Future Research

Suggestions for future work can be made with respect to the design and execution of the type of empirical studies as the one reported in this thesis. Addressing the limitations of the current study could be a good starting point.

More particularly, running an experiment based on the quasi-experimental design for tertiary education (see Section 4.2.2.4) would be the main suggestion. The results of such experiment could reveal potential effects of Koios on the development of tertiary students' programming knowledge. Moreover, the results of comparing Koios with a professional programming environment within the context of CS education could be quite informative about their suitability as programming tools. Additionally, experiments with tertiary as well as with secondary students would benefit from a random allocation of participants to experimental groups, which could cancel out any systematic initial differences. Running such experiments in 'real' learning environments is not, however, an easy task due to administrative reasons. Distance-learning courses could be employed to overcome problems posed by administrative reasons, but in this case, the absence of physical classes could, perhaps, introduce more serious problems. For example, the lack of physical presence of participants during lessons and knowledge measurements could result in collecting unreliable data or missing out important observations regarding the use of programming tools and overall performance. Based on the theoretical framework defined by constructivism and Cognitive Load Theory, it would also be useful to explore the effect of both the number of programming concepts and the time students are exposed to these concepts during similar experiments. This way, not only the effect of two important pedagogical factors on learning to program could be studied, but also an optimal exposure for specific sets of programming concepts could be determined. Suggestions regarding these parameters, based on the results of the experiment described in this thesis, would be to increase duration and number of lessons or decrease the number of programming concepts, or even both. In the case of a sufficient sample size (at the very least *N* > 100) a factor analysis could be performed in order to identify any existing patterns in the measurement structure of procedural knowledge tests.

The addition of the practical test in the second round of this study revealed a significant effect and with a large magnitude. Thus, further improvements to the research design could be made by including one or two additional practical tests. Perhaps, the introduction of qualitative measurements, for instance interviews with students or teachers, in the research design could help obtaining useful information for improving Koios. Moreover, a follow-up study could be conducted to investigate any possible long-term effects on students' knowledge development. However, running a follow-up study with high-school students of the third grade is not easy, because students' progression to the next grade involves moving to a different type of school. Thus, students from a particular school have a range of schools they can go to and they usually do not end up all in the same school. However, a follow-up study would presumably be more feasible with tertiary-education students who take an introductory programming course in their first year and who can be 'followed up' in subsequent years of study.

Based on the absence of a strong relationship between procedural and declarative programming knowledge, at least during the early stages of learning programming, research suggestions can be made in terms of knowledge measurement. Based on (a) the results of analysing declarative-knowledge data of both rounds and the ideas that (b) mental models, after their initial formation, do not substantially change before a significant improvement in procedural knowledge (Cañas et al., 1994), and (c) mental models are overall subject to slow change (Ma, 2007), two measurements of declarative knowledge – one in the beginning and the second in the end of the study – seem to be sufficient for a meaningful monitoring of declarative-knowledge development during a short introductory programming course. In such courses, the overall level of the acquired declarative knowledge is rather low and its modification is not very rapid – this is supported by the findings of this and other research projects (Cañas et al., 1994; Garner et al., 2005; Linn & Dalbey, 1989; McGill & Volet, 1997) as well as by mental-model theory (Ma, 2007) – and thus, a more frequent measurement of declarative knowledge does not offer useful information. In contrast, procedural knowledge appears to change faster than declarative knowledge in introductory courses and presumably its application can take place without extensive relying on declarative knowledge. Therefore, procedural-knowledge and practical tests could be employed as the main methods for assessing the level of acquired knowledge. The data obtained from these tests could provide more information regarding the development of procedural knowledge and its application. For example, a 'mini' procedural-knowledge and practical test could be administered every two or three lessons or after the completion of teaching each programming concept. After the teaching of two or three concepts, a more complex procedural-knowledge and practical test regarding these concepts could be administered. Alternatively, the final declarative-knowledge measurement of a set of programming concepts could take place when students have reached a significant level of procedural knowledge with respect to this set, regardless of the number of lessons required to reach it. However, the number of lessons required for reaching this level could indicate the perceived difficulty of this set.

From a theoretical point of view, exploring the following research question that has emerged from examining the results and drawing conclusions could reveal more information regarding the relationship between compliance with the set of principles and the development and application of procedural programming knowledge. Cognitive Load Theory and constructivism could provide an important theoretical background for tackling the following research question: *why does compliance with the set of the seven design principles facilitate the application of procedural knowledge, but not its development?*

This theoretical background could also be used to identify pedagogical requirements and needs that novice programmers may have during an introductory programming course and design it accordingly. Curriculum, teaching methods and material are aspects of this course that could be defined by the theoretical framework of constructivism and CLT. The findings of this research suggest that introductory courses may be more successful if they focus more on the development of procedural programming knowledge, without, however, neglecting the importance of declarative knowledge.

Finally, it could be worthwhile to explore the possibility of identifying a less programming-oriented and more generic version of this set of principles, where this is possible, for designing educational software that could be used by novices in scientific domains where the application of procedural knowledge is important. This new set could be based on the set identified by this research and modified using the following rationale. The principles that are too specific and have meaning only within a programming context, namely *syntax and semantics of instructions supported by the PL*, *level of abstraction provided by the PL* and *small set of instructions*, should not be included in the new, more generic set. However the remaining principles, namely *match of users' natural language with the linguistics attributes of the software*, *visualisation*, *provision of error messages* and *high level of interaction with the IDE* could be generalised and meaningful for designing educational software for different educational disciplines. In fact, at least some of these principles have already been recognised as principles of usable design (for example Nielsen, 1993). For example, providing a high level of interaction with the IDE is a principle that it can be incorporated in various educational programs, regardless of the subject that is learned. Obviously, this is not the exact set of principles identified by this study. Nevertheless, this 'core' of principles could be suggested as a good starting point for educational-software design, which could be further improved or refined according to the scientific domain. For instance, this core of principles could be used as a basis for designing educational software for math or physics.

6.6 Closing Remarks

From my viewpoint, the main aim set at the beginning of this study has to a large extent been accomplished. In my opinion, the identification of this set of principles, the effect that this seems to have on the application of procedural programming knowledge and the conclusions drawn from this study could be used to improve the teaching and learning of programming. Towards this aim, Koios, the programming tool that I developed as a manifestation of this set of design principles and was empirically evaluated, can be used as well. However, I believe that a programming tool cannot be a panacea for the difficulties of novice programmers during their first course, but rather contribute to students' learning. In ongoing attempts to improve teaching and learning of programming, other factors and their improvement should also be considered, for instance curricula, teaching methods and material, learning conditions and, perhaps, most important, the teachers. Thus, Koios is suggested as an enhanced programming tool that, within an overall enhanced learning environment, could be used to help novice programmers learn programming. Therefore, I conclude by paraphrasing and transferring Ludwig Wittgenstein's (1922) famous quote in the programming domain: do the limits of a first programming language mean the limits of novices' programming world? At the present time this does not seem to be the case.

References

Abdul-Rahman, S.-S. & du Boulay, B. (2014). Learning programming via worked-examples: Relation of learning styles to cognitive load. *Computers in Human Behavior, 30*(0), 286-298.

Ala-Mutka, K. (2004). Problems in learning and teaching programming *Codewitz Needs Analysis*: Institute of Software Systems, Tampere University of Technology,Finland.

Allan, V. H. & Kolesar, M. V. (1997). Teaching computer science: A problem solving approach that works. *SIGCUE Outlook, 25*(1-2), 2-10. doi: 10.1145/274375.274376

Allen, E., Cartwright, R. & Reis, C. (2003). Production programming in the classroom. *SIGCSE Bulletin, 35*(1), 89-93. doi: 10.1145/792548.611940

Allen, E., Cartwright, R. & Stoler, B. (2002). Drjava: A lightweight pedagogic environment for Java. *SIGCSE Bulletin, 34*(1), 137-141. doi: 10.1145/563517.563395

Allwood, C. M. (1986). Novices on the computer: A review of the literature. *International Journal of Man-Machine Studies, 25*(6), 633-658.

Amershi, S., Carenini, G., Conati, C., Mackworth, A. K. & Poole, D. (2008). Pedagogy and usability in interactive algorithm visualizations: Designing and evaluating cispace. *Interacting with Computers, 20*(1), 64-96. doi: http://dx.doi.org/10.1016/j.intcom.2007.08.003

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review, 89*, 369-406.

Anderson, J. R., Conrad, F. G. & Corbett, A. T. (1989). Skill acquisition and the lisp tutor. *Cognitive Science, 13*(4), 467-505.

Anderson, R. C. (1977). The notion of schemata and the educational enterprise: General discussion of the conference. In R. C. Anderson, R. J. Spiro & W. E. Montague (Eds.), *Schooling and the acquisition of knowledge* (pp. 415-431). Hillsdale, NJ: Erlbaum.

Areias, C. & Mendes, A. (2007). *A tool to help students to develop programming skills.* 2007 international conference on Computer systems and technologies, Bulgaria.

Ayres, P. & van Gog, T. (2009). State of the art research into cognitive load theory. *Computers in Human Behavior, 25*(2), 253-257.

Baddeley, A. (1992). Working memory. *Science, New Series, 255*(5044), 556-559.

Baddeley, A. (2000). The episodic buffer: A new component of working memory? *Trends in Cognitive Sciences, 4*(11), 417-423.

Banks, A. P. & Millward, L. J. (2007). Differentiating knowledge in teams: The effect of shared declarative and procedural knowledge on team performance. *Group Dynamics: Theory, Research, and Practice, 11*(2), 95-106.

Barnes, D. J., Fincher, S. & Thompson, S. (1997). *Introductory problem solving in computer science.* 5th Annual Conference on the Teaching of Computing, Centre for Teaching Computing, Dublin City University, Dublin 9, Ireland.

Baron, R. M. & Kenny, D. A. (1986). The moderator-mediator variable distinction in social psychological research: Conceptual, strategic and statistical considerations. *Journal of personality and social psychology, 51*(6), 1173-1182.

Barr, M., Holden, S., Phillips, D. & Greening, T. (1999). An exploration of novice programming errors in an object-oriented environment. *SIGCSE Bulletin, 31*(4), 42-46. doi: 10.1145/349522.349392

Bartlett, F. C. (1932). *Remembering: A study in experimental and social psychology*. Cambridge, England: Cambridge University Press.

Bayman, P. & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM, 26*(9), 677-679. doi: 10.1145/358172.358408

Bayman, P. & Mayer, R. E. (1984). Instructional manipulation of users' mental models for electronic calculators. *International Journal of Man-Machine Studies, 20*(2), 189-199.

Ben-Ari, M. (1998). Constructivism in computer science education. *ACM SIGCSE Bulletin 30*(1), 257-261. doi: http://doi.acm.org/10.1145/274790.274308

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching, 20*(1), 45-73.

Ben-Ari, M. (2002). *From theory to experiment to practice in CS education.* Koli Calling: Second Annual Finnish/Baltic Sea Conference on Computer ScienceEducation, Koli, Finland.

Ben-Ari, M., Bednarik, R., Ben-Bassat Levy, R., Ebel, G., Moreno, A., Myller, N. et al. (2011). A decade of research and development on program animation: The jeliot experience. *Journal of Visual Languages & Computing, 22*(5), 375-384.

Ben-Bassat Levy, R., Ben-Ari, M. & Uronen, P. A. (2003). The jeliot 2000 program animation system. *Computers & Education, 40*(1), 1-15. doi: http://dx.doi.org/10.1016/S0360-1315(02)00076-3

Bennedssen, J. & Caspersen, M. E. (2008). *Abstraction ability as an indicator of success for learning computing science?* 4th international Workshop on Computing Education Research, Sydney, Australia.

Bergin, J., Stehlik, M., Roberts, J. & Pattis, R. (2005). *A gentle introduction to the art of object-oriented programming in Java*. Retrieved from http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html

Berry, M. & Kölling, M. (2014). *The state of play: A notional machine for learning programming.* Proceedings of the 2014 conference on Innovation and Technology in Computer Science Education, Uppsala, Sweden.

Beynon, M. (2009). Constructivist computer science education reconstructed. *Innovation in Teaching And Learning in Information and Computer Sciences, 8*(2), 73-90.

Bishop-Clark, C., Courte, J., Evans, D. & Howard, E. V. (2007). A quantitative and qualitative investigation of using Alice programming to improve confidence, enjoyment and achievement among non-majors. *Journal of Educational Computing Research, 37*(2), 193-207.

Black, T. R. (2006). Helping novice programming students succeed. *Journal of Computing Sciences in Colleges, 22*(2), 109-114.

Blackwell, A. F. (2002a). *First steps in programming: A rationale for attention investment models.* IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02).

Blackwell, A. F. (2002b). *What is programming?* 14th Workshop of the Psychology of Programming Interest Group, Brunel University, London,UK.

Blackwell, A. F., Whitley, K. N., Good, J. & Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review, 15*(1-2), 95-114. doi: 10.1023/a:1006689708296

Bonar, J. & Soloway, E. (1983). *Uncovering principles of novice programming.* 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Austin, Texas.

Bonar, J. & Soloway, E. (1989). Preprogramming knowledge: A major source of misconceptions in novice programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 325-353). Hillsdale, NJ: Erlbaum.

Bruckman, A. & Edwards, E. (1999). *Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language.* SIGCHI conference on Human factors in computing systems: the CHI is the limit, Pittsburgh, Pennsylvania, United States.

Brusilovsky, P. (1994). *Teaching programming to novices : A review of approaches and tools.* ED-MEDIA 94 - World Conference on Educational Multimedia and Hypermedia, Vancouver, British Columbia, Canada.

Calloni, B. A. & Bagert, D. J. (1994). *Iconic programming in baccii vs. Textual programming: Which is a better learning environment?* 25th SIGCSE symposium on Computer science education, Phoenix, Arizona, United States.

Calloni, B. A. & Bagert, D. J. (1995). *Iconic programming for teaching the first year programming sequence.* Frontiers in Education Conference on 1995. Proceedings., 1995 vol 1. - Volume 01.

Calloni, B. A., Bagert, D. J. & Haiduk, H. P. (1997). *Iconic programming proves effective for teaching the first year programming sequence.* 28th SIGCSE technical symposium on Computer science education.

Cañas, J. J., Bajo, M. T. & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human-Computer Studies, 40*(5), 795-811.

Carley, K. & Palmquist, M. (1992). Extracting, representing, and analyzing mental models. *Social Forces, 70*(3), 601-636.

Carlisle, M. C., Wilson, T. A., Humphries, J. W. & Hadfield, S. M. (2005). Raptor: A visual programming environment for teaching algorithmic problem solving. *ACM SIGCSE Bulletin, 37*(1), 176-180.

Caspersen, M. E. & Bennedsen, J. (2007). *Instructional design of a programming course: A learning theoretic approach.* 3rd international workshop on Computing education research, Atlanta, Georgia, USA.

Chalmers, P. A. (2003). The role of cognitive theory in human-computer interface. *Computers in Human Behavior, 19*(5), 593-607.

Chandler, P. & Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition and Instruction, 8*(4), 293-332.

Chen, L.-H. (2011). Enhancement of student learning performance using personalized diagnosis and remedial learning system. *Computers & Education, 56*(1), 289-299.

Chipperfield, B. (2004). Cognitive load theory and instructional design, from http://www.usask.ca/education/coursework/802papers/chipperfield/

Chong, T. S. (2005). Recent advances in cognitive load theory research:Implications for instructional designers. *Malaysian Online Journal of Instructional Technology, 2*(3), 106-117.

Cilliers, C., Calitz, A. & Greyling, J. (2005). The effect of integrating an iconic programming notation into CS1. *ACM SIGCSE Bulletin, 37*(3), 108-112.

Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E. & Whalley, J. (2008). *The teaching of novice computer programmers: Bringing the scholarly-research* *approach to australia.* 10th conference on Australasian computing education, Wollongong, NSW, Australia.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). Hillsdale, New Jersey: Erlbaum.

Cooke, N. J. (1994). Varieties of knowledge elicitation techniques. *International Journal of Human-Computer Studies, 41*(6), 801-849.

Cooke, N. J. & Rowe, A. L. (1997). Measures of mental models: A synthesis of evaluative data. *Training Research Journal, 3*, 185-207.

Cooke, N. J. & Schvaneveldt, W. (1988). Effects of computer programming experience on network representations of abstract programming concepts. *International Journal of Man-Machine Studies 29*(4), 407-427. doi: http://dx.doi.org/10.1016/S0020-7373(88)80003-8

Corritore, C. L. & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies, 50*(1), 61-83. doi: 10.1006/ijhc.1998.0236

Corritore, C. L. & Wiedenbeck, S. (2001). An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies, 54*(1), 1-23.

Craik, K. (1943). *The nature of explanation*. Cambridge, UK: Cambridge University Press.

Dagdilelis, V., Evangelidis, G., Satratzemi, M., Efopoulos, V. & Zagouras, C. (2003). Delys: A novel microworld-based educational software for teaching computer science subjects. *Computers & Education, 40*(4), 307-325.

Dagdilelis, V., Satratzemi, M. & Evangelidis, G. (2004). Introducing secondary education students to algorithms and programming. *Education and Information Technologies, 9*(2), 159-173.

Darabi, A. A., Nelson, D. W. & Seel, N. M. (2009). Progression of mental models throughout the phases of a computer-based instructional simulation: Supportive information, practice, and performance. *Computers in Human Behavior, 25*(3), 723-730.

Davies, S. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies, 39*(2), 237-267.

de Jong, T. (2010). Cognitive load theory, educational research, and instructional design: Some food for thought. *Instructional Science, 38*(2), 105-134.

de Jong, T. & Ferguson-Hessler, M. G. M. (1996). Types and qualities of knowledge. *Educational Psychologist, 31*(2), 105-113. doi: 10.1207/s15326985ep3102_2

de Raadt, M. (2008). *Teaching programming strategies explicitly to novice programmers.* PhD thesis, University of Southern Queensland.

de Raadt, M., Watson, R. & Toleman, M. (2006). *Chick sexing and novice programmers: Explicit instruction of problem solving strategies.* 8th Australasian Conference on Computing Education, Hobart, Australia.

Deek, F. P. & Espinosa, I. (2005). An evolving approach to learning problem solving and program development: The distributed learning model. *International Journal on E-Learning, 4*(4), 409-426.

Deek, F. P. & McHugh, J. A. (1998). A survey and critical analysis of tools for learning programming. *Computer Science Education, 8*(2), 130-178.

Denny, P., Luxton-Reilly, A. & Carpenter, D. (2014). *Enhancing syntax error messages appears ineffectual.* Proceedings of the 2014 Innovation and

Technology in Computer Science Education Conference - ITICSE 2014, Uppsala, Sweden.

DePasquale, P., Lee, J. A. N. & Pérez-Quiñones, M. A. (2004). Evaluation of subsetting programming language elements in a novice's programming environment. *ACM SIGCSE Bulletin, 36*(1), 260-264. doi: http://doi.acm.org/10.1145/1028174.971392

Détienne, F. (1997). Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers, 9*(1), 47-72.

Dijkstra, E. W. (1989). On the cruelty of really teaching computing science. *Communications of the ACM, 32*(12), 1398-1404.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 283-299). Hillsdale, NJ: Erlbaum.

du Boulay, B. & Matthew, I. A. N. (1984). Fatal error in pass zero: How not to confuse novices. *Behaviour & Information Technology, 3*(2), 109-118. doi: 10.1080/01449298408901742

du Boulay, B., O'Shea, T. & Monk, J. (1999). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human-Computer Studies, 51*(2), 265-227.

Duke, R., Salzman, E., Burmeister, J., Poon, J. & Murray, L. (2000). *Teaching programming to beginners - choosing the language is just the first step.* Australasian conference on Computing education, Melbourne, Australia.

Ebel, G. & Ben-Ari, M. (2006). *Affective effects of program visualization.* 2nd international workshop on Computing education research, Canterbury, United Kingdom.

Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies, 41*(4), 457-480. doi: http://dx.doi.org/10.1006/ijhc.1994.1069

Edwards, J. (2005). *Subtext: Uncovering the simplicity of programming.* 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA, USA.

Efopoulos, V., Dagdilelis, V., Evangelidis, G. & Satratzemi, M. (2005). *Wipe: A programming environment for novices.* 10th annual SIGCSE conference on Innovation and technology in computer science education

Efopoulos, V., Evangelidis, G. & Dagdilelis, V. (2005, 5-8 July 2005). *Wipe - pilot testing and comparative evaluation.* 5th IEEE International Conference on Advanced Learning Technologies, 2005. ICALT 2005.

Evangelidis, G., Dagdilelis, V., Satratzemi, M. & Efopoulos, V. (2001). *X-compiler: Yet another integrated novice programming environment.* IEEE International Conference on Advanced Learning Technologies.

Faul, F., Erdfelder, E., Buchner, A. & Lang, A.-G. (2009). Statistical power analyses using g*power 3.1: Tests for correlation and regression analyses. *Behavior Research Methods, 41*, 1149-1160.

Faul, F., Erdfelder, E., Lang, A.-G. & Buchner, A. (2007). G*power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods, 39*, 175-191.

Field, A. (2009). *Discovering statistics using spss; and sex, drugs and rock'n'roll*. London: Sage.

Fitzgerald, S., Simon, B. & Thomas, L. (2005). *Strategies that students use to trace code: An analysis based in grounded theory.* 1st international workshop on Computing education research, Seattle, WA, USA.

Fix, V., Wiedenbeck, S. & Scholtz, J. (1993). *Mental representations of programs by novices and experts.* INTERACT '93 and CHI '93 conference on human factors in computing systems, Amsterdam, The Netherlands.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J. et al. (2007). Developing a computer science-specific learning taxonomy. *SIGCSE Bulletin, 39*(4), 152-170. doi: 10.1145/1345375.1345438

Garner, S. (2002a). *Colors for programming: A system to support the learning of programming.* Informing Science & IT Education Conference (InSITE), Cork, Ireland.

Garner, S. (2002b). *Reducing the cognitive load on novice programmers.* World Conference on Educational Multimedia, Hypermedia and Telecommunications 2002, Denver, Colorado, USA.

Garner, S. (2009). A quantitative study of a software tool that supports a part-complete solution method on learning outcomes. *Journal of Information Technology Education, 8*, 285-310.

Garner, S., Haden, P. & Robins, A. (2005). *My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems.* 7th Australasian conference on Computing education - Volume 42, Newcastle, New South Wales, Australia.

Gaspar, A. & Langevin, S. (2007). *Restoring "coding with intention" in introductory programming courses.* 8th ACM SIGITE conference on Information technology education, Destin, Florida, USA.

Gaspar, A., Langevin, S. & Boyer, N. (2008). Redundancy and syntax-late approaches in introductory programming courses. *Journal of Computing Sciences in Colleges, 24*(2), 204-212.

Gentner, D. & Stevens, A. (1983). *Mental models*. Hillsdale,NJ: Erlbaum.

Georgatos, F. (2002). *How applicable is Python as first computer language for teaching programming in a pre-university educational environment, from a teacher's point of view?* MSc thesis, Universiteit van Amsterdam, Amsterdam.

George, C. E. (2000). *Experiences with novices: The importance of graphical representations in supporting mental models* 12 th Annual Workshop of the Psychology of Programming Interest Group Cozenza Italy.

Gillan, D. J., Breedin, S. D. & Cooke, N. J. (1992). Network and multidimensional representations of the declarative knowledge of human-computer interface design experts. *International Journal of Man-Machine Studies, 36*(4), 587-615.

Gilmore, D. J. (1990). Expert programming knowledge: A strategic approach. In J.-M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 223-234). Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Gilmore, D. J. (1991). Models of debugging. *Acta Psychologica, 78*(1 - 3), 151-172. doi: http://dx.doi.org/10.1016/0001-6918(91)90009-O

Goldman, K. J. (2004a). *A concepts-first introduction to computer science.* 35th SIGCSE technical symposium on Computer science education, Norfolk, Virginia, USA.

Goldman, K. J. (2004b). An interactive environment for beginning Java programmers. *Science of Computer Programming, 53*(1), 3-24.

Goldsmith, T. E. & Johnson, P. J. (1990). A structural assessment of classroom learning *Pathfinder associative networks: Studies in knowledge organization* (pp. 241-254): Ablex Publishing Corp.

Goldsmith, T. E., Johnson, P. J. & Acton, W. H. (1991). Assessing structural knowledge. *Journal of Educational Psychology, 83*(1), 88-96.

Goldweber, M., Bergin, J., Lister, R. & McNally, M. (2006). *A comparison of different approaches to the introductory programming course.* 8th Australasian Conference on Computing Education - Volume 52, Hobart, Australia.

Gomes, A. & Mendes, A. J. (2007). *Learning to program - difficulties and solutions.* International Conference on Engineering Education – ICEE 2007, Coimbra, Portugal.

Gonzalez, G. (2004). Constructivism in an introduction to programming course. *Journal of Computing Sciences in Colleges, 19*(4), 299-305.

Good, J. (2011). Learners at the wheel: Novice programming environments come of age. 1-24. doi: 10.4018/ijpop.2011010101

Gosling, J. & McGilton, H. (1996). The Java language environment.

Grandell, L., Peltomäki, M., Back, R.-J., Salakoski, T. & (2006). *Why complicate things?: Introducing programming in high school using Python.* 8th Austalian conference on Computing education - Volume 52, Hobart, Australia.

Grandell, L., Peltomäki, M. & Salakoski, T. (2005). *High-school programming - a beyond-syntax analysis of novice programmers' difficulties.* 5th Koli Calling International Conference on Computer Science Education.

Gray, S., Clair, C. S., James, R. & Mead, J. (2007). *Suggestions for graduated exposure to programming concepts using fading worked examples.* 3rd international workshop on Computing education research, Atlanta, Georgia, USA.

Greca, I. M. & Moreira, M. A. (2000). Mental models, conceptual models, and modelling. *International Journal of Science Education, 22*(1), 1-11. doi: 10.1080/095006900289976

Green, T. R. G. (1990a). The nature of programming. In J.-M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 23-44). Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Green, T. R. G. (1990b). Programming languages as information structures. In J. M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gilmore (Eds.), *Psychology of Programming*. Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Green, T. R. G. & Petre, M. (1996). Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework. *Journal of Visual Languages & Computing, 7*(2), 131-174. doi: http://dx.doi.org/10.1006/jvlc.1996.0009

Greyling, J. H., Cilliers, C. B. & Calitz, A. P. (2006, 10-13 July 2006). *B#: The development and assessment of an iconic programming tool for novice programmers.* 7th International Conference on Information Technology Based Higher Education and Training, 2006. ITHET '06.

Gross, P. & Powers, K. (2005a). *Evaluating assessments of novice programming environments.* 1st international workshop on Computing education research, Seattle, WA, USA.

Gross, P. & Powers, K. (2005b). *Work in progress - a meta-study of software tools for introductory programming.* 35th ASEE/IEEE Frontiers in Education, Indianapolis, IN.

Guibert, N., Girard, P. & Guittet, L. (2004). *Example-based programming: A pertinent visual approach for learning to program.* Working conference on Advanced visual interfaces, Gallipoli, Italy.

Gupta, D. (2004). What is a good first programming language? *Crossroads, 10*(4), 7-7. doi: http://doi.acm.org/10.1145/1027313.1027320

Hadjerrouit, S. (2005). Constructivism as guiding philosophy for software engineering education. *SIGCSE Bulletin, 37*(4), 45-49. doi: 10.1145/1113847.1113875

Harvey, L. & Anderson, J. R. (1996). Transfer of declarative knowledge in complex information-processing domains. *Human-Computer Interaction, 11*(1), 69-96.

Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *Computer Science Education, 13*(2), 95-22.

Helminen, J. & Malmi, L. (2010). *Jype - a program visualization and programming exercise tool for Python.* 5th international symposium on Software visualization, Salt Lake City, Utah, USA.

Hendrix, T. D., James H. Cross, II & Larry, A. B. (2004). An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. *SIGCSE Bulletin, 36*(1), 387-391. doi: 10.1145/1028174.971433

Henriksen, P. & Kölling, M. (2004). *Greenfoot: Combining object visualisation with interaction.* 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Vancouver, BC, CANADA.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM 12*(10), 576-580. doi: http://doi.acm.org/10.1145/363235.363259

Hoc, J.-M., Green, T. R. G., Samurçay, R. & Gilmore, D. J. (1990). Psychology of programming Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Hollender, N., Hofmann, C., Deneke, M. & Schmitz, B. (2010). Integrating cognitive load theory and concepts of human-computer interaction. *Computers in Human Behavior, 26*(6), 1278-1288.

Hristova, M., Misra, A., Rutter, M. & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bulletin, 35*(1), 153-156. doi: 10.1145/792548.611956

Hsia, J. I., Simpson, E., Smith, D. & Cartwright, R. (2005). Taming Java for the classroom. *SIGCSE Bulletin, 37*(1), 327-331. doi: 10.1145/1047124.1047459

Hu, M. (2004). *Teaching novices programming with core language and dynamic visualisation.* 17th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ), Christchurch, New Zeland.

Hundhausen, C. D. (2002). Integrating algorithm visualization technology into an undergraduate algorithms course: Ethnographic studies of a social constructivist approach. *Computers & Education, 39*(3), 237-260. doi: 10.1016/s0360-1315(02)00044-1

Hundhausen, C. D. & Brown, J. L. (2007). What you see is what you code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing, 18*(1), 22-47. doi: http://dx.doi.org/10.1016/j.jvlc.2006.03.002

Hundhausen, C. D. & Brown, J. L. (2008). Designing, visualizing, and discussing algorithms within a CS 1 studio experience: An empirical study. *Computers & Education, 50*(1), 301-326.

Hundhausen, C. D., Douglas, S. A. & Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing, 13*(3), 259-290.

Hundhausen, C. D., Farley, S. F. & Brown, J. L. (2009). Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Transactions on Computer-Human Interaction, 16*(3), 1-40. doi: 10.1145/1592440.1592442

Ihantola, P., Karavirta, V., Korhonen, A. & Nikander, J. (2005). *Taxonomy of effortless creation of algorithm visualizations.* 1st international workshop on Computing education research, Seattle, WA, USA.

Jenkins, T. (2002). *On the difficulty of learning to program.* 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences.

Jih, H. J. & Reeves, T. C. (1992). Mental models: A research focus for interactive learning systems. *Educational Technology Research and Development, 40*(3), 39-53. doi: 10.1007/bf02296841

Johnson-Laird, P. N. (1983). *Mental models: Towards a cognitive science of language, inference and consciousness*. Cambridge, UK Cambridge University Press.

Johnson-Laird, P. N. (1989). Mental models. In M. I. Posner (Ed.), *The foundations of cognitive science*. Cambridge, MA, USA: The MIT Press.

Kaasbøll, J. J. (1999). *Exploring didactic models for programming.* Norwegian informatics Tapir, Trondheim

Kaczmarczyk, L. C., Petrick, E. R., East, J. P. & Herman, G. L. (2010). *Identifying student misconceptions of programming.* 41st ACM technical symposium on Computer science education, Milwaukee, Wisconsin, USA.

Kahler, S. E. (2002). *A comparison of knowledge acquisition methods for the elicitation of procedural mental models.* PhD thesis, North Carolina State University, Raleigh.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 209-228). Hillsdale, NJ: Erlbaum.

Kalyuga, S. (2011). Cognitive load theory: How many types of load does it really need? *Educational Psychology Review, 23*(1), 1-19.

Kasurinen, J., Purmonen, M. & Nikula, U. (2008). *A study of visualization in introductory programming.* 20th Annual Workshop of PPIG, Lancaster, United Kingdom.

Kelleher, C. & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys, 37*(2), 83-137.

Kellog, W. A. & Breen, T. J. (1990). Using pathfinder to evaluate user and system models. In R. W. Schvaneveldt (Ed.), *Pathfinder associative networks: Studies in knowledge organization.* (pp. 179-196). Norwood, NJ: Ablex.

Keppel, G. (1991). *Design and analysis: A researcher's handbook*. Upper Saddle River, NJ: Prentice Hall.

Keppel, G., Saufley, W. & Tokunaga, H. (1992). *Introduction to design and analysis: A student's handbook* (2nd ed.). USA: Freeman.

Kernighan, B. & Ritchie, D. (1988). *The C programming language (2nd edition)*: Prentice Hall.

Kinnunen, P. & Malmi, L. (2008). *CS minors in a CS1 course.* 4th international Workshop on Computing Education Research, Sydney, Australia.

Kiper, J. D., Howard, E. & Ames, C. (1997). Criteria for evaluation of visual programming languages. *Journal of Visual Languages & Computing, 8*(2), 175-192. doi: http://dx.doi.org/10.1006/jvlc.1996.0034

Knöll, R. & Mezini , M. (2006). *Pegasus: First steps toward a naturalistic programming language.* 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA.

Ko, A. J. & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*(16), 41-84.

Kodama, E., Sato, K. & Miyazaki, M. (2000). *Natural language programming for multimedia information.* 7th International Conference on Parallel and Distributed Systems: Workshops.

Kolikant, Y. B.-D. & Mussai, M. (2008). So my program doesn't run!" Definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education, 18*(2), 135-151.

Kölling, M. (2010). The Greenfoot programming environment. *ACM Transactions on Computing Education, 10*(4), 1-21. doi: 10.1145/1868358.1868361

Kölling, M., Quig, B., Patterson, A. & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education, 13*(4), 249-286.

Koppelman, H. (2008). *Pedagogical content knowledge and educational cases in computer science: An exploration.* Informing Science & IT Education Conference (InSITE), Varna, Bulgaria.

Kordaki, M. (2009). *Beginners' programming attempts to accomplish a multiple-solution based task within a multiple representational computer environment.* World Conference on Educational Multimedia, Hypermedia and Telecommunications 2009, Honolulu, HI, USA.

Kordaki, M. (2010). A drawing and multi-representational computer environment for beginners' learning of programming using C: Design and pilot formative evaluation. *Computers & Education, 54*(1), 69-87.

Kranch, D. (2012). Teaching the novice programmer: A study of instructional sequences and perception. *Education and Information Technologies, 17*(3), 291-313. doi: 10.1007/s10639-011-9158-8

Kurland, D. M. & Pea, R. D. (1985). Children's mental models of recursive logo programs. *Journal of Educational Computing Research, 1*(2), 235-244.

Kurland, D. M. & Pea, R. D. (1989). Children's mental models of recursive logo programs. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 325-325). Hillsdale, NJ: Erlbaum.

Kurland, D. M., Pea, R. D., Clement, C. & Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer*. Hillsdale, NJ: Erlbaum.

Lahtinen, E. (2008). *Students' individual differences in using visualizations: Prospects of future research on program visualizations.* 8th International Conference on Computing Education Research, Koli, Finland.

Lahtinen, E., Ala-Mutka, K. & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin, 37*(3), 14-18.

Larkin, J. H. & Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science, 11*(1), 65-100.

Lau, W. W. F. (2009). *Exploring the relationships among gender, learning style, mental model, and programming performance: Implications for learning and teaching of computer programming.* PhD thesis, The University of Hong Kong, Hong Kong.

Lau, W. W. F. & Yuen, A. H. K. (2010). Promoting conceptual change of learning sorting algorithm through the diagnosis of mental models: The effects of gender and learning styles. *Computers & Education, 54*(1), 275-288.

Lau, W. W. F. & Yuen, A. H. K. (2011). Modelling programming performance: Beyond the influence of learner characteristics. *Computers & Education, 57*(1), 1202-1213.

LCSI. (2008). Lcsi - soloutions - microworlds pro, from http://www.microworlds.com/solutions/mwpro.html

Leahy, W. & Sweller, J. (2011). Cognitive load theory, modality of presentation and the transient information effect. *Applied Cognitive Psychology, 25*(6), 943-951. doi: 10.1002/acp.1787

Lee, M. J. & Ko, A. J. (2011). *Personifying programming tool feedback improves novice programmers' learning.* 7th international workshop on Computing education research, Providence, Rhode Island, USA.

Lee, Y.-J. (2011). Empowering teachers to create educational software: A constructivist approach utilizing etoys, pair programming and cognitive apprenticeship. *Computers & Education, 56*(2), 527-538.

Lieber, T., Brandt, J. R. & Miller, R. C. (2014). *Addressing misconceptions about code with always-on programming visualizations.* Proceedings of the 32nd annual ACM conference on Human factors in computing systems, Toronto, Ontario, Canada.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14*(5), 14-29.

Linn, M. C. & Dalbey, J. (1989). Cognitive consequences of programming instruction. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 57-81). Hillsdale, NJ: Erlbaum.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M. et al. (2004). *A multi-national study of reading and tracing skills in novice programmers.* ITiCSE on Innovation and technology in computer science education, Leeds, United Kingdom.

Lister, R., Simon, B., Thompson, E., Whalley, J. L. & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *SIGCSE Bulletin, 38*(3), 118-122. doi: 10.1145/1140123.1140157

Lorenzen, T. & Anderson, V. (1993). *Design of experiments: A no-name approach*. New York: Marcel Dekker.

Lui, A. K., Kwan, R., Poon, M. & Cheung, Y. H. Y. (2004). Saving weak programming students: Applying constructivism in a first programming course. *ACM SIGCSE Bulletin, 36*(2), 72-76. doi: 10.1145/1024338.1024376

Ma, L. (2007). *Investigating and improving novice programmers' mental models of programming concepts.* PhD thesis, University of Strathclyde.

Ma, L., Ferguson, J., Roper, M., Ross, I. & Wood, M. (2008). *Using cognitive conflict and visualisation to improve mental models held by novice programmers.* 39th

SIGCSE technical symposium on Computer science education, Portland, OR, USA.

Ma, L., Ferguson, J., Roper, M., Ross, I. & Wood, M. (2009). *Improving the mental models held by novice programmers using cognitive conflict and jeliot visualisations.* 14th annual ACM SIGCSE conference on Innovation and technology in computer science education, Paris, France.

MacKinnon, D. P. (2008). *Introduction to statistical mediation analysis*. New York: Erlbaum.

Malan, D. J. & Leitner, H. H. (2007). Scratch for budding computer scientists. *ACM SIGCSE Bulletin 39*(1), 223-227. doi: http://doi.acm.org/10.1145/1227504.1227388

Mannila, L. (2007). Novices' progress in introductory programming courses. *Informatics in education, 6*(1), 139-152.

Mannila, L. & de Raadt, M. (2006). *An objective comparison of languages for teaching introductory programming.* 6th Baltic Sea conference on Computing education research: Koli Calling 2006, Uppsala, Sweden.

Marceau, G., Fisler, K. & Krishnamurthi, S. (2011). *Mind your language: On novices' interactions with error messages.* 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software, Portland, Oregon, USA.

Marcotty, M. & Ledgard, H. (1986). *The world of programming languages*. Berlin: Springer-Verlag.

Marhan, A. M., Micle, M. I., Popa, C. & Preda, G. (2012). A review of mental models research in child-computer interaction. *Procedia - Social and Behavioral Sciences, 33*(0), 368-372.

Mayer, R. E. (1989). The psychology of how novices learn computer programming. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 129-159). Hillsdale, NJ: Erlbaum.

Mayer, R. E. & Moreno, R. (2002). Aids to computer-based multimedia learning. *Learning and Instruction, 12*(1), 107-119.

Mayer, R. E. & Moreno, R. (2003). Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist, 28*(1), 43-52.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D. et al. (2001). *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students.* ITiCSE on Innovation and technology in computer science education, Canterbury, UK.

McGill, T. J. & Volet, S. E. (1997). A conceptual framework for analysing students' knowledge of programming. *Journal of Research on Computing in Education, 29*(3), 276-297.

McIver, L. (2000). *The effect of programming language on error rates of novice programmers.* 12th Annual Workshop of Psychology of Programmers Interest Group (PPIG), Corigliano.

McIver, L. (2002). *Evaluating languages and environments for novice programmers.* 14th Workshop of the Psychology of Programming Interest Group, Brunel University, UK.

McIver, L. & Conway, D. (1996). *Seven deadly sins of introductory programming language design.* 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96).

McIver, L. & Conway, D. (1999). *Grail: A zeroth programming language.* International Conference on Computing in Education, Chiba, Japan.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S. et al. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *ACM SIGCSE Bulletin, 38*(4), 182-194. doi: http://doi.acm.org/10.1145/1189136.1189185

Merrill, M. D. (2002). Knowledge objects and mental models. In D. Wiley (Ed.), *The instructional use of learning objects* (pp. 262-280). Bloomington, IN: Agency for Instructional Technology and the Association for Educational Communications and Technology.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review, 63*(2), 81-97.

Milne, I. & Rowe, G. (2002). Difficulties in learning and teaching programming—views of students and tutors. *Education and Information Technologies, 7*(1), 55-66.

Milner, W. W. (2010). *Concept development in novice programmers learning Java.* PhD thesis, The University of Birmingham, Birmingham.

Moons, J. & De Backer, C. (2009). Rationale behind the design of the eduvisor software visualization component. *Electronic Notes in Theoretical Computer Science, 224*(0), 57-65.

Moreno, R. (2006). When worked examples don't work: Is cognitive load theory at an impasse? *Learning and Instruction, 16*(2), 170-181.

Moreno, R. (2010). Cognitive load theory: More food for thought. *Instructional Science, 38*(2), 135-141.

Mow, I. T. C. (2008). Issues and difficulties in teaching novice computer programming. In M. Iskander (Ed.), *Innovative techniques in instruction* *technology, e-learning, e-assessment, and education* (pp. 199-204): Springer Netherlands.

Mulholland, P. (1997). *Using a fine-grained comparative evaluation technique to understand and design software visualization tools.* 7th workshop on Empirical studies of programmers, Alexandria, Virginia, USA.

Murnane, J. S. (1993). The psychology of computer languages for introductory programming courses. *New Ideas in Psychology, 11*(2), 213-228.

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing, 1*(1), 97-123. doi: http://dx.doi.org/10.1016/S1045-926X(05)80036-9

Myers, B. A., Ko, A. J. & Burnett, M. M. (2006). *Invited research overview: End-user programming.* CHI '06 extended abstracts on Human factors in computing systems, Montréal, Québec, Canada.

Myers, B. A., Pane, J. F. & Ko, A. (2004). Natural programming languages and environments. *Communications of the ACM, 47*(9), 47-52. doi: 10.1145/1015864.1015888

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C. D. et al. (2002). *Exploring the role of visualization and engagement in computer science education.* ITiCSE on Innovation and technology in computer science education, New York, NY, USA.

Navarro-Prieto, R. & Cañas, J. J. (2001). Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies, 54*(6), 799-829.

Návrat, P. (1996). A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology, 38*(1), 37-46.

Nielsen, J. (1993). *Usability engineering*. San Diego, CA: Academic Press.

Nienaltowski, M. M.-H., Pedroni, M. & Meyer, B. (2008). *Compiler error messages: What can help novices?* 39th SIGCSE technical symposium on Computer science education, Portland, OR, USA.

Nikolaidhs, S. (2008). Spinet - γλωσσομάθεια, from http://www.spinet.gr/glossomatheia/

Nishida, T., Harada, A., Yoshida, T., Nakamura, R., Nakanishi, M., Toyoda, H. et al. (2008). *Pen: A programming environment for novices - overview and practical lessons -.* World Conference on Educational Multimedia, Hypermedia and Telecommunications 2008, Vienna, Austria.

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. Stevens (Eds.), *Mental models* (pp. 7-14). Hillsdale, NJ: Erlbaum.

Odekirk-Hash, E. & Zachary, J. L. (2001). Automated feedback on programs means students need less help from teachers. *SIGCSE Bulletin, 33*(1), 55-59. doi: 10.1145/366413.364537

Ormerod, T. (1990). Human cognition and programming. In J.-M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 63-82). Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Paas, F., Renkl, A. & Sweller, J. (2003). Cognitive load theory and instructional design: Recent developments *Educational Psychologist, 38*(1), 1-4.

Paas, F. & Sweller, J. (2012). An evolutionary upgrade of cognitive load theory: Using the human motor system and collaboration to support the learning of complex cognitive tasks. *Educational Psychology Review, 24*(1), 27-45.

Paas, F., Van Gog, T. & Sweller, J. (2010). Cognitive load theory: New conceptualizations,specifications, and integrated research perspectives. *Educational Psychology Review, 22*(2), 115-121.

Paas, F. & van Merriënboer, J. J. G. (1994). Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach. *Journal of Educational Psychology, 86*(1), 122-133.

Pair, C. (1990). Programming, programming languages and programming methods. In J.-M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 9-19). Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Palumbo, D. B. (1990). Programming language/problem-solving research: A review of relevant issues *Review of Educational Research, 60*(1), 65-89.

Pane, J. F. & Myers, B. A. (1996). Usability issues in the design of novice programming systems (school of computer science technical report cmu-CS-96-132). Pittsburgh, PA: Carnegie Mellon University.

Pane, J. F. & Myers, B. A. (2000). *The influence of the psychology of programming on a language design: Project status report.* 12th Annual Meeting of the Psychology of Programmers Interest Group, Corigliano Calabro, Italy.

Pane, J. F. & Myers, B. A. (2006). More natural programming languages and environments. In H. Lieberman, F. Paternò & V. Wulf (Eds.), *End user development* (Vol. 9, pp. 31-50). Netherlands: Springer.

Pane, J. F., Myers, B. A. & Miller, L. B. (2002). *Using hci techniques to design a more usable programming system.* IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02).

Pane, J. F., Myers, B. A. & Ratanamahatana, C. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human - Computer Studies, 54*(2), 237-264. doi: 10.1006/ijhc.2000.0410

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books.

Paul, L. M., Simons, G. F. & Fennig, C. D. (2013). Ethnologue: Languages of the world Retrieved February, 2013, from http://www.ethnologue.com/statistics/size

Pea, R. D. (1984). Language-independent conceptual "bugs" in novice programming. Technical report no. 31. *Journal of Educational Computing Research, special issue on “Novice Programming”*.

Pea, R. D. & Kurland, D. M. (1987). On the cognitive effects of learning computer programming *Mirrors of minds: Patterns of experience in educational computing* (pp. 147-177). Norwood, NJ: Ablex Publishing Corp.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J. et al. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin 39*(4), 204-223.

Pedagogical Institute. (2004). Pedagogical institute - computer science - studies framework from http://www.pi-schools.gr/content/index.php?lesson_id=1

Pedhazur, E. J. (1997). *Multiple regression in behavioral research : Explanation and prediction* (3rd ed.). London: Harcourt Brace College Publishers.

Pedhazur, E. J. & Pedhazur Schmelkin, L. (1991). *Measurement, design, and analysis: An integrated approach*. Hillside, NJ. : Erlbaum.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research, 2*(1), 37-55.

Petre, M. (1995). Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM, 38*(6), 33-44. doi: 10.1145/203241.203251

Petre, M., Blackwell, A. F. & Green, T. R. G. (1998). Cognitive questions in software visualisation. In J. Stasko, J. Domingue, M. Brown & B. A. Price (Eds.), *Software visualization: Programming as a multi-media experience* (pp. 453-480). Cambridge, MA: MIT Press.

Pillay, N. & Jugoo, V. R. (2005). An investigation into student characteristics affecting novice programming performance. *SIGCSE Bulletin, 37*(4), 107-110. doi: 10.1145/1113847.1113888

Porter, R. & Calder, P. (2003). *Applying patterns to novice programming problems.* 2002 conference on Pattern languages of programs - Volume 13, Melbourne, Australia.

Powers, K. (2004). *Teaching computer architecture in introductory computing: Why? And how?* 6th Australasian Conference on Computing Education - Volume 30, Dunedin, New Zealand.

Powers, K., Gross, P., Cooper, S., McNally, M., Goldman, K. J., Proulx, V. et al. (2006). Tools for teaching introductory programming: What works? *ACM SIGCSE Bulletin, 38*(1), 560-561.

Price, B. A., Baecker, R. M. & Small, I. S. (1993). A principled taxonomy of software visualization. *Journal of Visual Languages & Computing, 4*(3), 211-266. doi: http://dx.doi.org/10.1006/jvlc.1993.1015

Putnam, R. T., Sleeman, D., Baxter, J. A. & Kuspa, L. K. (1989). A summary of misconceptions of high school basic programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 301-313). Hillsdale, NJ: Erlbaum.

Raaijmakers, J. G. W. (1993). The story of the two-store model: Past criticisms, current status, and future directions *Attention and performance xiv: Synergies in experimental psychology, artifical intelligence, and cognitive neuroscience* (pp. 467-488). Cambridge, MA, USA: MIT Press.

Rajala, T., Laakso, M.-J., Kaila, E. & Salakoski, T. (2008). Effectiveness of program visualization: A case study with the ville tool. *Journal of Information Technology Education 7*, 15-32.

Ramadhan, H. A., Deek, F. & Shihab, K. (2001). Incorporating software visualization in the design of intelligent diagnosis systems for user programming. *Artificial Intelligence Review, 16*(1), 61-84. doi: 10.1023/a:1011078011415

Ramalingam, V., LaBelle, D. & Wiedenbeck, S. (2004). *Self-efficacy and mental models in learning to program.* 9th annual SIGCSE conference on Innovation and technology in computer science education, Leeds, United Kingdom.

Reis, C. & Cartwright, R. (2004). Taming a professional IDE for the classroom. *SIGCSE Bulletin, 36*(1), 156-160. doi: 10.1145/1028174.971357

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K. et al. (2009). Scratch: Programming for all. *Communications of the ACM, 52*(11), 60-67. doi: 10.1145/1592761.1592779

Riker, A. (2010). *Natural language in programming: An English syntax-based approach for reducing the difficulty of first programming language acquisition.* MSc thesis, Brandeis University.

Rist, R. (1989). Schema creation in programming. *Cognitive Science, 13*, 389-414.

Rist, R. (1996). Teaching eiffel as a first language. *Journal of Object-Oriented Programming, 9*, 30-41.

Robillard, P. N. (1999). The role of knowledge in software development. *Communications of the ACM, 42*(1), 87-92. doi: 10.1145/291469.291476

Robins, A., Rountree, J. & Rountree, N. (2003). Learning and teaching programming: A review and discussion *Computer Science Education, 13*(2), 137-172.

Rogalski, J. & Samurçay, R. (1990). Acquisition of programming knowledge and skills. In J.-M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gilmore (Eds.), *Psychology of Programming*. Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Romero, P., du Boulay, B., Robertson, J., Good, J. & Howland, K. (2009). *Is embodied interaction beneficial when learning programming?* 3rd International Conference on Virtual and Mixed Reality: Held as Part of HCI International 2009, San Diego, CA.

Rose, J. M., Rose, A. M. & McKay, B. (2007). Measurement of knowledge structures acquired through instruction, experience, and decision aid use. *International Journal of Accounting Information Systems, 8*(2), 117-137.

Ross, R. (2000). Going backwards: Introductory programming languages. *SIGACT News, 31*(4), 65-73. doi: 10.1145/369836.571185

Sajaniemi, J. & Kuittinen, M. (2003). *Program animation based on the roles of variables.* 2003 ACM symposium on Software visualization, San Diego, California.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway & J. C. Spohrer

(Eds.), *Studying the novice programmer* (pp. 161-177). Hillsdale, NJ: Erlbaum.

Sanders, D. & Dorn, B. (2003). Classroom experience with jeroo. *Journal of Computing Sciences in Colleges, 18*(4), 308-316.

Satratzemi, M., Dagdilelis, V. & Evagelidis, G. (2001). A system for program visualization and problem-solving path assessment of novice programmers. *ACM SIGCSE Bulletin, 33*(3), 137-140.

Satratzemi, M., Xinogalos, S. & Dagdilelis, V. (2003). *An environment for teaching object-oriented programming: Objectkarel* 3rd IEEE International Conference on Advanced Learning Technologies.

Schnotz, W. & Kürschner, C. (2007). A reconsideration of cognitive load theory. *Educational Psychology Review, 19*(4), 496-508.

Schvaneveldt, R. W. (1990). *Pathfinder associative networks: Studies in knowledge organization*. Norwood, NJ: Ablex.

Scott, A. (2010). *Using flowcharts, code and animation for improved comprehension and ability in novice programming.* PhD thesis, University of Glamorgan, South Wales, UK.

Scott, A., Watkins, M. & McPhee, D. (2008, 7-11 April 2008). *E-learning for novice programmers; a dynamic visualisation and problem solving tool.* Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008.

Seifert, K. & Sutton, R. (2009). Educational psychology   Retrieved from http://globaltext.terry.uga.edu/booklist?cat=Education

Serrano, E., Quirin, A., Botia, J. & Cordón, O. (2009). Debugging complex software systems by means of pathfinder networks. *Information Sciences, 180*(5), 561-583.

Shaffer, D., Doube, W. & Tuovinen, J. (2003). *Applying cognitive load theory to computer science education.* 15th Workshop of the Psychology of Programming Interest Group, Keele UK.

Sheard, J., Simon, S., Hamilton, M. & Lönnberg, J. (2009). *Analysis of research into the teaching and learning of programming.* 5th international workshop on Computing education research workshop, Berkeley, CA, USA.

Sheil, B. A. (1981). The psychological study of programming. *ACM Computing Surveys, 13*(1), 101-120.

Shneiderman, B. & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences, 8*(3), 219-238.

Simon, B., Fitzgerald, S., McCauley, R., Haller, S., Hamer, J., Hanks, B. et al. (2007). *Debugging assistance for novices: A video repository.* ITiCSE on Innovation and technology in computer science education, Dundee, Scotland.

Sitze, K. L. (1992). Pcknot: Interlink. Retrieved from http://interlinkinc.net/KNOT.html

Sjøberg, S. (2007). Constructivism and learning. In E. Baker, B. McGaw & P. Peterson (Eds.), *International Encyclopaedia of Education 3rd Edition*. Retrieved from http://folk.uio.no/sveinsj/Constructivism_and_learning_Sjoberg.pdf

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850-858. doi: 10.1145/6592.6594

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. *Software Engineering, IEEE Transactions on, SE-10*(5), 595-609. doi: 10.1109/tse.1984.5010283

Soloway, E. & Spohrer, J. C. (Eds.). (1989). *Studying the novice programmer*. Hillsdale, NJ: Erlbaum.

Spohrer, J. C. & Soloway, E. (1989). Novice mistakes: Are the folk wisdoms correct? In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 401-416). Hillsdale, NJ: Erlbaum.

Spohrer, J. C., Soloway, E. & Pope, E. (1989). A goal/plan analysis of buggy pascal programs. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 355-399). Hillsdale, NJ: Erlbaum.

Stefik, A. & Gellenbeck, E. (2011). Empirical studies on programming language stimuli. *Software Quality Journal, 19*(1), 65-99. doi: 10.1007/s11219-010-9106-7

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257-285.

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction, 4*(4), 295-312.

Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review, 22*(2), 123-138.

Sweller, J., van Merriënboer, J. J. G. & Paas, F. (1998). Cognitive architecture and instructional design. *Educational Psychology Review 10*(3), 251-296.

Tang, T., Rixner, S. & Warren, J. (2014). *An environment for learning interactive programming.* Proceedings of the 45th ACM Technical Symposium on Computer Science Education - SIGCSE 2014, Atlanta, GA, USA.

Teague, D. & Lister, R. (2014). *Programming: Reading, writing and reversing.* Proceedings of the 2014 conference on Innovation and Technology in Computer Science Education, Uppsala, Sweden.

Teague, D. & Roe, P. (2008). *Collaborative learning: Towards a solution for novice programmers.* 10th conference on Australasian computing education - Volume 78, Wollongong, NSW, Australia.

Traver, V. J. (2010). On compiler error messages: What they *say* and what they *mean*. *Advances in Human-Computer Interaction, 2010*, 1-26. doi: 10.1155/2010/602570

Trumpower, D. L. & Goldsmith, T. E. (2004). Structural enhancement of learning. *Contemporary Educational Psychology, 29*(4), 426-446.

Truong, N. (2007). *A web-based programming environment for programmers.* PhD thesis, Queensland University of Technology, Brisbane.

Tu, J.-J. & Johnson, J. R. (1990). Can computer programming improve problem-solving ability? *SIGCSE Bulletin, 22*(2), 30-33. doi: 10.1145/126445.126451

Tudoreanu, M. E. & Kraemer, E. (2008). Balanced cognitive load significantly improves the effectiveness of algorithm animation as a problem-solving tool. *Journal of Visual Languages & Computing, 19*(5), 598-616. doi: http://dx.doi.org/10.1016/j.jvlc.2008.01.001

Tuovinen, J. E. (2000). *Optimising student cognitive load in computer education.* Australasian conference on Computing education, Melbourne, Australia.

Vagianou, E. (2006). *Program working storage: A beginner's model.* 6th Baltic Sea conference on Computing education research: Koli Calling 2006, Uppsala, Sweden.

Vainio, V. & Sajaniemi, J. (2007). *Factors in novice programmers' poor tracing skills*. Paper presented at the Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education, Dundee, Scotland.

van der Veer, G. C. (1989). Individual differences and the user interface. *Ergonomics, 32*(11), 1431-1449. doi: 10.1080/00140138908966916

van Haaster, K. & Hagan, D. (2004). *Teaching and learning with BlueJ: An evaluation of a pedagogical tool.* Information Science and Information Technology Education Joint Conference, Rockhampton, QLD, Australia.

van Merriënboer, J. J. G. (1990a). Instructional strategies for teaching computer programming: Interactions with the cognitive style reflection-impulsivity. *Journal of Research on Computing in Education, 23*(1), 45-53.

van Merriënboer, J. J. G. (1990b). Strategies for programming instruction in high school: Program completion vs. Program generation. *Journal of Educational Computing Research, 6*(3).

van Merriënboer, J. J. G. (1992). Strategies for computer-based programming instruction: Program completion vs. Program generation. *Journal of Educational Computing Research, 8*(3), 365-394.

van Merriënboer, J. J. G., Clark, R. E. & de Croock, M. B. M. (2002). Blueprints for complex learning: The 4c/id-model. *Educational Technology Research and Development, 50*(2), 39-61.

van Merriënboer, J. J. G. & Kirschner, P. A. (2007). *Ten steps to complex learning: A systematic approach to four-component instructional design*. Mahaw, New Jersey: Erlbaum.

van Merriënboer, J. J. G., Kirschner, P. A. & Kester, L. (2003). Taking the load off a learner's mind: Instructional design for complex learning. *Educational Psychologist, 38*(1), 5-13.

van Merriënboer, J. J. G. & Krammer, H. P. M. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science, 16*(3), 251-285.

van Merriënboer, J. J. G. & Paas, F. (1990). Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior, 6*(3), 273-289.

van Merriënboer, J. J. G., Schuurman, J. G., de Croock, M. B. M. & Paas, F. (2002). Redirecting learners' attention during training: Effects on cognitive load, transfer test performance and training efficiency. *Learning and Instruction, 12*(1), 11-37.

van Merriënboer, J. J. G. & Sweller, J. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review, 17*(2), 147-178.

van Mierlo, C., Jarodzka, H., Kirschner, F. & Kirschner, P. A. (2011). Cognitive load theory and e-learning. In Z. Yan (Ed.), *Encyclopedia of Cyberbehavior*: IGI Global.

Verhoeven, L., Schnotz, W. & Paas, F. (2009). Cognitive load in interactive knowledge construction. *Learning and Instruction, 19*(5), 369-375.

Vihavainen, A., Airaksinen, J. & Watson, C. (2014). *A systematic review of approaches for teaching introductory programming and their influence on success.* Proceedings of the tenth annual conference on International computing education research, Glasgow, Scotland, United Kingdom.

Virtanen, A. T., Lahtinen, E. & Järvinen, H.-M. (2005). *Vip, a visual interpreter for learning introductory programming with C++.* Koli Calling 2005 Conference, Koli, Finland.

Vogts, D., Calitz, A. & Greyling, J. (2008). *Comparison of the effects of professional and pedagogical program development environments on novice programmers.* 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, Wilderness, South Africa.

Vrachnos, E. & Jimoyiannis, A. (2008). *Dave: A dynamic algorithm visualization environment for novice learners.* 2008 8th IEEE International Conference on Advanced Learning Technologies.

Weinberg, G. M. (1971). *The psychology of computer programming*. New York, N.Y: Van Nostrand Reinhold Company.

Werhane, P. H., Hartman, L. P., Moberg, D., Englehardt, E. & Pritchard, M. (2009). *Social constructivism, mental models and problems of obedience.* EBEN Annual Conference Athens, Greece.

Westbrook, L. (2006). Mental models: A theoretical overview and preliminary study. *Journal of Information Science, 32*(6), 563-579.

White, G. L. & Sivitanides, M. P. (2003). A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics. *Journal of Information Systems Education, 13*(1), 59-66.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies, 23*(4), 383-390.

Wiedenbeck, S. & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies, 51*(1), 71-87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S. & Corritore, C. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11*(3), 255-282. doi: http://dx.doi.org/10.1016/S0953-5438(98)00029-0

Winslow, L. E. (1996). Programming pedagogy - a psychological overview. *ACM SIGCSE Bulletin, 28*(3), 17-22.

Wittgenstein, L. (1922). *Tractatus logico-philosophicus* London: Routledge & Kegan Paul.

Xinogalos, S., Satratzemi, M. & Dagdilelis, V. (2006). An introduction to object-oriented programming with a didactic microworld: Objectkarel. *Computers & Education, 47*(2), 148-171.

Yin, P. (2010). *Natural language programming based on knowledge.* 2010 International Conference on Artificial Intelligence and Computational Intelligence - Volume 02.

Young, R. M. (1981). The machine inside the machine: Users' models of pocket calculators. *International Journal of Man-Machine Studies, 15*(1), 51-85. doi: http://dx.doi.org/10.1016/S0020-7373(81)80023-5

Youngs, E. A. (1974). Human errors in programming. *International Journal of Man-Machine Studies, 6*(3), 361-376.

Ziegler, U. & Crews, T. (1999). An integrated program development tool for teaching and learning how to program. *SIGCSE Bulletin, 31*(1), 276-280. doi: 10.1145/384266.299786

Appendix A

INTERVIEW STUDY WITH GREEK TEACHERS OF COMPUTER SCIENCE

A.1. Consent Form

Participant Identification Number: _______________

PARTICIPANT'S CONSENT FORM (participant's copy)

Project title: A visual programming language and development environment for learning and teaching programming in Greece

- ► I understand that it is completely my decision to take part or not.
- ► I understand that I do not have to take part and do not have to give a reason why not.
- ► I understand that I am free to stop taking part at any time without giving a reason.
- ► I understand that results may appear in research publications.

  I understand my name or other personal information will not be mentioned.
- ► I understand that what I say will be recorded by the researcher, be examined to by anyone other than the research team without my written permission.
- ► I agree to take part in this study.

Name of Participant --------------------------- Signature--------------------Date --------

Name of Researcher --------------------------- Signature---------------------Date ------

Participant Identification Number: _______________

PARTICIPANT’S CONSENT FORM (researcher’s copy)

Project title: A visual programming language and development environment for learning and teaching programming in Greece

► I understand that it is completely my decision to take part or not.

► I understand that I do not have to take part and do not have to give a reason why not.

► I understand that I am free to stop taking part at any time without giving a reason.

► I understand that results may appear in research publications.

I understand my name or other personal information will not be mentioned.

► I understand that what I say will be recorded by the researcher, be examined to by anyone other than the research team without my written permission.

► I agree to take part in this study.

Name of Participant --------------------------- Signature--------------------Date --------

Name of Researcher --------------------------- Signature---------------------Date ------

A.2. Interview Questions

1. Which teaching programming languages and/or environments have you used for introductory programming to novices?
2. Do you use any other programming languages?
3. Which of the above do you nowadays use and what are the reasons of your choice?
4. Which feature do you consider to be the major advantage of the programming language and/or environment of your choice?
5. Do you find any disadvantages in the programming language and/or environment of your choice?
6. What features do you believe a programming language and/or environment should have to be suitable for teaching and learning programming to novices?
7. How would you describe your teaching method?
8. From your experience, which of the following programming issues do tend novices to have problems with during the introductory programming course?
    - Confuse input/output commands
    - Misunderstand the role and use of variables
    - Misunderstand the concept of data types
    - Misunderstand the use and function of iteration statements
    - Misunderstand the use and function of selection statements

- Misunderstand the use of arrays
- Misunderstand the use of functions
- Even knowing the steps of a solution to a problem and the commands of the programming language cannot create the program.
- Other

A.3. Executive Summary Report on Interviews with Greek Teachers

This report concerns a series of interviews with Greek teachers for this research project. An initial stage of this project is the identification of a set of design principles for educational programming software. A following stage is to design and implement a programming tool as an embodiment of this set of principles. Finally, the effectiveness of this set on novices' programming performance is studied via the evaluation of the programming tool.

The set of design principles is derived from pedagogical theories as well as from common difficulties novices face during their introductory programming module. A review of the international literature revealed a great deal of appropriate evidence for identifying the design principles. However, little information is available in the international literature about the current status of introductory programming education in Greece. This is the case for the difficulties that Greek novice programmers face as well. In order to inform the findings that are obtained from the review of the international literature on this matter, a series of informal interviews with teachers of computer science was carried out in Greece.

Interviews took place in the cities of Patras and Lefkas, between 01/11/2008 and 09/11/2008. Ethics approval for the interviews had previously been granted in advance. Four Greek teachers with experience of teaching programming to novices in secondary and tertiary education volunteered to participate and were informed about the aim of the research. During these interviews, I was keeping notes on the following issues:

- the software tool(s) that participants have used to teach programming and their opinion on its/their effectiveness;

- the problems that novices face during introductory programming courses, according to participants' experience; and
- the desired features for an educational programming tool for teaching and learning introductory programming

Due to the small number of respondents and questions, the analysis of the data that were collected from the interviews was rather simple. The answers of all respondents were categorised per question. Each category was created based on the answers of the respondents. More specifically, analysis was conducted either by identifying common answers and emerging patterns or by synthesising all responses to produce an aggregated/combined response for a category. The results of the analysis of the interviews follow.

The results of these informal interviews revealed that a set of general-purpose as well as educational programming languages was used for introductory programming in Greece. This set includes Basic, Fortran, C, Visual Basic, Java, and Glossomatheia. The main disadvantages of these languages are that the majority of these programming languages (a) are too complex and (b) provide features beyond those that are required for pedagogical purposes. In particular, they lack a user-friendly environment, they are in the English language and their set of instructions is too large. Thus, an unnecessary complexity is imposed by the use of these tools, which retards program creation.

With respect to the problems that novices face, interviews revealed that a major difficulty is that even when novices know the steps of a solution and the commands of a programming language they are unable to structure the final program. This occurs because novices seem to lack the knowledge and experience of using programming commands to translate the steps of the solution into a program. Other major problems that beginners have are distinguishing the commands they should use in a program and lacking knowledge of how data types, arrays and functions should be used. Issues that seem to cause fewer problems are input/output commands, iteration and selection statements. Most of the interviewees commented that mathematics and computer architecture could lead novices towards a more analytical and structured way of thinking, which could help them to conceptualise and create their own programs. These findings are in agreement with the difficulties of novices that are reported in the international literature and are reviewed in Section 2.4.

According to the interviewees, an educational programming language suitable for teaching introductory programming should have a set of desirable characteristics in order to serve its educational purpose. The results of the interviews revealed that this set should include a visual environment, which would be supportive and simple to learn and use, a small and simple set of instructions in the Greek language, easy syntax – close to natural language, and a visual representation of programming elements. Concisely, a new programming language and integrated development environment should support the following features:

- graphical development environment, which will help and guide students during the creation of programs;
- simple syntax in the Greek language;
- semantics that can be easily understood by non-programmers;
- visualisation of basic programming structures and examples;

- a small set of commands; and
- documentation of the set of instructions and supported features by the programming language and development environment.

Appendix B

THE KOIOS PROGRAMMING ENVIRONMENT AND CHARACTERISTICS

B.1. The Koios Programming Environment

The Koios software can be found on the attached CD. Requirements for running Koios and information on how to run and use Koios are in the file ReadMe1st.txt on the CD. A number of sample programs are also included.

[The CD will be included before submission]

B.2. Syntax of Koios

The valid symbols that can be used in a program that is written in the programming language supported by Koios are presented in the form of the state machine presented in Figure B.1. White spaces are used to separate identifiers, numbers and symbols as well as to format text code and increase its readability, for example using indentation.

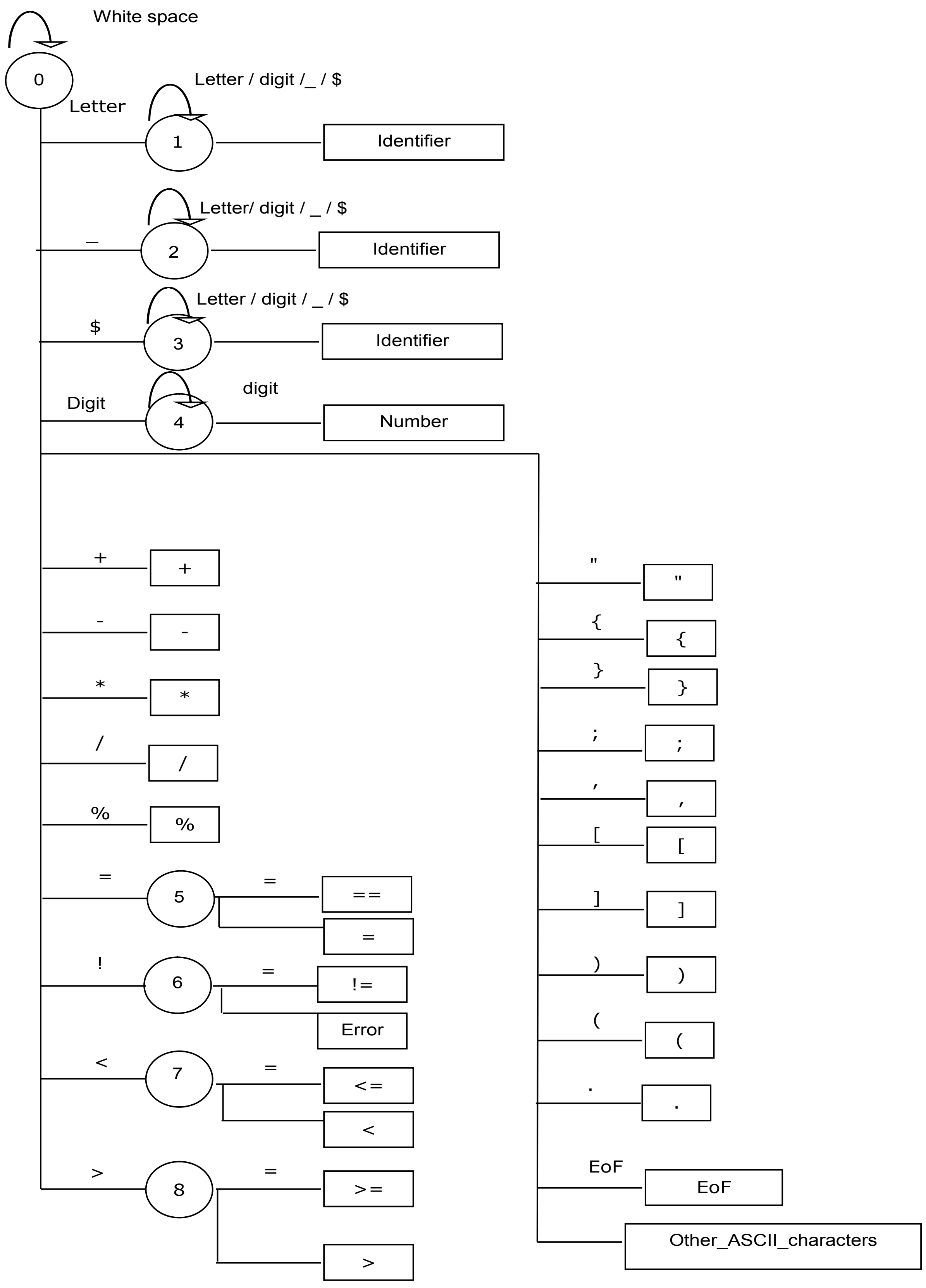


Figure B.1. The state machine that describes the syntactic rules of Koios.

B.3. Grammar of Language Supported by Koios

Based on the symbols produced by the lexical analyser (see Figure B.1), the grammar of the programming language that is supported by Koios follows in Backus-Naur Form (Marcotty & Ledgard, 1986). According to this form, terms within < > are non-terminal and can be replaced by the relevant production rule following the *::=* symbol. Terms in bold are terminal, while the | symbol indicates that one or more terms separated by it, can form a valid instance of the rule. The symbol ε is used to specify an empty term. Finally, a term suffixed with the symbol * means that this term can be repeated zero or more times.

<PROGRAM>::= **PROGRAM** IDENTIFIER <DECLARATIONS> <SUBPROGRAMS> **END OF PROGRAM**

<DECLARATIONS>::= (<CON_DECLERATION> | <VAR_DECLARATION> | <ARRAY_DECLARATION>)*

<CON_DECLERATION>::= **CONSTANT** IDENTIFIER **WITH INITIAL VALUE =** <INIT_VALUES> **;**

<INIT_VALUES>**::=** <IR_NUMBER> | <CHARACTER> | <STRING>

<IR_NUMBER>::= <OPTIONAL-SIGN> NUMBER | <OPTIONAL-SIGN> NUMBER**.**NUMBER

<CHARACTER>::= ‘ WHITE_CHARACTER | LETTER | DIGIT | **+** | **-** | * | **/** | **%** | **!** | **=** | **<** | **>** | _
| **$** | **.** | **;** | **,** | **[** | **]** | **{** | **}** | **(** | **)** | EOF | OTHER_ASCII_CHARACTERS ’

<STRING>::= **“** <CHARACTER> (<CHARACTER>)* **”**

<VAR_DECLARATION> ::= **VARIABLE** IDENTIFIER [**WITH INITIAL VALUE =** <INIT_VALUES>] **;**

<ARRAY_DECLARATION>::= **ARRAY** IDENTIFIER **[**NUMBER**]** (ε | **[** NUMBER **]** | **WITH INITIAL VALUES =** <ARRAY_VALUES> | **[** NUMBER **] WITH INITIAL VALUES =** <ARRAY_VALUES>) **;**

<ARRAY_VALUES>::= <ARRAY_VALUES_1D> | <ARRAY_VALUES_2D>

<ARRAY_VALUES_1D>::= **{** <INIT_VALUES>**,** <INIT_VALUES> (**,**<INIT_VALUES>)* **};**

<ARRAY_VALUES_2D>::= **{** <ARRAY_VALUES_1D>**,** <ARRAY_VALUES_1D> (**,**<ARRAY_VALUES_1D>)***}** **;**

<SUBPROGRAMS>::= (<PROC_OR_FUNC>)* PROCMAIN (<PROC_OR_FUNC>) *

<PROCMAIN>::= **MAIN ROCEDURE**<PROC_OR_FUNC_BODY>**END OF MAIN PROCEDURE**

<PROC_OR_FUNC>::=**PROCEDURE** IDENTIFIER <PROC_OR_FUNC_BODY> **END OF PROCEDURE ID** | <DATATYPE> **FUNCTION** IDENTIFIER <PROC_OR_FUNC_BODY> **END OF FUNCTION** IDENTIFIER

<DATATYPE>::= **INTEGER** | **REAL** | **CHARACTER** | **BOOLEAN |STRING**

<PROC_OR_FUNC_BODY>::= <PROC_ BODY> | < FUNC_BODY>

<PROC_BODY>::= <FORMALPARS> <DECLARATIONS> <SEQUENCE>

<FUNC_BODY>::= <FORMALPARS> <DECLARATIONS> <SEQUENCE><RETURN-STAT>

<FORMALPARS>::= **(**ε | <FORMALPARLIST> **)**

<FORMALPARLIST>::= <DATATYPE> IDENTIFIER(**,**<DATATYPE> IDENTIFIER)*

<SEQUENCE>::= <STATEMENT> (<STATEMENT>)*

<STATEMENT>::= ε | <READ_STAT>| <WRITE_STAT> | <IF-STAT> | <FOR-STAT> | <WHILE-STAT> | <DO-WHILE-STAT> | <ASSIGNMENT-STAT> | <CALL-STAT>

<READ_STAT>::= **READ** <READ_ARGUEMS> (**,**<READ_ARGUEMS>)* **;**

<READ_ARGUEMS>:== [<STRING>] IDENTIFIER | [<STRING>] IDENTIFIER **[**EXPRESSION**]** (ε | **[**EXPRESSION**]**)

<WRITE_STAT>::= **WRITE** <WRITE_ARGUEMS> (**,** <WRITE_ARGUEMS>)* **;**

<WRITE_ARGUEMS>::= <DATA_ARGUEMS>

<DATA_ARGUEMS>::=<EXPRESSION> | <CHARACTER> | <STRING> | <CONDITION>

<IF-STAT>::= **IF** <CONDITION> <SEQUENCE> <ELSEPART> **END OF IF**

<ELSEPART>::= ε | (**ELSE IF** <CONDITION> <SEQUENCE>)* | **ELSE** <SEQUENCE>

<FOR-STAT>::= **FOR** <EXPRESSION> **AND** <CONDITION> **LOOP WITH STEP** <EXPRESSION> <SEQUENCE> **END OF FOR**

<WHILE-STAT>::= **WHILE THE CONDITION** <CONDITION> **IS VALID REPEAT** <SEQUENCE> **END OF WHILE**

<DO-WHILE-STAT>::= **DO** <SEQUENCE> **WHILE THE CONDITION** <CONDITION> **IS TRUE**

<ASSIGNMENT-STAT>::= **ASSIGN TO** <EXPRESSION> **THE EXPRESSION** <EXPRESSION> **;**

<CALL-STAT>::= **CALL PROCEDURE** <IDENTIFIER> <ACTUALPARS> **;**

<RETURN-STAT>::= **RETURN** <DATA_ARGUEMS> **;**

<ACTUALPARS>::= **(**ε| <ACTUALPARLIST> **)**

<ACTUALPARLIST>::= <ACTUALPARITEM> (**,** <ACTUALPARITEM>)*

<ACTUALPARITEM>::= <DATA_ARGUEMS>

<CONDITION>::= <BOOLTERM> (**OR** <BOOLTERM>)*

<BOOLTERM>::= <BOOLFACTOR> (**AND** <BOOLFACTOR>)*

<BOOLFACTOR>::=**NOT (**<CONDITION>**)** | **(**<CONDITION>**)** | <EXPRESSION> <RELATIONAL-OPER> <EXPRESSION> | IDENTIFIER| IDENTIFIER <ACTUALPARS> | IDENTIFIER **[**EXPRESSION**]** (ε | **[**EXPRESSION**]**) | **TRUE** | **FALSE**

<EXPRESSION>::= <OPTIONAL-SIGN> <TERM> (<ADD-OPER> <TERM>)*

<TERM> ::= <FACTOR> (<MUL-OPER> <FACTOR>)*

<FACTOR>::= NUMBER | CONSTANT | **(**<EXPRESSION>**)** | IDENTIFIER | IDENTIFIER <ACTUALPARS> | IDENTIFIER **[**EXPRESSION**]** (ε | **[**EXPRESSION**]**)

<RELATIONAL-OPER>::= **==** | **!=** | **<** (ε | **=**) | **>** (ε | **=**)

<ADD-OPER>::= **+** | **-**

<MUL-OPER>::= ***** | **/** | **%**

<OPTIONAL-SIGN>::= ε | <ADD-OPER>

B.4. Errors Detected by Koios

The Koios programming environment can detect a number of errors and notify users about these with messages, whose wording is meant to be appropriate, understandable and non-technical. Checks for errors are made during the creation or modification of a programming item and, if applicable, before dropping or pasting a programming item. The errors that can be detected regarding each programming item follow.

Constants

The name of constants

- cannot be empty;
- must start with letter, $ or _;
- must contain only letters, numbers, $ and _;
- cannot be a reserved word.

In the case of global constants, their name

- cannot be an identifier that is already used for another local or global constant, variable, array or parameter;
- cannot be an identifier that is already used as a parameter of a function or procedure.

In the case of local constants, their name

- cannot have the same name with the procedure or function within the same scope;
- cannot have the same name with an input parameter within the same scope;

- cannot have the same name with an identifier that is already used for other local programming items;
- cannot have the same name with a global constant, variable or array.

The initial value of constants must be initialised during the creation of the constant.

The initial value of *integer* constants

- must contain only integer numbers;
- must be between Integer.MIN(= -2147483648) and Integer.MAX (= 2147483647).

The initial value of *real* constants

- must contain only real numbers;
- must be between Float.MIN (= 1.401298464324817E-45f) and Float.MAX(= 3.4028234663852886E38f).

The initial value of *character* constants must contain just a single character.

Variables

The name of variables

- cannot be empty;
- must start with letter, $ or _;
- must contain only letters, numbers, $ and _;
- cannot be a reserved word.

In the case of global variables, their name

- cannot be an identifier that is already used for another local or global constant, variable, array or parameter;

- cannot be an identifier that is already used as a parameter of a function or procedure.

In the case of local variables, their name

- cannot have the same name with the procedure or function within the same scope;
- cannot have the same name with an input parameter within the same scope;
- cannot have the same name with an identifier that is already used for other local programming items;
- cannot have the same name with a global constant, variable or array.

The initial value of variables can be either 418ninitialized or initialised. In the latter case it cannot be empty.

The initial value of *integer* variables

- must contain only integer numbers;
- must be between Integer.MIN(= -2147483648) and Integer.MAX (= 2147483647).

The initial value of *real* variables

- must contain only real numbers;
- must be between Float.MIN (= 1.401298464324817E-45f) and Float.MAX(= 3.4028234663852886E38f).

The initial value of *character* variables must contain just a single character.

Arrays

The name of arrays

- cannot be empty;

- must start with letter ,$ or _;
- must contain only letters, numbers, $ and _;
- cannot be a reserved word.

In the case of global arrays, their name

- cannot be an identifier that is already used for another local or global constant, variable, array or parameter;
- cannot be an identifier that is already used as a parameter of a function or procedure.

In the case of local arrays, their name

- cannot have the same name with the procedure or function within the same scope;
- cannot have the same name with an input parameter within the same scope;
- cannot have the same name with an identifier that is already used for other local programming items;
- cannot have the same name with a global constant, variable or array.

The dimension(s) of arrays

- cannot be empty;
- must be integer(s);
- must be greater than one.

The initial values of arrays can be either 419ninitialized or initialised. In the latter case they cannot be empty.

The values of the elements of *integer* arrays

- must contain only integer numbers;

- must be between Integer.MIN(= -2147483648) and Integer.MAX (= 2147483647).

The values of the elements of *real* arrays

- must contain only real numbers;
- must be between Float.MIN (= 1.401298464324817E-45f) and Float.MAX(= 3.4028234663852886E38f).

The values of the elements of *character* arrays must contain just a single character.

The values of the elements of *Boolean* arrays must be *true* or *false*.

Procedures

The name of procedures

- cannot be empty;
- must start with letter, $ or _;
- must contain only letters, numbers, $ and _;
- cannot be a reserved word;
- cannot be an identifier that is already used for another procedure or function.

The name(s) of parameter(s) of procedures

- cannot be empty;
- must start with letter, $ or _;
- must contain only letters, numbers, $ and _;
- cannot be a reserved word;
- cannot be an identifier that is already used for a global constant, variable or array;

- cannot be an identifier that is already used for a local constant, variable or array;
- cannot be the same with the name of its procedure;
- cannot be the same with the name of a previous parameter of the same procedure.

Functions

The name of functions

- cannot be empty;
- must start with letter, $ or _;
- must contain only letters, numbers, $ and _;
- cannot be a reserved word;
- cannot be an identifier that is already used for another procedure or function.

The name(s) of parameter(s) of function

- cannot be empty;
- must start with letter,$ or _;
- must contain only letters, numbers,$ and _;
- cannot be a reserved word;
- cannot be an identifier that is already used for a global constant, variable or array;
- cannot be an identifier that is already used for a local constant, variable or array;
- cannot be the same with the name of its procedure;

- cannot be the same with the name of a previous parameter of the same procedure.

The return command of a function

- cannot be missing from the body of the function and all clauses of an if-statement ;
- can be missing from the body of the function if it is contained in all clauses of an if-statement ;
- can be missing from all clauses of an if-statement if it is contained in the body of the function;
- cannot be missing from all clauses of an if-statement if it is contained at least in one clause;
- cannot have more than one instances in the body of the function or in each clause of an if-statement;
- must be the last command of the function or of each clause of an if-statement.

Read commands must have at least one argument.

Call commands cannot be empty.

Assign commands

- At least one variable or array element must be used in an assign command.

Appendix C

MATERIAL FOR DATA-COLLECTION

C.1. Evaluating Compliance of Programming Tools with the Set of Design Principles

The criteria that were used for evaluating compliance of Koios, Glossomatheia, MicroworldsPro and C with each design principle follow.

Match of users' natural language with the linguistic attributes of the PL and *IDE*

The main aspect that was checked for deciding on the compliance between PLs and IDEs and this principle was whether they incorporated users' natural language. Because these programming tools were going to be used by Greek students, they were considered fully compliant with this principle, only if the language used for the communication and interaction between users and programming tool, namely the PL and IDE, was the Greek language.

Syntax and semantics of instructions supported by the PL

The following features of PLs and IDEs were taken into account for measuring compliance with the second principle (a) closeness of PL's syntax to users' natural language, (b) the degree of ease with which commands' semantics can convey their functionality and (c) readability. Factors that contribute to the readability of programs are, for example, the use of different colours for the textual form of different programming concepts in programs and the indentation of blocks of code to identify commands that belong to different control flow structures, such as conditional and iteration statements.

Visualisation

The features used to measure the compliance of programming tools and visualisation were (a) the amount of information regarding program creation and modification that is graphically presented, (b) the support of multiple views of programs (for example, textual or visual), (c) visual control over the execution of programs and (d) visual feedback and explanations during execution.

Abstraction of commands provided by the PL

The measurement of compliance between the PLs and IDEs and abstraction of commands was based on the ease of using each command without having an in-depth knowledge of the technical details regarding its execution and implementation. For example, using an output statement does not require knowing the type of the variables used in this statement.

Small set of instructions

Compliance with this principle was measured using the total number of the commands supported by the PL. Koios, Glossomatheia and MicroworldsPro were considered to support a small number of commands. This was not the case for C, which as a professional PL, offers support for advanced programming instructions.

Provision of error messages

In order to decide on the compliance of programming tools and the provision of their error messages, two factors were considered. The first one was the comprehensibility and precision of explaining the source of errors. The second was the accuracy and clarity in describing the actions required for correcting errors.

A high level of interaction with the IDE

The compliance between the last design principle and programming tools was measured by the degree of interaction between the users and the IDE during program creation and execution. Features provided by IDEs, which were considered to provide a higher level of interaction include (a) supporting ‘dialogues’ for error prevention and guiding users during program creation, (b) users’ control over the execution of programs and (c) providing immediate feedback to users, for example with information regarding commands, execution or errors.

C.2. Consent Form for Parents/Guardians of Secondary-Education Participants

Participant Identification Number:_______________

PARTICIPANT'S CONSENT FORM (participant's copy)

Project title: A Greek visual programming language and development environment for novice programmers

► I understand that it is completely my decision to take part or not.

► I understand that I do not have to take part and do not have to give a reason why not.

► I understand that I am free to stop taking part at any time without giving a reason.

► I understand that results may appear in research publications.

► I understand my name or other personal information will not be mentioned.

► I understand that my written answers within the scope of this module will be recorded/filed by the researcher and examined by members of the research team and the computer-science teacher of my class.

► I agree to take part in this study.

Name of Participant ------------------- Signature---------------------Date --------

Name of Researcher ---------------------- Signature---------------Date ------

Participant Identification Number:_______________

PARTICIPANT'S CONSENT FORM (researcher's copy)

Project title: A Greek visual programming language and development environment for novice programmers

► I understand that it is completely my decision to take part or not.

► I understand that I do not have to take part and do not have to give a reason why not.

► I understand that I am free to stop taking part at any time without giving a reason.

► I understand that results may appear in research publications.

► I understand my name or other personal information will not be mentioned.

► I understand that my written answers within the scope of this module will be recorded/filed by the researcher and examined by members of the research team and the computer-science teacher of my class.

► I agree to take part in this study.

Name of Participant ------------------ Signature--------------------Date --------

Name of Researcher ---------------------- Signature---------------Date ------

C.3. Information for Secondary-Education Participants

Thank you for taking part in this study. The aim of the project is to evaluate programming tools for Greek students of computer programming. You will be asked to use Koios, MicroworldsPro and Glossomatheia, depending on experimental condition as your basic tool in the introductory programming module. During this time you will be asked to take some tests as part of your course evaluation and fill in some questionnaires. Your views or any comments on your experience during this course would be more than welcome.

C.4. Lessons/Measurements Plan for Secondary-Education Quasi-Experiments

Table C.1 presents the plan used for conducting the study with secondary-education students. This plan was based on the official CS curriculum for the third grade of secondary education in Greece and incorporated the knowledge measurements according to the quasi-experimental design for secondary-education (see Section 4.2.2.3).

Table C.1.
Plan of Lessons and Measurements

Week	Programming Concept	Example in Classroom*	Homework Assignment**	Declarative-Knowledge Measurement***	Procedural-Knowledge Measurement****
1	Introduction to experiment and presentation of programming environment			At the beginning of the lesson	
2	Output statement	Lesson 1 Example(s)	Lesson 1 was introductory and did not include any homework		
3	Variables I – Input statement	Lesson 2 Example(s)	Lesson 2 Homework		
4	Variables II	Lesson 3 Example(s)	Lesson 3 Homework		
5	Assignment statement I	Lesson 4 Example(s)	Lesson 4 Homework		At the beginning of the lesson
6	Assignment statement II	Lesson 5 Example(s)	Lesson 5 Homework		

Table C.1 (continued)

Week	Programming Concept	Example in Classroom*	Homework Assignment**	Declarative-Knowledge Measurement***	Procedural-Knowledge Measurement****
7	Second declarative-knowledge and procedural-knowledge measurement			At the beginning of the lesson	After declarative-knowledge measurement
8	Iteration Statement	Lesson 6 Example(s)	Lesson 6 Homework		
9	Procedures	Lesson 7 Example(s)	Lesson 7 Homework		
10	Conditional Statement I	Lesson 8 Example(s)	Lesson 8 Homework		
11	Conditional Statement II	Lesson 9 Example(s)	Lesson 9 Homework		
12	Third declarative-knowledge and procedural-knowledge measurement			At the beginning of the lesson	After declarative-knowledge measurement
13	Practical test ***** (administered only in the second round of data collection)				

* The examples that were used in both rounds of data collection can be found in Appendix C.5.
** The homework assignments that were used in both rounds of data collection can be found in Appendix C.6.
*** The declarative-knowledge measurement material that was used in both rounds of data collection can be found in Appendix C.7.
**** The procedural-knowledge measurement material that were used in the first and second round of data collection can be found in Appendices C.8, C.9 and C.10 and Appendices C.11, C.13 and C.14, respectively.
*****The items of the practical test can be found in Appendix C.15

C.5. Classroom Examples Used in the Secondary-Education Quasi-Experiments

The following example programs were used during the lesson per week.

Lesson 1 Example(s)

Subject: Output statement.

1. Create simple programs that calculate the result of simple arithmetic expressions with numbers, for example write 25*12 , followed by more complicated expressions like write (17*3) / (2+5).
2. Create simple programs using strings. Use write command to print simple messages to screen.
3. Create programs combining strings and arithmetic expressions.

Lesson 2 Example(s)

Subject: Variables I - Input statement.

1. Create a program with a string variable that requests from the user to input his/her name to the variable and prints the value of the variable to screen.
2. Create a program with an integer variable that requests from the user to input his/her favourite number to the variable and prints a message and the value of the variable to screen.

Lesson 3 Example(s)

Subject: Variables II

1. Create a program with a string variable that requests from the user to input his/her name to the variable and prints a message and the value of the variable to screen.
2. Create a program with an integer variable that requests from the user to input a number to the variable and prints a message and the result of multiplying the value of the variable by two to screen.

Lesson 4 Example(s)

Subject: Assignment statement I

1. Create a program with a string variable STATE that assigns to STATE the value “water”, prints to screen the value of STATE, assigns to STATE the value of “ice” and prints again the new value of STATE.
2. Create a program with two integer variables A and B, that assigns the value of 100 to variable A, assigns the value of A to B and prints a message, the value of A, a second message and the value of B to screen.

Lesson 5 Example(s)

Subject: Assignment statement II

1. Create a program with two integer variables A and B that assigns the value of 100 to variable A, requests from the user to input an integer value to variable B and prints a message and the sum of variables A and B to screen.

Lesson 6 Example(s)

Subject: Iteration Statement

1. Create a program that uses a for-statement to print to screen the numbers from one to 10.
2. Create a program with an integer variable A that assigns a value to A and uses a for-statement to print to screen the products: A * 1, A * 2, …, A * 10 .
3. Create a program with an integer variable A that requests from the user to input an integer value A and uses a for-statement to print to screen the products: A * 1, A * 2, …, A * 10.

Lesson 7 Example(s)

Subject: Procedures

1. Create a program with a procedure that prints to screen the numbers from one to 10.

2. Create a program with an integer variable A and a procedure with an integer input parameter that prints to screen the second power of the input parameter, requests from the user to input an integer value to A and calls the procedure to calculate and print the second power of A.

Lesson 8 Example(s)

Subject: Conditional statement

1. Create a program with an integer variable A that requests from the user to input an integer value to A and prints a message to screen if the value of A is positive (>= 0) or negative.

Lesson 9 Example(s)

Subject: Conditional statement II

1. Create a program with two integer variables A and B that requests from the user to input two integer values to A and B. Assuming that A and B are the two coefficients of a linear equation (A * X + B = 0) the program either calculates and prints to screen the solution of the equation or prints to screen a message informing that the equation does not have a solution.

C.6. Homework Assignments Used in the Secondary-Education Quasi-Experiments

The following homework assignments were given at the end of each lesson.

Lesson 2 Homework

Subject: Variables I – Input statement.

- Complete the statements of or describe in natural language the steps for creating a program with a string and an integer variable that requests from the user to input a name and an age to the variables and prints to screen the name and the age.

Lesson 3 Homework

Subject: Variables II

- Complete the statements of or describe in natural language the steps for creating a program with a string and an integer variable that requests from the user to input a name and an age to the variables and prints to screen: the message “My name is ”, the value of the string variable, the message “and I am” and the value of the integer variable.

Lesson 4 Homework

Subject: Assignment statement I

- Complete the statements of or describe in natural language the steps for creating a program with two integer variables A and B, that assigns the value of 100 to variable A, assigns the value of A plus 10 to B and prints the message, the value of A, a second message and the value of B to screen.

Lesson 5 Homework

Subject: Assignment statement II

- Given that ANIMAL is a sting variable and X is an integer variable, write next to each output statement the result of its execution. Do not forget to consider possible effects of previous statements on each statement.
    - Assign “LION” to ANIMAL
    - Output_statement ANIMAL
    - Output_statement LION
    - Output_statement “ANIMAL”
    - Assign DOG to ANIMAL
    - Output_statement animal
    - Output_statement “I have a” ANIMAL
    - Assign 3 to X

  - Output_statement X
  - Output_statement “X”
  - Output_statement 12+5*X

Lesson 6 Homework

Subject: Iteration Statement II

- Complete the statements of or describe in natural language the steps for creating a program with an integer variable A that requests from the user to input an integer value A and uses a for-statement to print to screen the products: A * 11, A * 12, …,A * 20.

Lesson 7 Homework

Subject: Procedures

- Complete the statements of or describe in natural language the steps for creating a program with an integer variable A and a procedure with an integer input parameter. The procedure prints to screen the number from one to the value of the input parameter. The program requests from the user to input an integer value to A and calls the procedure to print to screen the numbers of one A.

Lesson 8 Homework

Subject: Conditional Statement

- Complete the statements of or describe in natural language the steps for creating a program with an integer variable A that requests from the user to input an integer number to A and prints the absolute value of this number.

Lesson 9 Homework

Subject: Conditional statement II

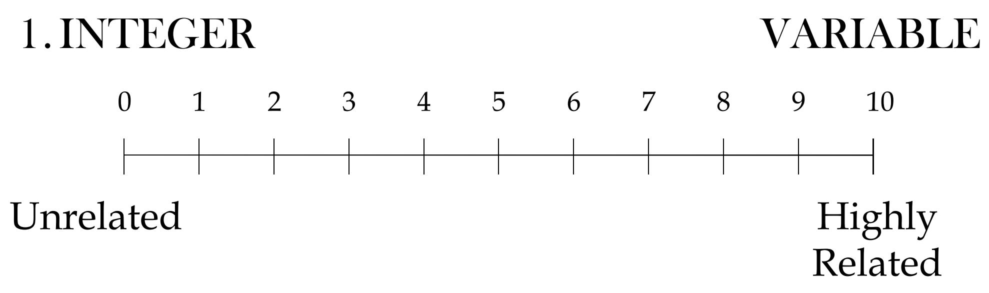
- Complete the statements of or describe in natural language the steps for creating a program with two integer variables A and B that requests from the user to input two integer numbers to A and B prints to screen which is the greater of the two numbers.

C.7. Material for Every Declarative-Knowledge Measurement for Secondary-Education Quasi-Experiments (First and Second Round)

The 78 different pairs that were used for ratings of concept pairs during declarative-knowledge measurement follow. The pairs were randomised before the rating task.

1	integer	variable
2	integer	string
3	integer	input parameter
4	integer	assignment
5	integer	arithmetic expression
6	integer	input command
7	integer	conditional statement
8	integer	output command
9	integer	logical condition
10	integer	procedure
11	integer	procedure call
12	integer	iteration
13	variable	string
14	variable	input parameter
15	variable	assignment
16	variable	arithmetic expression
17	variable	input command
18	variable	conditional statement
19	variable	output command
20	variable	logical condition
21	variable	procedure
22	variable	procedure call
23	variable	iteration
24	string	input parameter
25	string	assignment
26	string	arithmetic expression
27	string	input command
28	string	conditional statement
29	string	output command
30	string	logical condition
31	string	procedure
32	string	procedure call
33	string	iteration
34	input parameter	assignment
35	input parameter	arithmetic expression
36	input parameter	input command
37	input parameter	conditional statement

38	input parameter	output command
39	input parameter	logical condition
40	input parameter	procedure
41	input parameter	procedure call
42	input parameter	iteration
43	assignment	arithmetic expression
44	assignment	input command
45	assignment	conditional statement
46	assignment	output command
47	assignment	logical condition
48	assignment	procedure
49	assignment	procedure call
50	assignment	iteration
51	arithmetic expression	input command
52	arithmetic expression	conditional statement
53	arithmetic expression	output command
54	arithmetic expression	logical condition
55	arithmetic expression	procedure
56	arithmetic expression	procedure call
57	arithmetic expression	iteration
58	input command	conditional statement
59	input command	output command
60	input command	logical condition
61	input command	procedure
62	input command	procedure call
63	input command	iteration
64	conditional statement	output command
65	conditional statement	logical condition
66	conditional statement	procedure
67	conditional statement	procedure call
68	conditional statement	iteration
69	output command	logical condition
70	output command	procedure
71	output command	procedure call
72	output command	iteration
73	logical condition	procedure
74	logical condition	procedure call
75	logical condition	iteration
76	procedure	procedure call
77	procedure	iteration
78	procedure call	iteration



1. INTEGER VARIABLE

0 1 2 3 4 5 6 7 8 9 10

Unrelated

Highly Related

Figure C.1. A slide that was used in the presentation of the 78 concept pairs. Similar slides were used for each pair during the presentation

C.8. First Procedural-Knowledge Test for Secondary-Education Quasi-Experiment (First Round)

The following items were included in the first procedural-knowledge test. This measurement took place in the beginning of the fourth lesson (since the beginning of the study), and it took approximately 10-15 minutes to complete. The procedural knowledge of integer, arithmetic expression, string, output-statement, input-statement and variable was measured.

[5 points] Question I: Write under each command the output produced after its execution:
1. [Output statement] 12*3 *[2.5 points if correct]*

....
2. [Output statement] “12*3” *[2.5 points if correct]*

…………………………………………………………...

[5 points] Question II: Given that the value of variable A is 10, write under each command the output produced after its execution:
1. [Output statement] “A+10” *[2.5 points if correct]*

………………………………………………………………………………
2. [Output statement] A+10 *[2.5 points if correct]*

……………………………………………………………………...

[5 points] Question III: Complete the statements of or describe in natural language the steps for writing a program that asks the user for:
1. an integer number *[1.75 points if correct]*
2. a name *[1.75 points if correct]*

and prints to screen the values of the two variables: the number <space> the name. *[2.5 points if correct]*

[5 points] Question IV: Modify the following program so that instead of printing to screen the value of A+1, it prints the message “value of A to the power of two =”, followed by the value of A*A.

```
[Declare Integer Variable] A
[Input statement] A
[Output statement] A+1
```

C.9. Second Procedural-Knowledge Test for Secondary-Education Quasi-Experiment (First Round)

The following items were included in the second procedural-knowledge test. This measurement took place immediately after the second declarative-knowledge measurement (after five lessons since the beginning of the study) and it took approximately 15-20 minutes to complete. The procedural knowledge of integer, arithmetic expression, string, output-statement, input-statement, variable and assignment-statement was measured.

[10 points] Question I: Write under each group of commands the output produced after its execution:

1. [Declare Integer Variable] Age
   [Assignment statement] Age <- 15
   [Output statement] Age + 10 *[2.5 points if correct]*

....

2. [Declare String Variable] Title
   [Assignment statement] Title <- "2nd School"
   [Output statement] Title *[2.5 points if correct]*

…………………………………………………………………...

3. [Declare Integer Variable] Mark
   [Assignment statement] Mark <- 10
   [Assignment statement] Mark <- 20
   [Output statement] Mark *[2.5 points if correct]*

………………………………………………………………………………………

4. [Declare String Variable] Address
   [Input statement] Address
   [Assignment statement] Address <- "Lefkas 10"
   [Output statement] Address *[2.5 points if correct]*

……………………………………………………………………………...

[5 points] Question II: Complete the statements of or describe in natural language the steps for writing a program with two integer variables A and B, that

1. assigns the value of 10 to variable A *[1.75 points if correct]*
2. asks the user to input a value to variable B and *[1.75 points if correct]*

3. prints to screen the message “A times B equals ” and the result of A*B*. [2.5 points if correct]*

[5 points] Question III: Modify the following program, so that

1. instead of assigning a value to B, the user inputs a value to B *[2.5 points if correct]*
2. it prints to screen the result of A+B, instead of printing the values of A and B *[2.5 points if correct]*

```
[Declare Integer Variable] A
[Declare Integer Variable] B
[Assignment statement] A <- 10
[Assignment statement] B <- 20
[Output statement] A
[Output statement] B
```

C.10. Third Procedural-Knowledge Test for Secondary-Education Quasi-Experiment (First Round)

The following items were included in the third procedural-knowledge test. This measurement took place after the third declarative-knowledge measurement (at the end of the study) and it took approximately 20-25 minutes to complete. The procedural knowledge of integer, arithmetic expression, string, output-statement, input-statement, variable, assignment-statement, iteration-statement, procedure, procedure call, input parameter, logical condition and conditional statement was measured.

[5 points] Question I: Write under each group of commands the output to be produced after its execution:

```
1.  [Declare String Variable] City
    [Assignment statement] City <-  “Lefkas”
    [Output statement] “I live in ”, City [1.25 points if correct]
```

....

```
2.  [Declare Integer Variable] Age
    [Declare String Variable] Name
    [Assignment statement] Age <- 15
    [Assignment statement] Title <-  “I am ”
    [Output statement] Name , Age  [1.25 points if correct]
```

……………………………………………………………………...

```
3.  [Declare Integer Variable] K
    [Iteration statement: K = 1 TO 3]
        [Output statement]  K + 1
    [End Iteration statement] [1.25 points if correct]
```

……………………………………………………………………………………

```
4.  [Declare Integer Variable] Mark
    [Assignment statement] Mark <- 8
    [If statement] (Mark < 10)
        [Output statement] “Failed”
    [Else]
        [Output statement] “Passed”
    [End If statement]   [1.25 points if correct]
```

…………………………………………………………………………………...

[5 points] Question II: Complete the statements of or describe in natural language the steps for writing a program, that

1. requests from the user to input a name *[2.5 points if correct]*
2. prints the name five times to screen*. [2.5 points if correct]*

[5 points] Question III: Modify the following program, so that

1. the logical condition checks if A is greater than 100,instead of checking if it is greater than 0*[2.5 points if correct]*
2. the output messages are “Value greater than 100” and “Value less than 100”, respectively *[2.5 points if correct]*

```
[Declare Integer Variable] A
[Input statement] A
[If statement]  (A>0)
        [Output statement] “Value greater than 0”
[Else]
        [Output statement] “Value less than 0”
[End If statement]
```

[5 points] Question IV: Complete the missing commands in the following program, so that

1. procedure power3 calculates the value of input parameter x to the power of three and prints it to the screen *[2.5 points if correct]*
2. the main procedure uses power3 to calculate the value of variable A to the power of three *[2.5 points if correct]*

```
[Procedure power3 (integer variable x)]
   [Output statement]  …………………………..
[End Procedure]


[Main Procedure]
    [Declare Integer Variable] A
    [Output statement] A
    [Call statement]……………………………….
[End Main Procedure]
```

C.11. Cross-Tabulation Results for the Procedural-Knowledge Tests Used in the Secondary-Education Quasi-Experiment of the First Round

The following tables report the results of cross-tabulation analysis that was performed on the scores that students achieved on items of the procedural-knowledge tests of the first round of data collection. In each table, the first column reports the number of the test item, while the second the number of the subitem. The third and fourth columns report the minimum and maximum scores of each subitem, respectively. The fifth column reports all the different scores that students achieved on each subitem. Finally, the sixth column reports the number of students that achieved the particular score, while the seventh column presents the percentage of students that achieved this score. The results of cross-tabulation analysis on students' scores on the items of the first, second and third procedural-knowledge test are reported in Table C.2, Table C.3 and Table C.4, respectively.

Based on the results of cross-tabulation analysis of the first procedural-knowledge test's scores (Table C.2), the first subitem of the first item seemed to be relatively easy, while the second and third items seemed to be relatively hard. The results reported in Table C.3, revealed that the first second and fourth subitems of the first item seemed to be relatively easy, while the second and third subitems of the second item and the first subitem of the third item seemed to be relatively hard. Finally, the results presented in Table C.4 suggested that the first second and fourth subitems of the first item and the second subitem of the third item seemed to be relatively easy. The second subitem of the second item and the first and second subitems of the fourth item seem to be relatively difficult. All (sub)items that were identified as relatively easy or relatively difficult were modified accordingly in order to become better in discriminating procedural knowledge among quasi-experimental groups in the second round of data collection.

Table C.2.
Results of Cross-Tabulation Analysis on the Scores of the First Procedural-Knowledge Test (First Round).

Item Number	Subitem Number	Minimum Score on Subitem	Maximum Score on Subitem	Score on Subitem	Number of Students Achieving This Score	Percentage of Students Achieving This Score
1	1	0.000	2.500	0.000	18	26.87%
				2.500	49	73.13%
	2	0.000	2.500	0.000	37	55.23%
				2.500	30	44.77%
2	1	0.000	2.500	0.000	39	58.20%
				2.500	28	41.80%
	2	0.000	2.500	0.000	27	40.29%
				2.500	40	59.71%
3	1	0.000	1.250	0.000	52	77.61%
				0.625	1	1.49%
				0.938	1	1.49%
				1.250	13	19.41%
	2	0.000	1.250	0.000	50	74.63%
				0.625	1	1.49%
				0.938	1	1.49%
				1.250	15	22.39%
	3	0.000	2.500	0.000	53	79.10%
				1.250	5	7.46%
				1.875	2	2.99%
				2.500	7	10.45%
4	1	0.000	5.000	0.000	44	65.67%
				1.250	8	11.94%
				2.500	10	14.93%
				5.000	5	7.46%

Table C.3.
Results of Cross-Tabulation Analysis on the Scores of the Second Procedural-Knowledge Test (First Round).

Item Number	Subitem Number	Minimum Score on Subitem	Maximum Score on Subitem	Score on Subitem	Number of Students Achieving This Score	Percentage of Students Achieving This Score
1	1	0.000	2.500	0.000	9	12.86%
				2.500	61	87.14%
	2	0.000	2.500	0.000	8	11.43%
				2.500	62	88.57%
	3	0.000	2.500	0.000	37	52.86%
				2.500	33	47.14%
	4	0.000	2.500	0.000	11	15.71%
				2.500	59	84.29%
2	1	0.000	1.250	0.000	35	50.00%
				1.250	35	50.00%
	2	0.000	1.250	0.000	49	70.00%
				0.625	5	7.14%
				1.250	16	22.86%
	3	0.000	2.500	0.000	46	65.71%
				0.625	1	1.43%
				1.250	9	12.86%
				2.500	14	20.00%
3	1	0.000	2.500	0.000	45	64.29%
				1.250	8	11.43%
				2.500	17	24.29%
	2	0.000	2.500	0.000	38	54.29%
				1.250	3	4.29%
				2.500	29	41.43%

Table C.4.
Results of Cross-Tabulation Analysis on the Scores of the Third Procedural-Knowledge Test (First Round).

Item Number	Subitem Number	Minimum Score on Subitem	Maximum Score on Subitem	Score on Subitem	Number of Students Achieving This Score	Percentage of Students Achieving This Score
1	1	0.000	1.250	0.000	6	8.57%
				1.250	64	91.43%
	2	0.000	1.250	0.000	4	5.71%
				1.250	66	94.29%
	3	0.000	1.250	0.000	34	48.57%
				1.250	35	50.00%
	4	0.000	1.250	0.000	17	24.29%
				1.250	53	75.71%
2	1	0.000	2.500	0.000	39	55.71%
				1.250	5	7.14%
				2.500	26	37.14%
	2	0.000	2.500	0.000	51	72.86%
				0.625	1	1.43%
				1.250	10	14.29%
				1.875	4	5.71%
				2.500	4	5.71%
3	1	0.000	2.500	0.000	27	38.57%
				1.875	1	1.43%
				2.500	42	60.00%
	2	0.000	2.500	0.000	23	32.86%
				1.250	1	1.43%
				1.875	2	2.86%
				2.500	44	62.86%
4	1	0.000	2.500	0.000	43	61.43%
				1.250	4	5.71%
				1.875	1	1.43%
				2.500	22	31.43%
	2	0.000	2.500	0.000	56	80.00%
				1.250	8	11.43%
				2.500	6	8.57%

C.12. First Procedural-Knowledge Test for Secondary-Education Quasi-Experiment (Second Round)

The following items were included in the first procedural-knowledge test. This measurement took place at the start of the fourth lesson since the beginning of the study and it took approximately 15 minutes to complete. The procedural knowledge of integer, arithmetic expression, string, output-statement, input-statement and variable was measured**.**

[5 points] Question I: Write under each command the output produced after its execution:
1. [Output statement] (22-2)/2-3*2 *[2.5 points if correct]*

..
2. [Output statement] “12*3” *[2.5 points if correct]*

……………………………………………………………...

[5 points] Question II: Given that the value of variable A is 10, write under each command the output produced after its execution:
1. [Output statement] “A+10” *[2.5 points if correct]*

………………………………………………………………………………………
2. [Output statement] A+10 *[2.5 points if correct]*

……………………………………………………………………………...

[5 points] Question III:
a. Fill the missing commands in the following program so that it
   1. requests from the user to input an integer number *[1.75 points if correct]*
   2. prints to screen the result of the multiplication of this number by five *[1.75 points if correct]*

   [Declare Integer Variable] Number
   ………………………………
   [Output statement] Number ……………………………

b. Which of the following commands produces the output

The number you entered is [followed by the number entered by the user]. [*2.5 points if correct*]

1. [Output statement] “The number you entered is ”
2. [Input statement] “The number you entered is ”, number
3. [Output statement] number, “The number you entered is ”
4. [Output statement] “The number you entered is ”, number

[5 points] Question IV:

a. Which of the following commands would you modify in order to produce the output:

   value of A to the power of two = (followed by the value of A^2)

   instead of the output:

   A+1

[Declare Integer Variable] A
[Input statement] A
[Output statement] A+1

b. Write the command that produces the following output:

   value of A to the power of two = [followed by the value of A*A]

C.13. Second Procedural-Knowledge Test for Secondary-Education Quasi-Experiment (Second Round)

The following items were included in the second procedural-knowledge test. This measurement took place after the second declarative-knowledge measurement (after five lessons since the beginning of the study) and it took approximately 15-20 minutes to complete. The procedural knowledge of integer, arithmetic expression, string, output-statement, input-statement, variable and assignment-statement was measured.

[10 points] Question I: Write under each group of commands the output produced after its execution:

1. [Declare Integer Variable] Age
   [Assignment statement] Age <- 20
   [Output statement] “Age/2 =” , Age/2 *[2.5 points if correct]*

....

2. [Declare String Variable] Title
   [Assignment statement] Title <- “6th School” [Output Statement] “I go to the ”, Title *[2.5 points if correct]*

………………………………………………………………...

3. [Declare Integer Variable] Mark
   [Assignment statement] Mark <- 10
   [Assignment statement] Mark <- 20
   [Output statement] Mark *[2.5 points if correct]*

………………………………………………………………………………………

4. [Declare String Variable] Address
   [Input statement] Address
   [Assignment statement] Address <- “Patras 10”
   [Output statement] Address, Address *[2.5 points if correct]*

……………………………………………………………………………..

[5 points] Question II: Suppose that a program includes two integer variables, namely P and R,

1. Write (in natural or programming language) the command that assigns value 3 to variable P*[1.25 points if correct]*

   …………………………………………………………

2. Which of the following command requests from the user to input a value to variable R *[1.25 points if correct]*
   a. [Assignment statement] P
   b. [Input statement] R
   c. [Output statement] P
   d. [Assignment statement] R
   e. [Output statement] R
   f. [Input statement] P

3. Complete the following command so that it produces the output:
   P times R equals (followed by the result of P*R). *[2.5 points if correct]*

   [Output statement]…………………………………………………………

[5 points] Question III:

1. a. Which of the following commands should be modified, so that the value of B is given by the user (input) B, instead of being assigned to B. *[1.25 points if correct]*

   b. Write a command that requests from the user to give (input) a value to variable B*[1.25 points if correct]*

…………………………………………………………………………………………….

2. Modify the appropriate command(s) of the following program, so that its execution prints to screen the result of A+B, instead of printing the values of A and B *[2.5 points if correct]*

```
[Declare Integer Variable] A
[Declare Integer Variable] B
[Assignment statement] A <- 10
[Assignment statement] B <- 20
[Output statement] A
[Output statement] B
```

C.14. Third Procedural-Knowledge Test for Secondary-Education Quasi-Experiment (Second Round)

The following items were included in the third procedural-knowledge test. This measurement took place after the third declarative-knowledge measurement (at the end of the study) and it took approximately 20-25 minutes to complete. The procedural knowledge of integer, arithmetic expression, string, output-statement, input-statement, variable, assignment-statement, iteration-statement, procedure, procedure call, input parameter, logical condition and conditional statement was measured.

[5 points] Question I: Write under each group of commands the output produced by its execution:

1.

```
[Declare String Variable] City
[Assignment statement] City <-  "in the city of "
[Output statement] "I live in ", City, "Patras" [1.25 points if correct]
```

....

2.

```
[Declare Integer Variable] Age
[Declare String Variable] Name
[Assignment statement] Age <- 7
[Assignment statement] Title <-  "I am "
[Output statement] Title , Age * 2, "years old"  [1.25 points if correct]
```

…………………………………………………………………...

3.

```
[Declare Integer Variable] K
[Iteration statement: K = 1 TO 3]
      [Output statement]  K + 1
[End Iteration statement] [1.25 points if correct]
```

………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………

4.

```
[Declare Integer Variable] Mark
[Assignment statement] Mark <- 8
[If statement] (Mark + 5 < 10)
      [Output statement] "Failed"
```

```
[Else]
    [Output statement] “Passed”
[End If statement]   [1.25 points if correct]
```

……………………………………………………………………………………..

[5 points] Question II:

a. Complete the missing commands in the following program so that
   1. it requests from the user to give (input) a name to string variable NAME *[2.5 points if correct]*
   2. it prints to screen the value of NAME five times *[2.5 points if correct]*

```
[Declare String Variable] NAME
………………………………
[Iteration Statement]…………………….
     [Output Statement]  ……………………………
[End of Iteration Statement]
```

[5 points] Question III:

1. Modify the following program, so that the logical condition checks whether the value of variable A is less than 100, instead of greater than 0*[2.5 points if correct]*
2. Given the new logical condition, modify the appropriate commands so that the output of program’s execution is **Value greater than 100** (when A>=100) and **Value lesser than 100** (when A<100) *[2.5 points if correct]*

```
[Declare Integer Variable] A
[Input statement] A
[If statement]  (A>0)
    [Output statement] “Value greater than 0”
[Else]
    [Output statement] “Value lesser than 0”
[End If Statement]
```

[5 points] Question IV:

1. Complete the missing commands in the following procedure, so that it calculates the value of input parameter x to the power of three and prints it to screen *[2.5 points if correct]*

```
[Procedure power3 (integer variable X)]
  [Output statement]  …………………………..
[End Procedure]
```

2. Which of the following commands should be used in the main procedure, so that it calls procedure power3 to calculate the value of variable A to the power of three *[2.5 points if correct]*

```
[Main Procedure]
    [Declare Integer Variable] A
    [Input statement] A
    ……………………………….
[End Main Procedure]
```

a. [Call statement] power3 ()
b. [Output statement] A * A * A
c. [Call statement] power3 (A)
d. [Input statement] A
e. [Call statement] power3 (X)

C.15. Practical Test for Secondary-Education Quasi-Experiment (Second Round)

The practical test consisted of the following items and was administered only in the second round of data collection. Students were asked to create the following three programs with the help of the programming tool they had been using during the study. The practical test took approximately 30 minutes to complete.

Program 1

Create a program using Koios / Glossomatheia / MicroworldsPro that requests from the user to give (input) a name to the string variable NAME and an integer number to integer variable TIMES. The program prints to the screen the value of NAME as many times as the value of TIMES.

Program 2

Use Koios / Glossomatheia / MicroworldsPro to create a program with a procedure named OPP. Procedure OPP takes an integer input parameter X and prints to the screen the opposite number of X, namely -X. The main procedure requests from the user to give (input) an integer number to the integer variable NUMBER, and with the use of OPP prints to screen the opposite value of NUMBER.

Note: MicroworldsPro does not support the expression -X, to calculate the opposite of X. Therefore, the following hint was given to MicroworldsPro users only.

Hint: Calculate –X using the formula: -X = -1 * X.

Program 3

Create a program using Koios / Glossomatheia / MicroworldsPro that requests from the user to give (input) an integer number to the integer variable GRADE. If the value of GRADE is less than 10, then the string FAIL will appear on the screen, otherwise the string PASS will appear on the screen.

C.16. Marking Scheme for Practical Test

The following scheme was used in order to objectively assess the programs produced by students during the practical test. Each produced program received specific points for including the appropriate programming element(s) or presenting particular programming properties. The mark of each program was the sum of the points received. The maximum number of points awarded for each programming element/property for the three programs is presented in Tables C.5, C.6 and C.7.

Table C.5.
Marking Scheme for the First Program Of Practical Test

Elements and Properties of Program 1	Points
Declaration of variables.	0.75
Correct syntax of declaration of variables.	0.25
Use of input statement.	1.25
Correct syntax of input statement.	0.50
Use of output statement.	1.25
Correct syntax of output statement.	0.50
Use of iteration statement.	1.75
Correct syntax of iteration statement.	0.75
Correct order of commands.	1.00
The program runs.	1.00
The program runs correctly.	1.00
TOTAL	10.00

Table C.6.
Marking Scheme for the Second Program Of Practical Test

Elements and Properties of Program 2	Points
Declaration of variables.	0.750
Correct syntax of declaration of variables.	0.250
Use of input statement.	0.750
Correct syntax of input statement.	0.250
Use of output statement.	0.750
Correct syntax of output statement.	0.250
Creation of procedure and input parameter.	1.375
Correct syntax of procedure and input parameter creation.	0.625
Calling procedure.	1.375
Correct syntax of procedure calling.	0.625
Correct order of commands.	1.000
The program runs.	1.000
The program runs correctly.	1.000
TOTAL	10.000

Table C.7.
Marking Scheme for the Third Program Of Practical Test

Elements and Properties of Program 3	Points
Declaration of variables.	0.75
Correct syntax of declaration of variables.	0.25
Use of input statement.	0.75
Correct syntax of input statement.	0.25
Use of output statement.	0.75
Correct syntax of output statement.	0.25
Use of conditional statement/formation of the appropriate condition.	1.75
Correct syntax of conditional statement/condition.	0.75
Use of else statement.	1.00
Correct syntax of else statement.	0.50
Correct order of commands.	1.00
The program runs.	1.00
The program runs correctly.	1.00
TOTAL	10.00