Exploring student perceptions about the use of visual programming environments, their relation to student learning styles and their impact on student motivation in undergraduate introductory programming modules

Maira Kotsovoulou (BSc, MSc)

July 2019

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Department of Educational Research,
Lancaster University, UK.

Exploring student perceptions about the use of visual programming environments, their relation to student learning styles and their impact on student motivation in undergraduate introductory programming modules

Maira Kotsovoulou (BSc, MSc)

This thesis results entirely from my own work and has not been offered previously for any other degree or diploma.

The word count is 57,743 excluding references.

Signature ....

Maira Kotsovoulou (BSc, MSc)

Exploring student perceptions about the use of visual programming environments, their relation to student learning styles and their impact on student motivation in undergraduate introductory programming modules

Doctor of Philosophy, July 2019

Abstract

My research aims to explore how students perceive the usability and enjoyment of visual/block-based programming environments (VPEs), to what extent their learning styles relate to these perceptions and finally to what extent these tools facilitate student understanding of basic programming constructs and impact their motivation to learn programming.

My overall methodological approach is a case study that explores the nature of potential benefits to using a VPE in an introductory programming module, within the specific context of an English-speaking institution of higher learning in Southern Europe. Part 1 of this research is a pilot study, which uses participatory action research as a methodological practice to identify which visual programming environment will be selected for the main study. Part 2 uses an evaluative methodological practice within the case, aimed at addressing the research questions. Data collection is performed using mixed methods. For the quantitative part, 92 participants provided their feedback using a questionnaire, including 3 main sections: a) an adaptation of the Technology Acceptance Model (Davis, 1985); b) an adaptation of the Motivated Strategies for Learning Questionnaire (MSLQ) (Pintrich & de Groot, 1990b) and the Science Motivation Questionnaire (SMQ-II) (Glynn, *et al.*, 2009); and c) the Index of Learning Styles (Felder & Soloman, 1993). For the qualitative part, feedback was collected both by interviewing students and compiling field notes during class observations. Descriptive statistics, t-tests and Spearman correlations were used to analyse the quantitative data, while the constant comparative method was used to generate the categories, whose relationships emerged from the coding process of the qualitative data.

Results from Part 1 revealed a student preference for Scratch over the other three visual programming environments used in the experiment. Findings from Part 2 suggest that students found Scratch to be easy, useful, enjoyable and engaging, but only within the scope and purpose of the module. On the other hand, students demonstrating strong intrinsic motivation to learn programming and high levels of self-efficacy did not perceive Scratch to be as useful as other students did. Results also indicate that a relationship exists between the acceptance of a visual programming environment and students' learning style preferences; Scratch was found more useful and enjoyable by those reporting visual and sequential learning approaches. Furthermore, overall student performance and pass-fail rates showed considerable improvement following the introduction of Scratch.

Contents

Abstract .... *i*

Contents .... *iii*

Acknowledgements .... *vi*

Dedication .... *vii*

Publications Derived from Work on the Doctoral Programme .... *viii*

List of Abbreviations .... *ix*

List of Figures .... *x*

List of Tables .... *xii*

Chapter 1 Introduction ***1***

1.1 Introduction to the Study **1**

1.2 Research Questions **3**

1.3 Contextual Information **4**

1.4 Thesis Structure **8**

Chapter 2 Computer Programming ***9***

2.1 What is Computer Programming **9**

2.2 Programming Paradigms and Programming Languages **10**

2.3 Types of Programming Environments **14**

2.3.1 A Text-Editor and Command-Line Compiler 16
2.3.2 Program Visualisation Environment 16
2.3.3 Integrated Development Environment (IDE) 19
2.3.4 Visual Programming Environments 20

2.4 Conclusion **31**

Chapter 3 The Theoretical Framework ***34***

3.1 Educational Theories **34**

3.2 Learning Approaches, Learning Styles and Assessment Tools **42**

3.3 Motivation and Self-Determination **51**

3.4 Conclusion **57**

Chapter 4 Teaching and Learning Computer Programming ***60***

4.1 Cognitive Aspects of Programming **60**

4.2 Troublesome Programming Constructs and Skills **68**

4.3 A Short History of Educational Programming Environments **76**

4.4 A Classification of Educational Programming Environments **81**

4.5 Using Visualisations to Teach Programming 85
4.6 Related Research on Greenfoot, Alice, AppInventor and Scratch 90
4.7 Conclusion 93
Chapter 5 The Pilot Study 94
5.1 Purpose 94
5.2 Participatory Action Research 94
5.2.1 The Survey Tool 98
5.2.2 Validity and Reliability for the Pilot Study Survey Tool 99
5.2.3 Action Research Cycle 1 (Greenfoot) 103
5.2.4 Action Research Cycle 2 (Alice) 105
5.2.5 Action Research Cycles 3 and 4 (Workshops on AppInventor and Scratch) 107
5.2.6 Action Research Cycle 3 (AppInventor) 107
5.2.7 Action Research Cycle 4 (Scratch) 109
5.3 Action Research Findings and Discussion 110
5.4 Conclusion 117
Chapter 6 Research Design and Methodology 120
6.1 Purpose 120
6.2 Case Study and Data Collection Approaches 120
6.3 Pedagogic Design: Teacher's Role and Students' Activity 128
6.4 Development of the Questionnaire Survey Tool 130
6.4.1 Section 0 – Participant Information Sheet and Consent Form 131
6.4.2 Section 1 - Participant Demographic Information 132
6.4.3 Section 2 - Overall Evaluation and Acceptance of Scratch 132
6.4.4 Section 3 - Perceived Ease of Use and Perceived Usefulness 134
6.4.5 Section 4 - Motivated Strategies for Learning 137
6.4.6 Section 5 - Index of Learning Styles 145
6.5 Validity and Reliability 146
6.5.1 Validity 146
6.5.2 Validity Issues Addressed in this Study 147
6.5.3 Reliability 152
6.5.4 Reliability Issues Addressed in this Study 152
6.6 Qualitative Data Collection - Interviews 153
6.6.1 Interview Protocol 153
6.6.2 Interview Questions for Students 153
6.7 Ethical Framework 154
6.8 Conclusion 155
Chapter 7 Data Analysis and Findings 156
7.1 Introduction to Data Analysis 156
7.2 Data Gathering and Demographics 157
7.3 Analysis of Student Grades 160

7.4 Results from Student Surveys ... **165**
7.4.1 Student Acceptance of Scratch (TAM) Analysis ... 165
7.4.2 Student Index of Learning Styles (ILS) Analysis ... 169
7.4.3 Student Motivation (MSLQ) ... 173
7.5 Results from the Analysis of Interview Data and Class Observations – Qualitative Feedback ... **176**
7.6 Conclusion ... **183**
Chapter 8 Conclusions ... ***185***
8.1 Contribution of this Study to the Research Literature ... **185**
8.2 Limitations of the Study ... **188**
8.3 Recommended Areas for Future Research ... **189**
8.4 A Final Reflection ... **190**
References ... ***191***
Appendix One – Main Survey Instrument ... ***212***
Section 1 - Demographic Information ... 212
Section 2 - Introduction to Programming - General Questions ... 212
Section 3a - Overall Evaluation and Acceptance for Scratch ... 213
Section 3b - Technology Acceptance Model ... 214
Section 4 - Motivated Strategies for Learning Questionnaire ... 215
Section 5 - Index of Learning Styles (ILS) Learning Style Questionnaire ... 217
Appendix Two – Selection of Questions for the Motivation section of the Main Survey ... ***222***
Appendix Three – Qualitative Analysis ... ***226***
Analysis of Interviews ... **226**
Sample Interview Transcript – Participant 1 ... **228**
Sample Interview Transcript – Participant 4 ... **230**
Summary from Class Observation Notes ... **232**
Appendix Four – Ethics Approval Forms ... ***234***
Lancaster University ... **234**
College XYZ ... **235**
Appendix Five – Pilot Study's Programming Activities ... ***236***
Greenfoot – Guided Activity ("Creating Java Programs with Greenfoot") ... **236**
Alice – Guided Activity ... **237**
APP Inventor – Workshop Activities ... **238**
Scratch – Workshop Activities ... **240**

Acknowledgements

I would like to thank first of all my husband Steve and my two children Nikolas and Chris, who have shown their understanding, patience, love and support during this long journey. I understand that I have not always been there for them due to the vast amount of work required for this research study.

I would also like to thank my brother Nassos, who has always been there for me, encouraging me to continue working when I was about to quit. I will never forget the letter he wrote me a few years ago, reminding me of all the reasons I started this journey…

I am very grateful to the administration of College XYZ, for their financial assistance and their overall support towards the completion of this degree.

I would also like to thank my supervisor, Dr. Don Passey, for his support, guidance and always valuable and timely feedback.

Dedication

This thesis is dedicated to the memory of my parents, Xanthi and Christos Kotsovoulos, who showed me that with hard work anything is possible. They encouraged me to pursue this degree and would have been very proud of this accomplishment.

Publications Derived from Work on the Doctoral Programme

Kotsovoulou, M., Stefanou V. and Makri, D. (2017). Utilization of learning resources by undergraduate-level students in computer programming courses: An exploratory study. *WSEAS Transactions on Advances in Engineering Education, 14*(1), 1–11.

Kotsovoulou, M., & Stefanou, V. (2016). Student perceptions on the effectiveness of collaborative problem-based learning using online pair programming tools. In Strouhal J. (Ed.) *Proceedings of the 12th International Conference on Educational Technologies - EDUTE'16* (pp. 32–39). Barcelona, Spain: WSEAS Press.

Kotsovoulou, M. (2013). Collaborative tagging of Java learning resources: Bridging the gap between teachers and students. *International Journal of Knowledge Society Research 4*(1), 12–29. Web.

Stefanou, V., and Kotsovoulou M. (2016) Use of PowerPoint in the classroom: A participatory research project. *International Journal of Knowledge Society Research 7*(4), 38–50. Web.

Stefanou, V., Kotsovoulou, M., & Makri, D. (2018). Using E-assessment software to support formative assessment: a Phenomenographic Study of Instructors' Experiences. In Chova L, Martinez A., & Torres I. (Eds.) *Proceedings of the 12th International Technology, Education and Development Conference - INTED 2018* (pp. 1066–1075). Valencia, Spain: IATED.

List of Abbreviations

CT	Computational Thinking
CFA	Confirmatory Factor Analysis
ICT	Information and Communications Technology
IDE	Integrated Development Environment
ILE	Initial Learning Environment
IT	Information Technology
MSLQ	Motivated Strategies for Learning Questionnaire
OOP	Object Oriented Programming
PPE	Pedagogical Programming Environment
TAM	Technology Acceptance Model
UML	Unified Meta Language
VPE	Visual Programming Environment

List of Figures

Figure 1.1: College failure rate comparison in two Java programming courses (2012 – 2017) ... 2
Figure 1.2: Teaching methodology ... 7
Figure 1.3: Learning resources ... 7
Figure 2.1: Programming languages popularity tag cloud ... 12
Figure 2.2: Programming languages used in Greek schools ... 13
Figure 2.3: A text-editor and a command line compiler ... 16
Figure 2.4: BlueJ class inspection feature and the text-based code editor ... 17
Figure 2.5: Jeliot programming environment - Theatre mode ... 18
Figure 2.6: Jeliot programming environment - Call tree mode ... 19
Figure 2.7: Eclipse IDE ... 20
Figure 2.8: Alice programming environment ... 23
Figure 2.9: Greenfoot's program design editor ... 24
Figure 2.10: Greenfoot's text-based Java Editor ... 24
Figure 2.11: Greenfoot's frame-based editor ... 25
Figure 2.12: LogoBlocks ... 26
Figure 2.13: EToys ... 26
Figure 2.14: Scratch stage and sprites ... 27
Figure 2.15: Image in Scratch ... 27
Figure 2.16: Sound in Scratch ... 27
Figure 2.17: Scripts in Scratch ... 27
Figure 2.18: Scratch development area ... 28
Figure 2.19: AppInventor: Program design ... 30
Figure 2.20: AppInventor: Block-based code editor ... 31
Figure 3.1: The experiential learning cycle ... 47
Figure 3.2: The nine learning styles in the KLSI 4.0 ... 48
Figure 3.3: The four scales/dimensions of the Felder-Silverman Model and their respective learning style continuum ... 49
Figure 3.4: Differences and similarities between SMQ-II and MSLQ ... 57
Figure 3.5: Conceptual research framework ... 59
Figure 4.1: Programming concepts rated by 105 students and 34 professors ... 72
Figure 4.2: Educator perceptions about VPEs ... 76
Figure 4.3: Example of a BASIC program source code and runtime. ... 78
Figure 4.4: Logo programming environment with a virtual turtle ... 78
Figure 4.5: MicroworldsEX ... 79
Figure 4.6: a Lego car construction controlled by Logo programming language ... 79
Figure 4.7: Scratch active users ... 80
Figure 4.8: Programming development environment classification ... 85
Figure 4.9: Traditional chalkboard visualisation ... 85
Figure 4.10: Forms of software visualisation (Sorva *et al.*, 2013) ... 86
Figure 4.11: BlueJ variable inspection feature and the text-based code editor 88

Figure 4.12: Sample algorithm execution in VisuAlgo 89
Figure 4.13: Scratch visual programming code editor.................................. 89
Figure 5.1: Action research cycles.. 96
Figure 5.2: Adaption of the Technology Acceptance Model 99
Figure 5.3: Percentage of students who submitted their project per VPE...... 111
Figure 5.4: Mean score per question per VPE.. 112
Figure 5.5: Comparison of distributions: Student rating for each VPE 113
Figure 5.6: Participant recommendations for the adoption of each tool 118
Figure 6.1: Case study research design 127
Figure 6.2: TAM2 extended to include enjoyment and output quality 133
Figure 6.3: TAM of Scratch 137
Figure 6.4: Intrinsic motivation scores 141
Figure 6.5: Self-efficacy scores.. 142
Figure 6.6: Self-determination scores 143
Figure 6.7: Extrinsic Motivation (Career and grade) scores 144
Figure 7.1: Participant age distribution in year ranges 158
Figure 7.2: Participant gender distribution 159
Figure 7.3: Participant distribution of majors 159
Figure 7.4: Participant perceived current computer programming level of expertise 160
Figure 7.5: Mean student grades per semester from 2013 – 2018 161
Figure 7.6: Grade comparison before and after the use of Scratch............... 162
Figure 7.7: Pass/fail rates 163
Figure 7.8: Mean coursework scores.. 164
Figure 7.9: NVivo coding of Scratch advantages as perceived by 12 students 178
Figure 7.10: NVivo coding of Scratch disadvantages as perceived by 12 students 178
Figure 7.11:Scratch "hidden" features.. 180
Figure 8.1: Hierarchy chart of nodes in the advantages theme 226
Figure 8.2: Eat the barel 236
Figure 8.3: Race Game... 237
Figure 8.4: Magic-8 Ball... 239
Figure 8.5: MoleMash 239
Figure 8.6: IP packet switch... 240
Figure 8.7: "Guess the Flag" game splash screen 241

List of Tables

Table 2.1: Popularity of programming language index 12
Table 2.2: Comparison of the main characteristics of Greenfoot, Alice, AppInventor, Scratch 32
Table 4.1: Java programming: Difficult concepts – student perceptions 69
Table 4.2: Java programming: Difficult concepts – professor perceptions..... 70
Table 4.3: Java Programming: Ability to understand and write code – student perceptions 73
Table 5.1: Grading rubric for the formative assessment used in all action research cycles 97
Table 5.2: CFA - Action Research Survey 100
Table 5.3: Cronbach's alpha – Action Research Survey 101
Table 5.4: AVE and CR calculations for Motivation Scale 101
Table 5.5: AVE and CR values for all scales 102
Table 5.6: Correlation matrix for motivation scale components..... 102
Table 5.7: Correlation coefficients for enjoyment and usefulness scales 103
Table 5.8: Student evaluation of Greenfoot programming environment..... 105
Table 5.9: Student evaluation of Alice programming environment..... 106
Table 5.10: Student evaluation of AppInventor programming environment. 108
Table 5.11: Student evaluation of Scratch programming environment..... 109
Table 5.12: Action research study – Demographics: Gender 110
Table 5.13: Action research study – Demographics: Majors 110
Table 5.14: One-way ANOVA test for the equality of medians across VPEs .. 111
Table 5.15: Shapiro-Wilk’s test of normality 114
Table 5.16: Levene’s test of homogeneity of variances..... 115
Table 5.17: One-way ANOVA test for the equality of means..... 115
Table 5.18: Post-hoc multiple comparisons between VPEs - Tukey HSD test 116
Table 6.1: Case study designs/themes 122
Table 6.2: Standardisation matrix - Card Sorting..... 134
Table 6.3: Cronbach alpha for TAM..... 147
Table 6.4: CFA for TAM 148
Table 6.5: AVE and CR values for TAM..... 148
Table 6.6: Cronbach alpha’s for Motivation Scales..... 150
Table 6.7: CFA for Motivation scales..... 150
Table 6.8: AVE and CR values for motivation scales 151
Table 7.1: Number of participants across the years of the study 158
Table 7.2: Descriptive statistics of Scratch acceptance 166
Table 7.3: Descriptive statistics of Scratch acceptance per professor 167
Table 7.4: One-way ANOVA - Student acceptance of Scratch between professors..... 168
Table 7.5: Spearman’s rho correlations between Scratch acceptance and student grades 168
Table 7.6: Student dominant learning styles in the 4 dimensions..... 170

Table 7.7: Correlations between student learning styles and their perceptions about Scratch 172
Table 7.8: Motivational Component Mean Scores 174
Table 7.9: Kruskal Wallis test for motivational scale distribution across students taught by different professors 174
Table 7.10: Spearman's rho correlation between student motivation to learning programming and acceptance of Scratch 175
Table 7.11: Spearman’s rho correlation between student motivation to learning programming and performance 176
Table 7.12: Summary of notes from class observations 183

Chapter 1 Introduction

1.1 Introduction to the Study

As an information technology (IT) educator for over 20 years, with an emphasis on teaching programming at all levels (primary, higher and vocational), one of the many challenges I face is to make the student learning experience as meaningful, interesting and engaging as possible, while also preparing graduates for the real-world software development environment. I am constantly concerned with improving my teaching, utilising and testing various techniques and approaches that could provide students with different ways of experiencing computer programming. The diversity of these experiences could possibly make more students understand how they can efficiently write computer code, appreciate the challenges, and positively relate to the process.

While the worldwide demand for computer programmers has increased and is expected to increase even more in the following years (up to 24% from 2016 - 2026, (Bureau of Labor Statistics, 2019)), anecdotal evidence on teaching and learning computer programming, especially at the introductory level, shows that many students fail introductory courses (Bennedsen & Caspersen, 2007; Watson & Li, 2014). Based on my 20-year teaching experience I have evidence to support the same view. Statistics collected from all introductory programming modules from all courses at the English-speaking institution of higher learning in Southern Europe where this research takes place, shows an overall failure rate of 52%. Figure 1.1 shows a comparison of the failure rates between introduction to programming and object-oriented programming modules, which are the first and the second required programming modules in the progression list for the software development track of the information technology (IT) major.

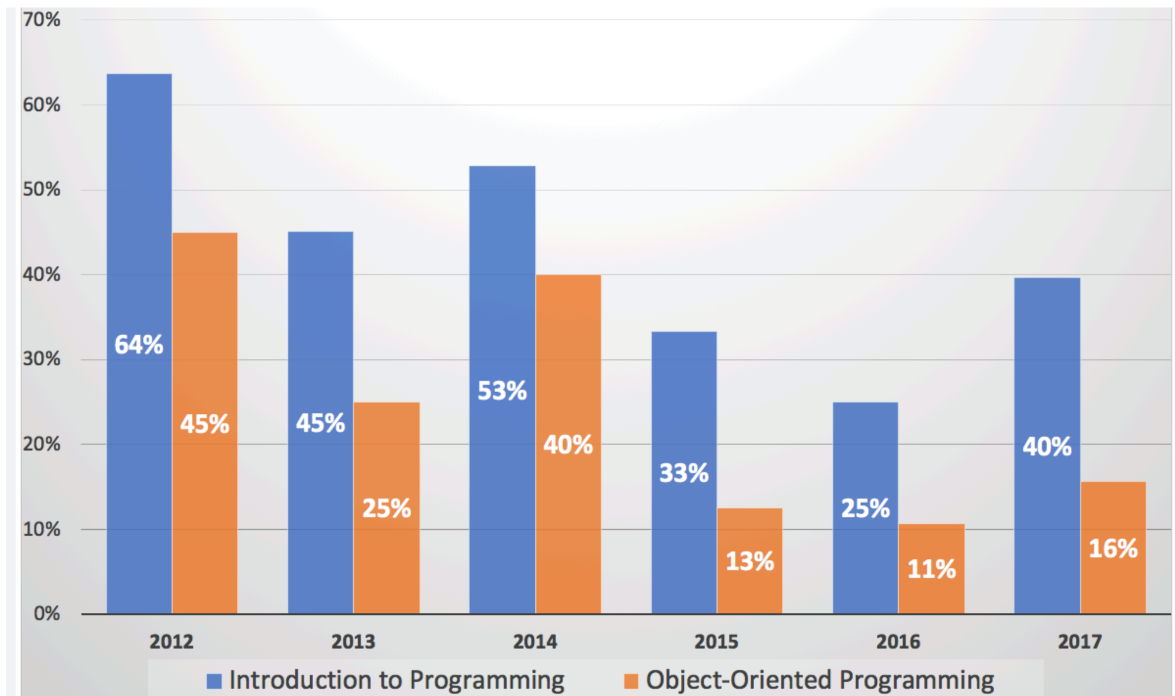


Figure 1.1: College failure rate comparison in two Java programming courses (2012 – 2017)

Research also indicates that many students perceive computer-programming concepts as being overly difficult to understand (Eckerdal *et al.*, 2005; Eckerdal, 2006; Giraffa *et al.*, 2014). Some of the identified reasons for this failure include students' lack of problem-solving skills (Lahtinen *et al.*, 2005; Ismail *et al.*, 2010;) and inability to construct mental models of "abstract" programming concepts (Ma *et al.*, 2009).

As new hardware devices and programming methodologies evolve, affecting the way novice programmers might understand and visualise computer programming, additional research is warranted in order to assess the impact of these technologies on student learning. In recent years, a number of visual programming tools, such as Scratch, Alice, Greenfoot and App Inventor, have been used to introduce programming to students. Although each was created for use by different age groups (CS1/pre-CS1 for Alice, 8-16 year olds for Scratch, and 14+ year olds for Greenfoot (Utting *et al.*, 2010) they all share a common principle; they use visualisations and fixed blocks of code as a means to convey fundamental programming and object oriented programming concepts to learners.

My beliefs, as far as teaching and learning programming are concerned, have been influenced by Bowden and Marton who claim: *"Variation must be present in the learning environment…"* (2003, p.11). I agree that learners should be exposed to a variety of experiences that could potentially allow them to change the "*way of seeing*" several aspects of computer programming, focus on the *"critical dimensions of these experiences"*, and relate intangible concepts to more tangible ones.

My foremost concern is to make my lectures meaningful, interesting and up-to-date. In this study, I wish to explore the extent to which usage of innovative instructional approaches impacts student motivation and performance. Although such impact can be qualified in multiple forms, I am mostly interested in a) performance in hands-on programming assignments and theoretical assessments; b) enjoyment; c) level of engagement; d) perceptions of programming difficulty; and e) perceptions of value of the new technology used in class.

Teaching computer programming is more than teaching a programming language. Consequently, in this research, I will focus on understanding the processes of learning and teaching programming by exploring other disciplines including psychology, learning theories and knowledge representation, learning approaches and motivation along with computer science. I aim to improve the teaching and learning process by providing students with the most effective learning environment and experience.

1.2 Research Questions

In the context of the Introduction to Programming module in this English-speaking institution of higher learning in Southern Europe, Scratch software was used to enable students to undertake visual programming.

My research questions in this context are:

1) How do visual programming environments affect students' performance in the course (assessment and final grades)?
2) How do students perceive visual programming environments?
   a) How do they perceive enjoyability, ease of use, usability and usefulness?
   b) How do they relate these qualities to their achievement of the module's learning objectives (output quality)?
3) How does students' motivation for learning programming relate to their perceptions about visual programming environments?
4) How do students' learning styles relate to their perceived enjoyment, ease of use, usability and usefulness of visual programming environments?

1.3 Contextual Information

The current research takes place at an English-speaking institution of higher learning in Southern Europe, which will be referred to as college XYZ. XYZ college was founded in 1875 in Smyrna, Asia Minor, by missionaries from Boston, Massachusetts and has been accredited by the New England Association of Schools and Colleges (NEASC) since 1981, which is the oldest and largest accrediting organisation in the United States.

In 2010, XYZ college partnered with the Open University of the United Kingdom (UK), which is the largest programme validation institution in Europe and currently offers twenty-eight undergraduate programmes validated by the Open University, UK.

The Information Technology (IT) major is fairly new at XYZ college. It was created in 2010 and the first IT major students are currently employed in the business sector. The major went through OU revalidation in 2016, where all module learning outcomes were revised and updated in order to reflect latest trends in technology and to conform to the Quality Assurance Agency for Higher Education Computing Standards. As of spring semester 2016, Scratch was used to introduce programming to students during the first two weeks of the module.

The “Introduction to Programming” module introduces students to structured and basic object-oriented computer programming, with an emphasis on problem-solving strategies. The course requires no prior programming experience and is the first programming prerequisite for students majoring in “Information Technology”. Emphasis is given on problem analysis, algorithm design, coding and testing using the Java programming language. The module has five learning objectives, for which students are assessed on both a theoretical and practical level.

According to the module’s syllabus, upon successful completion of the course, students should be able to:

1) Demonstrate understanding of fundamental programming concepts and solve basic problems using fundamental programming constructs.
2) Create an algorithmic solution to a programming problem using pseudo-code.
3) Demonstrate understanding of how to trace source code and correctly predict the results.
4) Make use of basic data structures and search/sort algorithms to design, implement, test, and debug programs.

5) Develop well-documented, structured and maintainable programs.

The “Introduction to Programming” module’s method of teaching and learning includes 3-hours of lecture per week and 2-hours of laboratory practical sessions. Blackboard (TM) is used as the course management system and supports class communication through lecture notes, web resources, assignment instructions, and timely announcements, user forums for troubleshooting, formative quizzes and online submission of assignments.

The “Introduction to Programming” has two formal assessments: a mid-term examination that counts for 40% and a coursework project that counts for 60% of the final module grade. The coursework project contains 3 parts: Part A evaluates student understanding of fundamental programming concepts and how they can solve basic problems using fundamental programming constructs, in Scratch. Part B tests their ability to create an algorithmic solution to a programming problem using pseudo-code. Part C tests their ability to write a well-documented, structured and maintainable Java program that utilises data structures and searching/sorting algorithms.

The module covers the following content areas:

1) Introduction to algorithms and block-based programming
2) Learning to code using Scratch
    a) Variables, arithmetic, operators
    b) User input
    c) Selection and iteration
    d) Count controlled loops/condition-controlled loops
    e) Complex conditions
    f) Procedures (custom blocks)
    g) Introduction to event-driven programming concepts and multitasking
    h) Sprite cloning (object instantiation)
    i) Creating a game
        i) Requirements specification
        ii) Interface design
        iii) Code design
        iv) Implementation

v) Testing

3) Overview of computers and programming languages, numeric systems
4) Introduction to Java programming language, Software Development Kit (SDK), Java Development Kit (JDK), Java Virtual Machine (JVM) and command line tools
   a) Variables, primitive datatypes, arithmetic, operators
   b) Strings
   c) Input/output
   d) Tracing programs and debugging
   e) Relational operators, selection
   f) Complex conditions
   g) Iteration
   h) Count controlled loops
   i) Condition controlled loops
   j) User defined methods
   k) Arrays
   l) Command-line arguments
   m) Basic searching and sorting algorithms
   n) Exception handling
5) Introduction to Object-Oriented Programming (OOP) concepts

Figure 1.12 and Figure 1.3 depict the teaching methodology and the learning resources of the module.

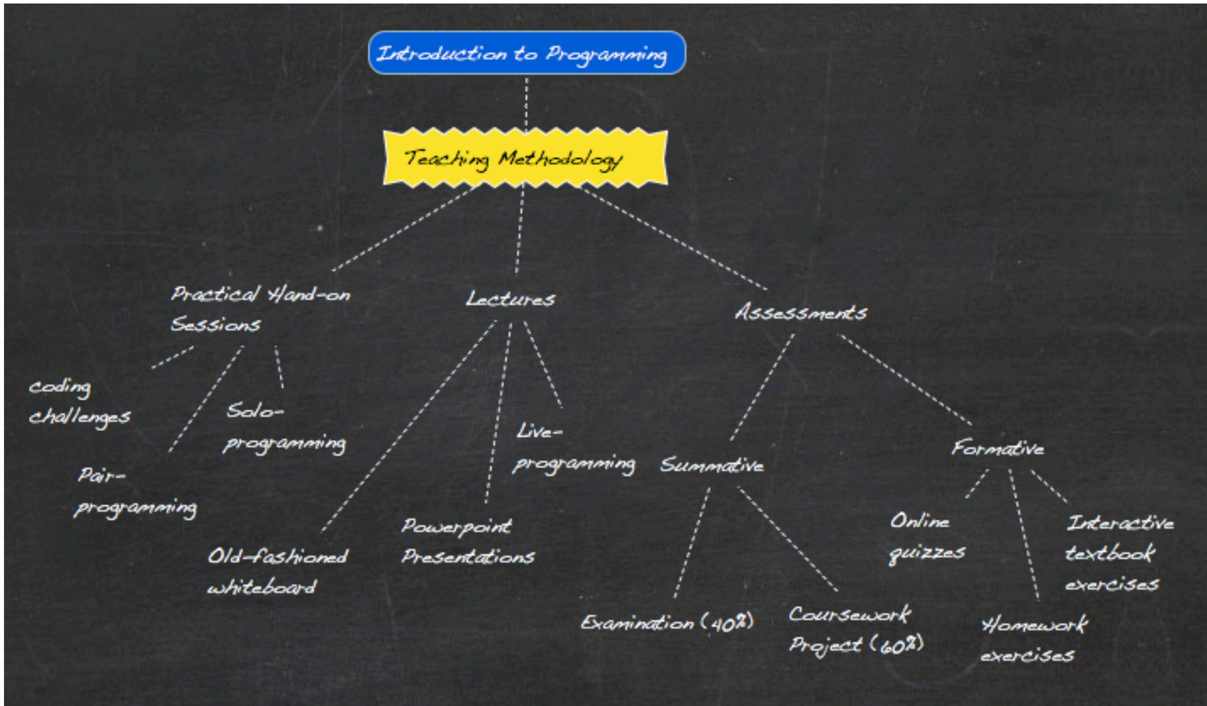


Figure 1.2: Teaching methodology

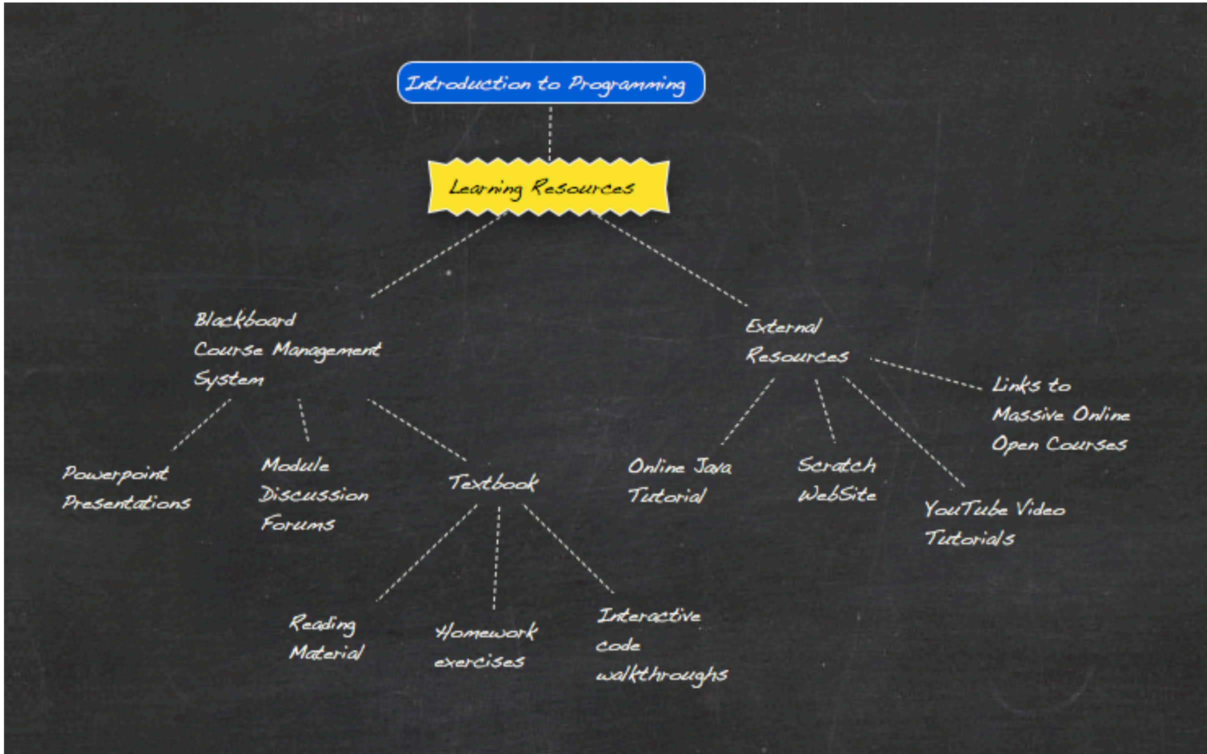


Figure 1.3: Learning resources

This module is a requirement for all students majoring in IT and is offered four times a year. Each occurrence of the module has a registration limit of 18-20 students. It has been observed, though, that some students who initially choose the "Software Development" pathway of the major, tend to shift to either "Network Technologies"

or “Digital Media” which are perceived by students as being easier. The basis of this statement is grounded on input obtained from informal conversations with students during the past seven years.

1.4 Thesis Structure

This thesis begins with an overview of computer programming and a presentation of programming environments, providing a relevant context for the reader and preparing the ground for justifying choice of tools used in this research project. To address the research questions, the thesis then provides an overview of underlying conceptual frameworks from relevant learning theories and approaches, from motivation theory, as well as research on measurement instruments, assessment tools and related methodologies.

A literature review then follows that explores cognitive aspects of computer programming, difficulties imposed on novice learners, classification of programming environments, and the rationale behind the need for educational and visual programming environments (VPEs). The thesis follows with a review of research related to teaching novices how to program using Scratch, App Inventor, Alice and Greenfoot, and the effects these environments have on student motivation. Research findings indicate a positive impact on student motivation for all four VPEs mentioned above. Consequently, a two-year participatory action research study (referred to henceforth as the pilot study) was conducted with the aim of identifying the most appropriate VPE. Participants assessed Scratch to be the most suitable tool.

The thesis advances with an evaluation of a case study using mixed data collection methodologies, and a justification as to why a combined approach was considered appropriate, followed by a presentation of the overall research design. The steps involved in the development and validation of the assessment instrument, which was created by adapting 2 different tools (MSLQ, TAM) and incorporating the Index of Learning Styles Questionnaire was an important part of the study.

Finally, a description of the collected data, their analysis and presentation of the results complements the findings of the pilot study, before leading to the final chapter, where conclusions of the study, its contributions to the literature, and its limitations are discussed.

Chapter 2 Computer Programming

With an eye to utilising and testing various visual programming environments that could provide students with different ways of experiencing computer programming and potentially increase their motivation to learn, I begin by introducing the concepts of computer programming, different programming paradigms, as well as types of programming environments that exist in the market today. The purpose of this introduction is to provide a relevant context, as well as preparing the ground for justifying the choice of tools used in this research project.

2.1 What is Computer Programming

"Programming will help you learn the importance of clarity of expression" (Madan, 2003, p.97)

Pea and Kurland (1983) defined the core sense of computer programming as "that set of activities involved in developing a reusable product consisting of a series of written instructions that make a computer accomplish some task" (Pea & Kurland, 1983, p.5).

In other words, computer programming is a process that enables people to write a set of directives to instruct the computer how to perform a specific task. A computer program is like a very precise recipe. It requires a list of specific ingredients and an exact set of ordered steps for the machine to follow in order to perform something. The recipe should produce exactly the same result (output) each time the steps are executed using the same ingredients (inputs).

In his work, Papert (1980) argues that a profound understanding of computer programming can help students form "new relationships" with knowledge and receive educational benefits in diverse learning domains: "computers can be carriers of powerful ideas and of the seeds of cultural change, how they can help people form new relationships with knowledge that cut across the traditional lines separating humanities from sciences and knowledge of the self from both of these" (Papert, 1980, p.4).

The computer program is written in a programming language. There are numerous programming languages which can be used to program a computer ranging from low to high level. The lower the level of the programming language used the closer the program looks like 0s and 1s, which is what the computer actually "understands". That is, the presence or non-presence of electrical current through its circuits.

High-level programming languages resemble human-like instructions, for example: if (x>5) then print “Greater than 5”. In order for the computer to be able to execute a program like this, a compiler is needed. The compiler will “translate” the text written in the programming language to 0s and 1s. The higher the level of a programming language, the higher the level of abstraction that it imposes on the programmer.

In the next section, I explain what a programming paradigm is, and types of programming languages used to teach and learn computer programming. This also explores the dilemmas faced by most instructors in identifying the most suitable programming language and environment for novices.

2.2 Programming Paradigms and Programming Languages

A programming paradigm defines a way of thinking about software development and is based on a mathematical theory or a coherent set of principles (Van Roy, 2009). Different approaches to programming (paradigms) have been developed over time. The most popular ones used for teaching computer programming are: the imperative; structured/procedural; and object-oriented.

Imperative programming focuses on how a program operates. It changes state information as needed in order to achieve a goal. Programs are composed of variables, assignments and calculations, statements for input and output, control statements such as selection and iteration. There is an implied sequential nature in the program’s activities: input, processing, and output.

Structured programming relies on procedure calls to create modularised code. A programming methodology, formulated by Dijkstra (1970), extends imperative programming and works in two phases. In the first phase, the programmer breaks down each problem into concrete sub-problems (problem decomposition) following a top-down approach. In phase two, the programmer works upwards, providing solutions to the smaller problems until the whole problem is solved. In structured programming, programs are composed of callable blocks of code called functions and procedures, and include all the constructs mentioned above (variables, input/output, control statements, etc.). Even though the procedural coding style is an older form of application development, it is still a viable approach when a task lends itself to step-by-step execution.

The ultimate goal of both imperative programming and structured programming paradigms is to “produce a program with exactly one entry point that can only be

built and executed after all its parts are (in some sense) completed" (Kölling, 1999, p. 4).

Object-oriented programming is a programming methodology that is based on the concept of objects. The programmer should in phase one identify the objects (entities) involved in the problem and then identify how these objects are related or interact with each other. In phase two, the programmer should specify the relevant data for each object and the possible operations to be performed on these data, and then design a user interface. Interaction with a user interface is not at all a sequential process but rather event-driven. Objects exist independently of each other, and operations can be executed on them. As a result, a user should be able to interactively create objects of any available class, manipulate these objects and call their interface methods. Booch (1989) stated: "Let there be no doubt that object-oriented design is fundamentally different from traditional structured design approaches: it requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture."

There is no best approach to tackling a computer problem. Each paradigm supports a set of concepts that makes it most applicable for a certain kind of problem (Van Roy, 2009). For some cases, the structured programming approach is more appropriate than the object-oriented one. For example, if the purpose of the program is to solve a mathematical formula and a Graphical User Interface (GUI) is not a requirement, then structured programming seems more appropriate. On the other hand, if the purpose of the program is to handle student grades in courses, then the object-oriented approach will be more efficient. Using the object-oriented approach does not eliminate the application of structured programming constructs; rather it is using them within a different context. Most programming languages nowadays are multi-paradigm ones (Van Roy, 2009).

"A multi-paradigm programming language is a programming language that supports more than one programming paradigm. The central idea of a multi-paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms. The design goal of such languages is to allow programmers to use the best tool for a job, admitting that a single paradigm cannot solve all problems in the easiest or most efficient way." (Mozilla Developer Network, 2013)

Table 2.1 shows the top ten programming languages based on the PPLI (Popularity of Programming Language Index) which is created by analysing how often language tutorials are searched on Google. The percentage change is calculated by comparing

the same data retrieved a year earlier in November 2017. From all the languages included in the list below, only C is not considered object-oriented.

Rank	Change	Language	Share	Trend
1	↓	Java	21.4 %	-1.9 %
2	↑	Python	18.6 %	+5.2 %
3	↓	PHP	8.2 %	-1.5 %
4	↑	JavaScript	8.0 %	+0.5 %
5	↓	C#	7.6 %	-0.9 %
6	↓	C++	6.3 %	-0.7 %
7	↓	C	6.3 %	-0.9 %
8	↓	Objective-C	3.9 %	-0.6 %
9	↑	R	3.8 %	+0.6 %
10	↑	Swift	3.1 %	+0.3 %

Table 2.1: Popularity of programming language index
(retrieved from http://pypl.github.io/PYPL.html, Nov. 2018)

The same source, in May 2019 in a tag cloud, shows Python, Java and JavaScript as the first three most popular programming languages (see Figure 2.2).

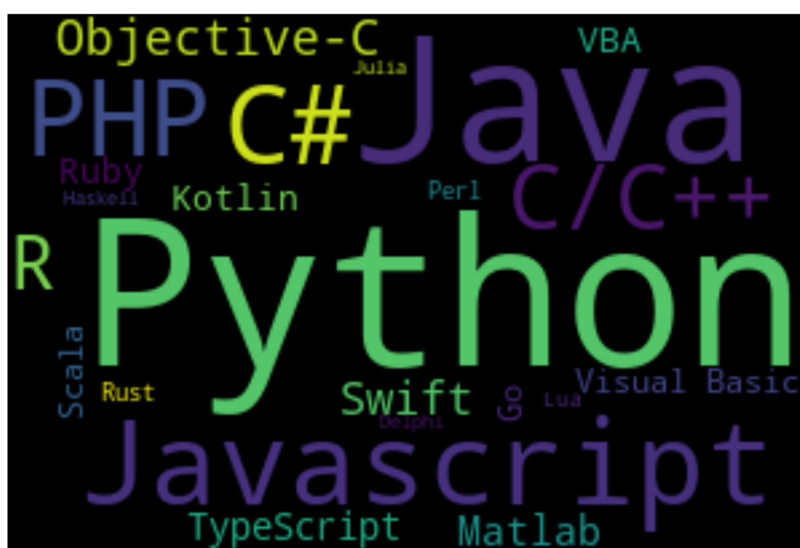


Figure 2.1: Programming languages popularity tag cloud
(retrieved from http://pypl.github.io/PYPL.html, May 2019)

In the 1990s, introductory programming education shifted towards object-oriented programming (Morris *et al.*, 1999; Pears *et al.*, 2007; Davies, *et al.*, 2011; Decker & Simkins, 2016) and until today most universities choose an object-oriented language for their introductory course. Nevertheless, the fact that an object-oriented programming language can also be used to teach fundamental programming

constructs using the "imperatives-first" approach makes them even more popular amongst educators.

A short survey administrated during Fall Semester 2017 to 50 educators in high schools and universities in Greece, demonstrated that most educators (40%) currently use Python to introduce programming concepts to students while Java still holds a strong share (23%) either with an emphasis on objects (14%) or imperatives-first (19%) (see Figure 2.2).

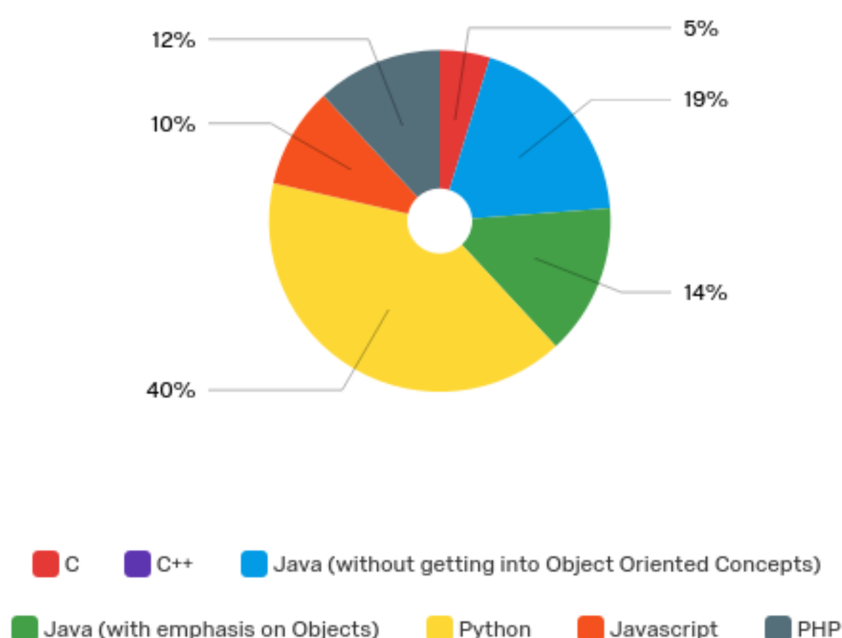


Figure 2.2: Programming languages used in Greek schools

Choosing the "imperatives-first" or the "objects-first" paradigm seems to be a defining factor for many introductory courses. Over the past ten years there has also been a trend to introduce students to "safer" programming languages (a move from lower-level languages such as C to higher-level languages such as Java and C++), or to scripting and loosely-typed languages (such as Python or JavaScript) or even to syntax-light ones (such as Alice and Scratch), but the initial debate still stands (Davies *et al.*, 2011; ACM Computing Curricula Task Force, 2013).

The "objects-first" approach to teaching programming seems to prevail over the "imperatives-first" (Iling *et al.*, 2003; Hu, 2004; Xinogalos *et al.*, 2006), but the

debate as far as which of the two approaches is more effective for teaching introductory programming courses still exists (Dale, 2006; Pears *et al.*, 2007).

Researchers that support the introduction to programming using the "imperatives-first" or "objects-later" paradigm argue that the object-oriented paradigm is far more complex and has a longer learning curve (Wiedenbeck *et al.*, 1999) and hence is more difficult (Thomasson *et al.*, 2006). Additionally, knowledge and experience gained from structured programming is a requirement to form a solid basis to work effectively with objects later on (Hu, 2004; Reges, 2006).

On the other hand, researchers that support an introduction to programming using an "objects-first" paradigm argue that since there has been a shift in professional programming towards object orientation (White & Sivitanides, 2005), learners should be familiarised with it as early as possible (Decker, 2003). They also argue that a high percentage of novice programmers only "know" how to interact with the computer using their mouse, in a windows interface, and possibly they have never seen a command line environment (Culwin, 1999).

To minimise the perceived difficulties and to support the "objects-first" strategy, various educational software tools have been developed such as BlueJ, JEliot, Greenfoot and Alice (Xinogalos *et al.*, 2006, Sun, 2010; Dann *et al.*, 2012; University of Kent, 2014;) that allow the interaction with objects from the beginning. Studies have shown that these tools can help novice programmers build a more concrete understanding by providing appropriate conceptual models (Yiğit *et al.*, 2015).

The "Introduction to Programming" module in XYZ college historically follows the "imperatives-first" and "objects-later" approach to programming using Java, which is one of the most widely used programming languages both in education and in professional software development. The task of writing a program can be accomplished using a number of programming environments. The choice of a programming environment could potentially affect the understanding and performance of a novice programmer. In this respect, in the next section, I present a taxonomy of programming environments.

2.3 Types of Programming Environments

Writing a computer program in its pure form requires a text-editor and a command line tool to compile and execute a program. Over the years, programming environments have evolved and have integrated the text-editor, the compiler, the

execution environment and many more assistive features for programmers. There are varying levels of assistive features provided by different programming environments, ranging from low assistive features to very high ones.

A text-editor such as notepad and a command like compiler, is on the low assistive side. There are no assistive features for the programmers. A program visualisation environment such as JEliot and BlueJ contains a simplified text-editor with an integrated compiler. Again, there are almost no assistive features for the programmers. Code is written in Java but enables the learner to visualise a step-by-step execution of the program. Method calls, variables and their values, arrays, operations and output are displayed on a screen as the animation goes on.

An Integrated Development Environment (IDE) such as Eclipse, NetBeans or JDeveloper, is considered to be on the moderate assistive side since it integrates the text editor and the compiler and offers a number of features for authoring, modifying, compiling, deploying, versioning and debugging software. Most professional programmers use integrated development environments to write software.

A number of programming environments have been developed through the years to introduce programming concepts to younger students. Their design is fundamentally different from professional IDEs due to their pedagogical purpose for use and have been termed Initial Learning Environments (ILE) (Fincher & Utting, 2010). ILEs include Visual Programming Environments (VPEs) such as Scratch, Alice and AppInventor. These environments are considered to be on the high assistive side, since the programmer will focus only on the programming logic and will not be required to type any code. In a symposium discussion on Computer Science Education about the goals and effects of Alice, Scratch and Greenfoot, Steven Cooper argues that the power of visualisation comes when an animation does not work correctly, and students are able to understand where the "error(s)" in the code resides. He also mentions that the focus of these programming environments is on providing an engaging experience for the students so that they will want to learn programming (Utting *et al.*, 2010). Although the concept of program visualisation and visual programming is mentioned here within the context of programming environments, a more detailed analysis follows in Chapter 4, with a focus on the difficulties students face when learning how to program and the role of visualisations in the facilitation of learning computer programming.

A short presentation of each type of programming environment (along with a representative software) follows in the next sections.

2.3.1 A Text-Editor and Command-Line Compiler

Using a text editor to type your program code, save it and use a command prompt in order to compile it and execute your program requires a very strong knowledge of both the programming language and operating system commands. Although such an environment does not require the knowledge of using a specialised environment with a complex set of features, it can be frustrating for novice programmers (see Figure 2.3).

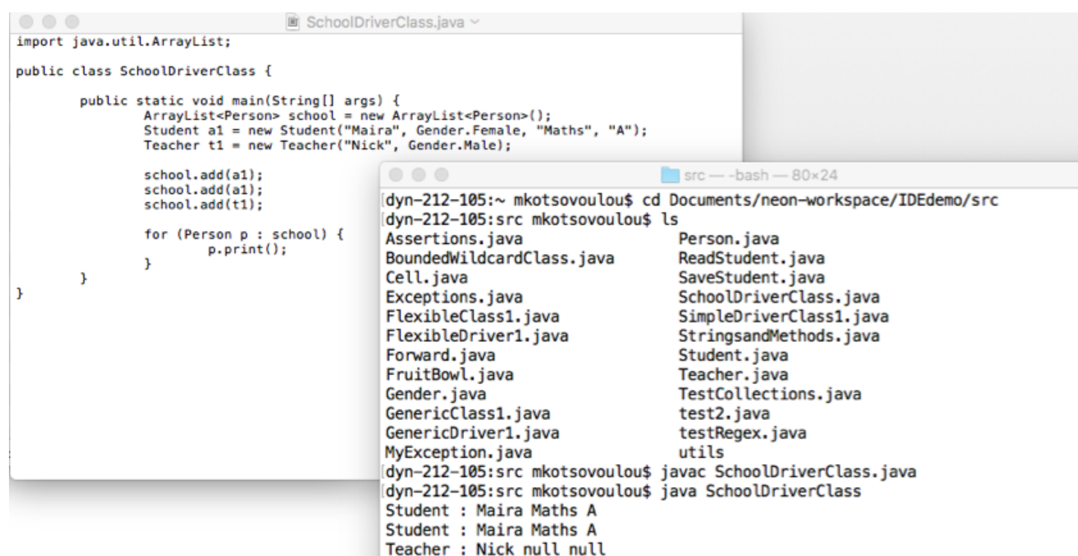


Figure 2.3: A text-editor and a command line compiler

On the other hand, text editors have also evolved and can provide colour-coding, syntax highlighting and formatting which could be helpful for novice programmers.

Without overlooking the frustration caused to students by this environment, anecdotal research (Chen & Marx, 2005) shows that some educators might still choose to introduce students to writing programs using a text editor and a command line compiler. The rationale behind this choice is to provide students with a broader understanding of programming fundamental concepts such as writing code, compiling, executing and editing to enhance their mental models of the programming life cycle at a lower level.

2.3.2 Program Visualisation Environment

There are two well-known program visualisation systems which are widely used in the educational setting: BlueJ and JEliot. BlueJ offers static visualisation of Java classes, while JEliot offers a dynamic visualisation of program execution.

BlueJ is one of the first programs developed, aiming at teaching introductory object-oriented programming, in 1999. BlueJ integrates a simple text-editor with a Java compiler and offers some assistive features to the learners, such as syntax and scope highlighting (each code block is coloured) and this helps in spotting syntax errors and misplaced curly brackets. The main feature of BlueJ is the static visualisation of a class structure (attributes and method) as a Unified Meta Language (UML) diagram and animates the creation of all possible instances of a class at run-time (see Figure 2.4). Furthermore, it allows the learner to interact with the object instances by creating them, calling their methods and inspecting their state with easy-to-use menus and dialogs.

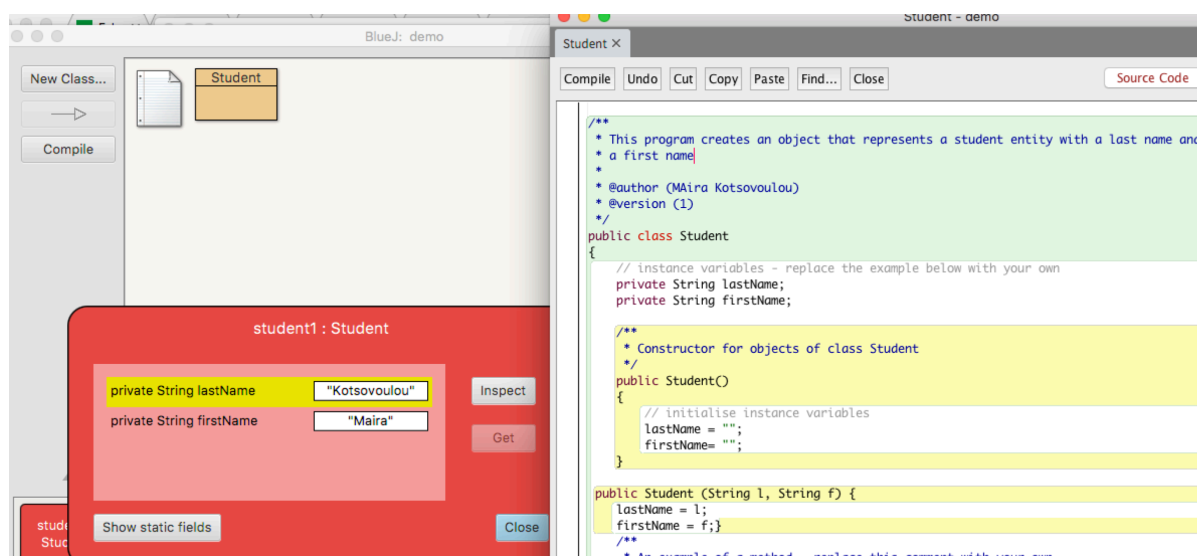


Figure 2.4: BlueJ class inspection feature and the text-based code editor

However, it does not provide any dynamic visualisation of the program execution.

Jeliot on the other hand is a program visualisation application. The development of the Jeliot family took more than ten years and was research oriented. Several versions of the concept of visualising the execution of a program have been developed, namely Eliot (developed at the University of Helsinki, Finland in 1993), Jeliot I (developed at the University of Helsinki, Finland), Jeliot 2000 (developed at the Weizmann Institute, Israel) and JEliot3 where the software has become product-like, both usable and stable. Each version of the program incorporated findings from the previous version's empirical evaluations (Moreno *et al.*, 2004).

Jeliot integrates a simple text editor and a compiler plus a live-theatre mode. The learner has to type a program using the Java programming language and compile it. Unfortunately, the compiler neither highlights possible syntax errors while typing the

program, nor during the compilation phase. The errors will appear to the user when he/she chooses to execute the program.

The main feature of Jeliot is the "theatre mode" and the "call tree". When the program does not contain any syntax errors, the execution starts by animating all methods, variables, method calls, expressions and their possible evaluations in the theatre mode (Figure 2.5) The user can slow down, speed up or pause the animation to observe the results.

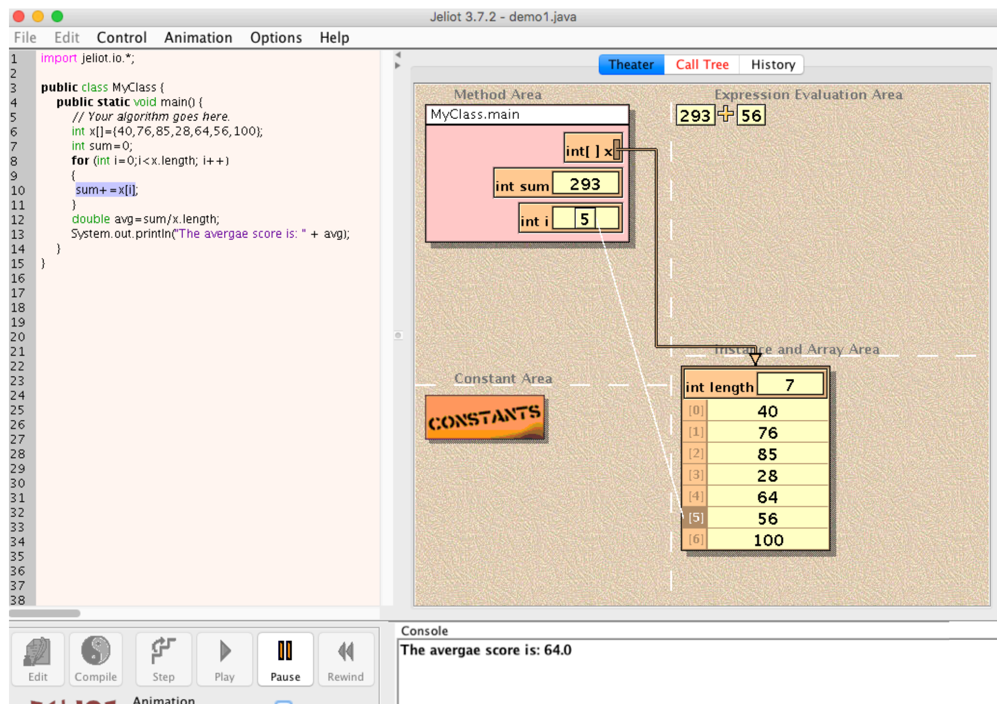


Figure 2.5: Jeliot programming environment - Theatre mode

In the Call Tree mode, the user can observe the hierarchy of method calls. Starting from the main method, all other method calls are depicted in a tree along with their actual parameter(s) values and their respective return values (see Figure 2.6).

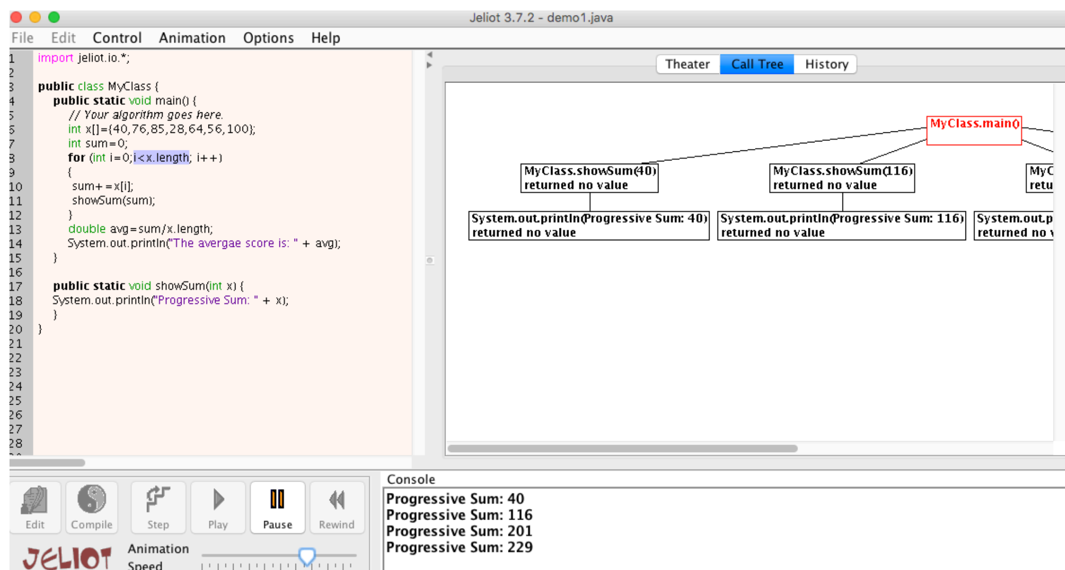


Figure 2.6: Jeliot programming environment - Call tree mode

As mentioned before, the main disadvantage of JEliot is its over-simplified text-editor which does not highlight possible typographical errors or syntax errors, but Jeliot can be incorporated into BlueJ as an extension and provide the learners with required editor functionality.

2.3.3 **Integrated Development Environment (IDE)**

An integrated development environment (IDE) is a programming environment packaged as an application. IDEs provide software developers with many tools that assist them in writing their programs. Features provided include: colour coding, code completion/suggestion, matching of brackets, code formatting/indentation, debugger, creating of code documentation, version control and application deployment (see Figure 2.7).

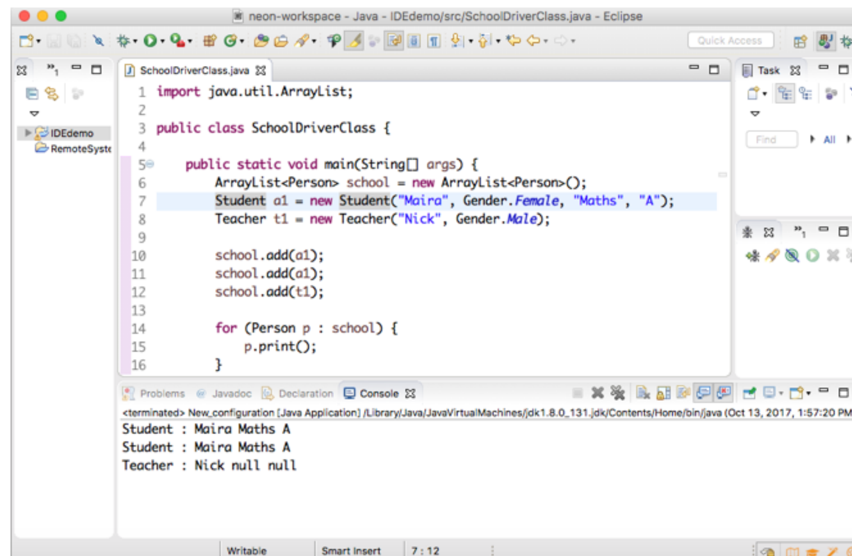


Figure 2.7: Eclipse IDE

IDEs abstract the process of compilation and execution since it happens automatically with the press of a button. An IDE compiles the code, and if compilation is successful, it executes the program inside the same environment (an integrated console). As a result, there is no switching back and forth between the editor and command prompt. In case of a syntax error, the IDE highlights the line number with the error and even suggests possible ways to correct it.

All modern integrated development environments (IDEs) provide users with a debugger system. The debugger is used to perform advanced step-by-step program tracing. Using the debugger, the student can monitor the contents of the memory as the program executes, and pause the execution upon request.

Although an IDE supports programmers with writing their code, it has a higher learning curve than using plain text editors and command prompts. Research also shows that students often rely too much on the automated tasks, but that they do not really understand what is happening behind the scenes (Chen & Marx, 2005).

A study conducted by Dillon *et al.* (2012) showed that students struggled with using a command prompt environment regardless of their prior experience and confidence with programming, but they were able to use IDEs more effectively.

2.3.4 **Visual Programming Environments**

Since the early 1960s, researchers identified the need to make programming accessible to a larger number of people. Since then, a number of programming languages and environments have been built with this intention. Kelleher and Pausch

(2005) and Guzdial (2004) provided us with a taxonomy of programming environments' design to make programming more accessible to novice programmers of all ages, up to the time their article was published.

Historically speaking, the purpose of visual programming environments as identified by research has been three-fold: a) to make programming more accessible to some particular audiences; b) to improve the correctness with which people perform programming tasks; and c) to improve the speed with which people perform programming tasks (Burnett, 1999).

Nowadays, there has been a shift from this purpose towards the engagement of the student/developer to design programs within the context of their actual and specific interests (stories, games, simulations, etc.) and to the immediate feedback provided by the environment.

This is in contrast to conventional programming exercises, which ask students to create programs that display "hello world", perform calculations and sort numbers. Furthermore, in visual programming environments, syntactic complexity is hidden, and tasks are directed to hands-on problem solving. These environments are designed to avoid common beginners' mistakes in programming such as syntax errors and aim to bridge the gap between program-code and the visual/human representation of the code output. Therefore, instead of typing commands, students can drag-and-drop blocks of code into a predefined structure to form a computer program. Because of their shape, these blocks can only be placed in a sequence that makes sense, and the compiler will never give an error message due to mismatched braces or a missing semi-colon. The main focus of visual programming environments is to facilitate hands-on problem solving and to encourage and retain "at risk" students (Utting *et al.*, 2010).

Just a simple search of the term "visual programming" in the ACM Digital Library (November 2017) resulted in 134,883 articles and with conjunction with the term "novice programming" resulted in 97,473 articles. This shows an impressive research interest in visual programming environments.

In the next section, I will briefly introduce the most widely-used visual programming environments: Alice, Greenfoot, AppInventor and Scratch. In Chapter 4, a discussion on how using visualisations can assist students to overcome the barriers associated with computer programming follows. In Chapter 5, each one of these programming environments will be evaluated, using a participatory action research methodology, in order to investigate student perceptions about each one - the tools' enjoyment,

usability and suitability towards the achievement of the specific module's learning objectives, to observe how each of these tools affected students' motivation to learn programming, and to identify the one to be used in the main study.

2.3.4.1 Alice

Alice was created by a Research Group at Carnegie Mellon University under the direction of Randy Pausch (http://www.alice.org) and, as described by its creator, is "designed to be a student's first exposure to object-oriented programming" by allowing students to easily create interactive animated stories and/or games that take place in virtual 3-Dimensional worlds.

Alice provides students with a drag-and-drop interface that allows them to focus on programming concepts while also protecting them from syntax errors. Initially, students are presented with a gallery of template worlds and choose the world setting that they will work with. Next, they instantiate numerous objects, animals and/or people. Additionally, students define how objects will move and interact with each other. Movement and interactions are created with scripts. A script is constructed by dragging and dropping commands into the procedure area and changing related properties. "Move forward 1 meter" or "turn left 30 degrees" are examples of Alice commands. Commands can be performed in sequence (Do-in-Sequence) or simultaneously (Do-Together). Loops can also be used (Do 5 times, or while distance < 3 repeat a block of code).

Sprites (objects) also respond to user interaction provided via mouse or keyboard (Cooper *et al.*, 2000; Utting *et al.*, 2010). At each point during development, students can run their animation, visually observing and directly relating to the results of their specific programming actions. Feedback is immediate and highly visual (Figure 2.8). "This leads to an understanding of the actual functioning of different programming language constructs" (Cooper *et al.*, 2000).

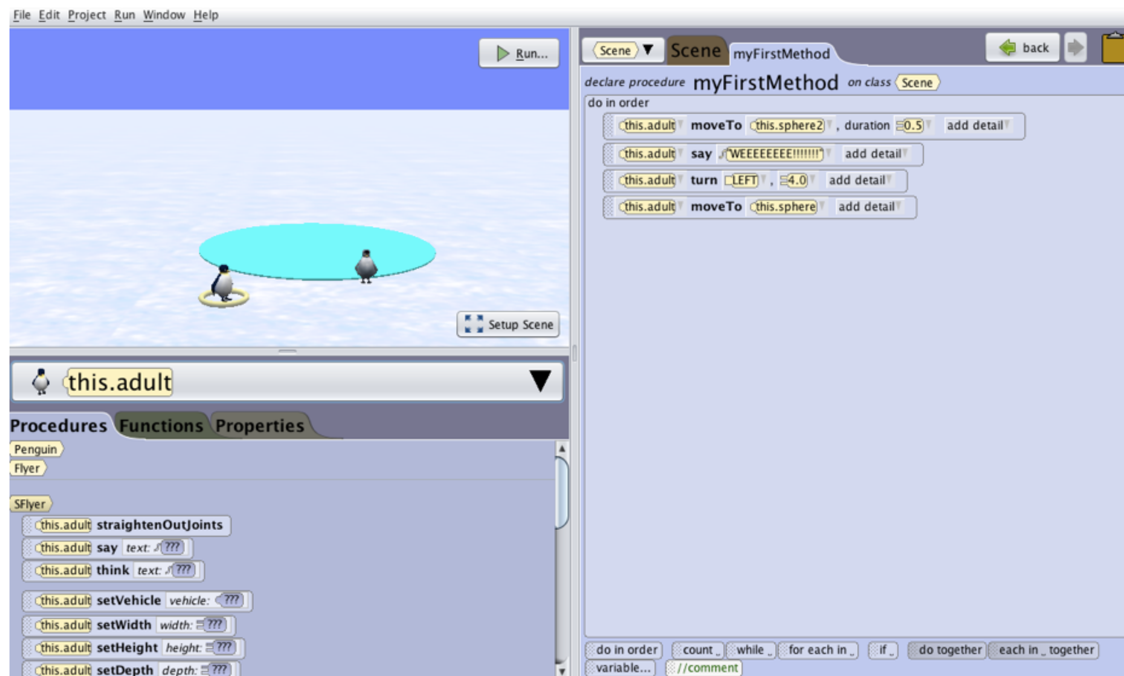


Figure 2.8: Alice programming environment

2.3.4.2 Greenfoot

Greenfoot was created by Michael Kolling and Poul Henriksen at the University of Kent (http://www.greenfoot.org).

The Greenfoot system uses the metaphor of a World subclass and one or more Actor subclasses that are placed in the world. Actors act and interact with the world or other Actors to implement the application idea (scenario). Each time a student places an object on the world, a new named subclass of the actor is created with its own image, size and placement within the world. The idea behind Greenfoot is to introduce students to concepts of object-oriented programming, such as inheritance, instantiation, polymorphism, properties and methods, in a way that is easier to understand. Students can view and modify the source code that is automatically generated for each object created. The level of abstraction provided by Greenfoot is comparatively lower than that in Alice, as it contains all elements of an integrated programming environment: a compile button and execution control (Figure 2.9); a text-based code editor (Figure 2.10) or a frame-based code editor (Figure 2.11).

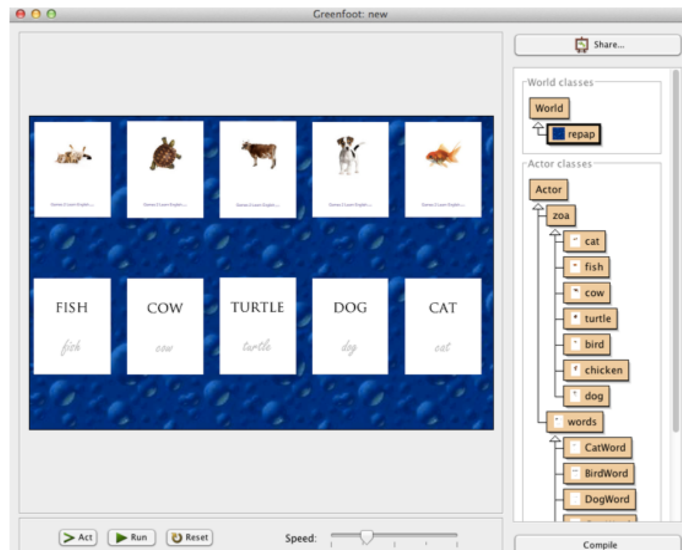


Figure 2.9: Greenfoot's program design editor

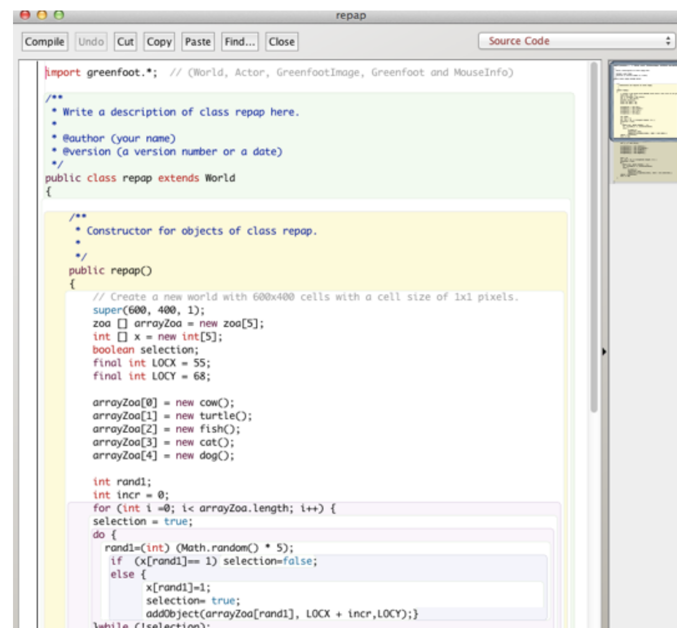


Figure 2.10: Greenfoot's text-based Java Editor

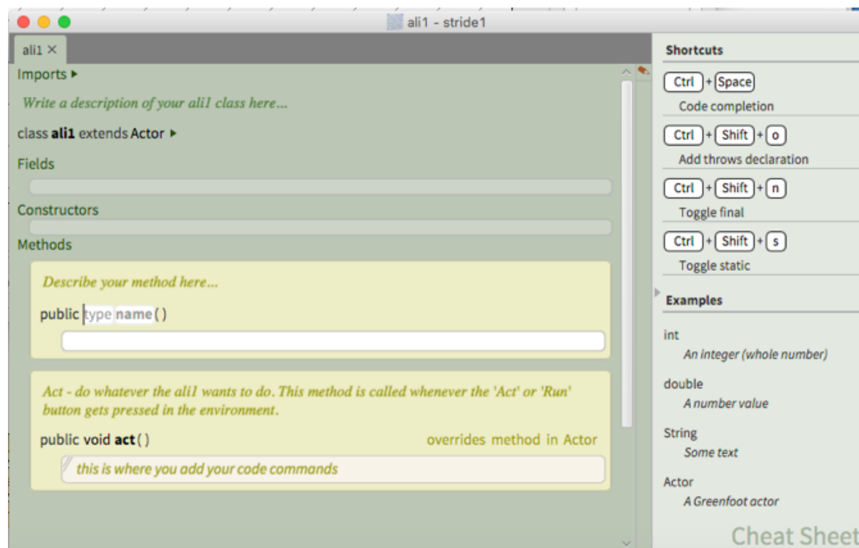


Figure 2.11: Greenfoot's frame-based editor

Using Greenfoot, students come one level closer to coding their programs, but the nature of the program is different from traditional IDEs. Greenfoot's latest version 3 also includes an intermediate coding environment: a frame-based editor named Stride.

A preliminary investigation on the usability of this frame-based editor by McKay and Kolling (2013) showed that novice programmers performed insertions, modifications, deletions and code replacements considerably faster than other coding editors.

2.3.4.3 Scratch

Scratch was developed in 2007 by Mitchel Resnick and Natalie Rusk as a project of the "Life Long Kindergarten" group in MIT (https://www.media.mit.edu/projects/scratch).

Scratch is based on the ideas of Logo (Papert, 1980) to support constructionist learning, but replaces typing code with a drag-and-drop tile-based approach inspired by LogoBlocks (Begel, 1996) and EToys (Kay, 2005). LogoBlocks (Figure 2.12) and EToys (Figure 2.13) were both developed around 1996 and they followed a similar drag-and-drop approach of jigsaw-like puzzle pieces or tiles that contained programming instructions.

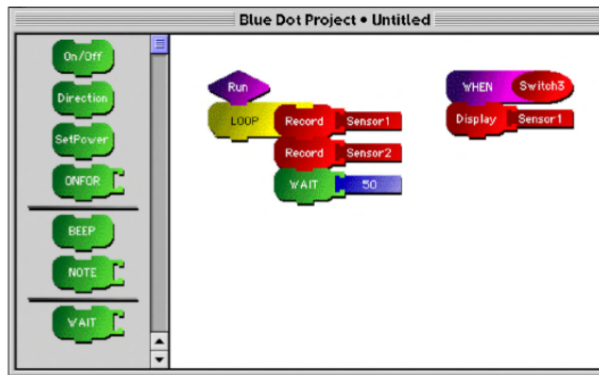


Figure 2.12: LogoBlocks

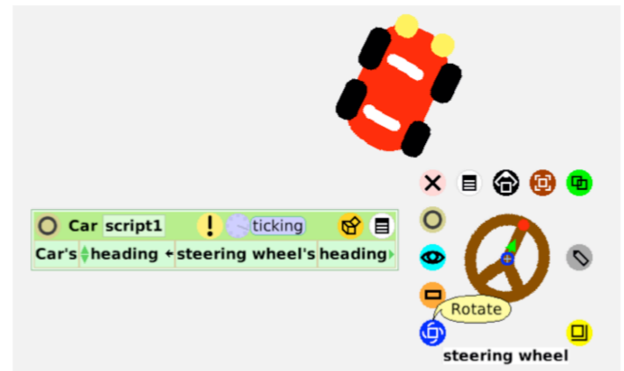


Figure 2.13: EToys

Although Scratch was inspired by these tools, the main design principle was to make it more ‘tinker-able’, more meaningful and more social than its predecessors or other programming environments in the same category (Resnick *et al.*, 2009). Scratch took its name from the *“scratching”* technique used by disc jockeys, when they move back and forth a vinyl record (or a Compact Disk or even a virtual disk on a computer) to create a percussive or rhythmic sound while mixing music clips together in creative ways. Thus, “scratching” in computer terms, according to Lamb (Lamb & Larry, 2011) refers to reusable pieces of code which can be combined, shared and adapted.

Scratch pedagogy is grounded on the ideas of “creativity”, “interactivity”, “sharing” along with “mathematical and computational ideas”. Resnick and his team based the development of Scratch on the idea that a computing environment should have a low floor (easy to get started) and a high ceiling (opportunities to create increasingly complex projects over time). This metaphor was initially introduced by Seymour Papert in Mindstorms (1980), but the Scratch development team also based the development of the tool on the idea that languages need “wide walls”. An environment with “wide walls” supports many different types of projects so people with different interests and learning styles can become engaged. They also argue that the development of Scratch with this triplet: low-floor/high-ceiling/wide-walls was not easy (Resnick *et al.*, 2009).

The key Scratch component is media manipulation and supports programming activities that align with the interests of young people, such as creating animated stories, games, and interactive presentations. Re-mixing is another key component of Scratch pedagogy. Re-mixing a Scratch project allows a user to copy another users’ project, see the code inside, learn from it, experiment with it, and extend it (always retaining a reference to the original work).

A Scratch project consists of a fixed stage (backdrop) and several movable objects (sprites). In the following example (Figure 2.14) the airplane is the sprite and the sky with the clouds is the stage. Each sprite contains its own set of images (Figure 2.15), sounds (Figure 2.16), variables, and scripts (Figure 2.17).

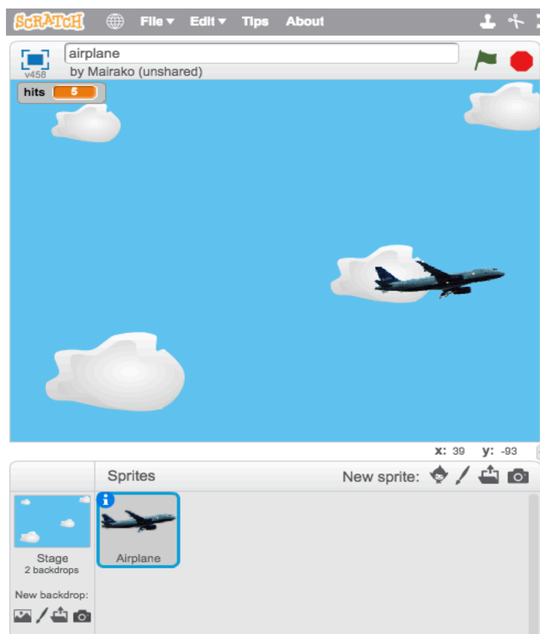


Figure 2.14: Scratch stage and sprites

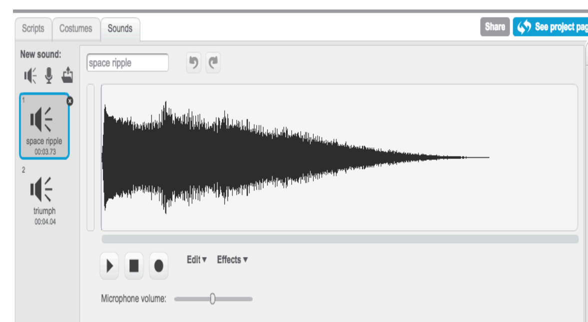


Figure 2.16: Sound in Scratch

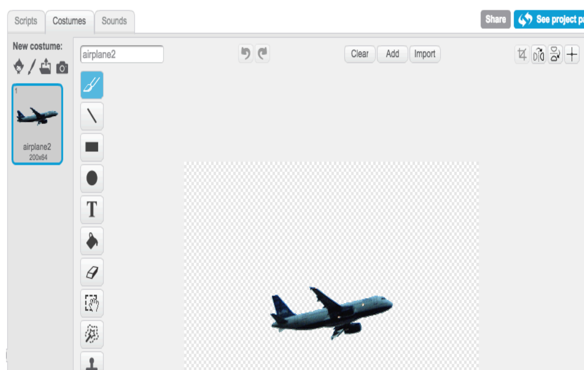


Figure 2.15: Image in Scratch

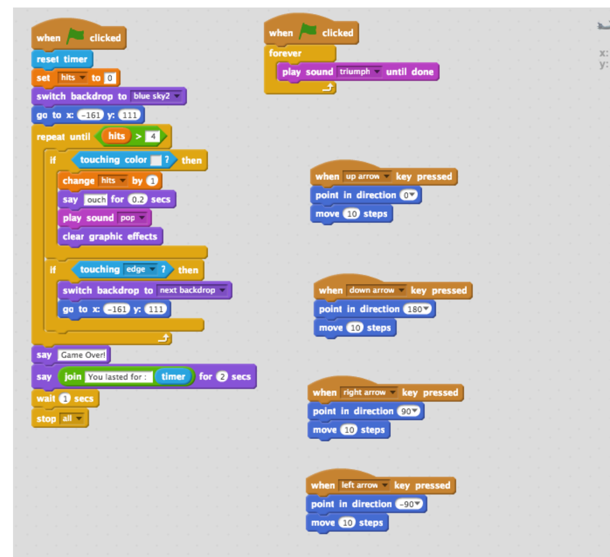


Figure 2.17: Scripts in Scratch

In order to create a program, the learner drags command blocks from a palette (Figure 2.18) and drops them into the code area by sticking them together in order. The whole process is like putting together puzzle pieces.

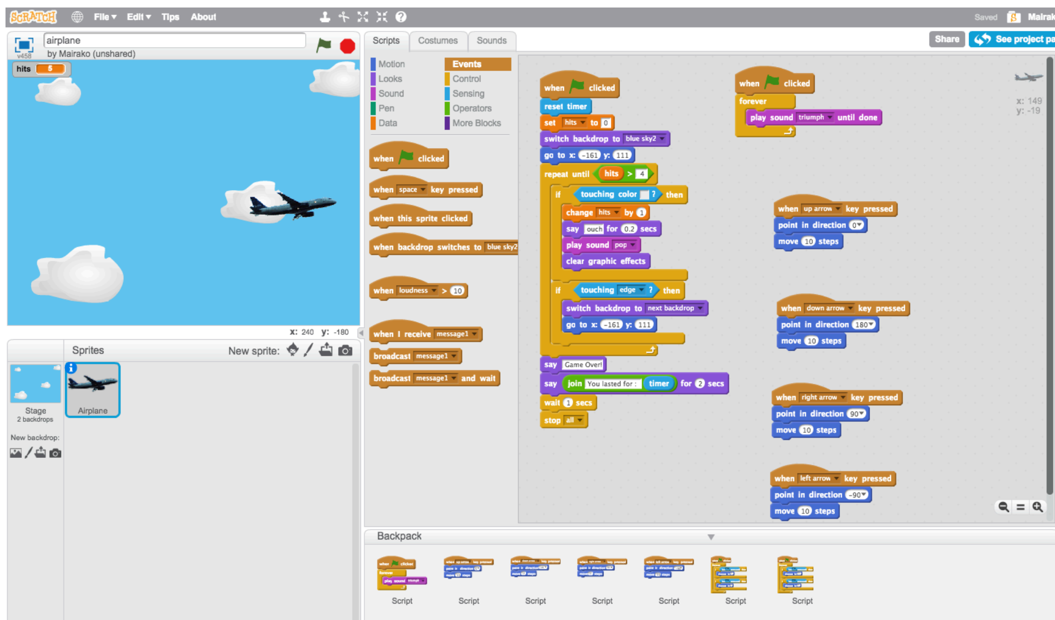
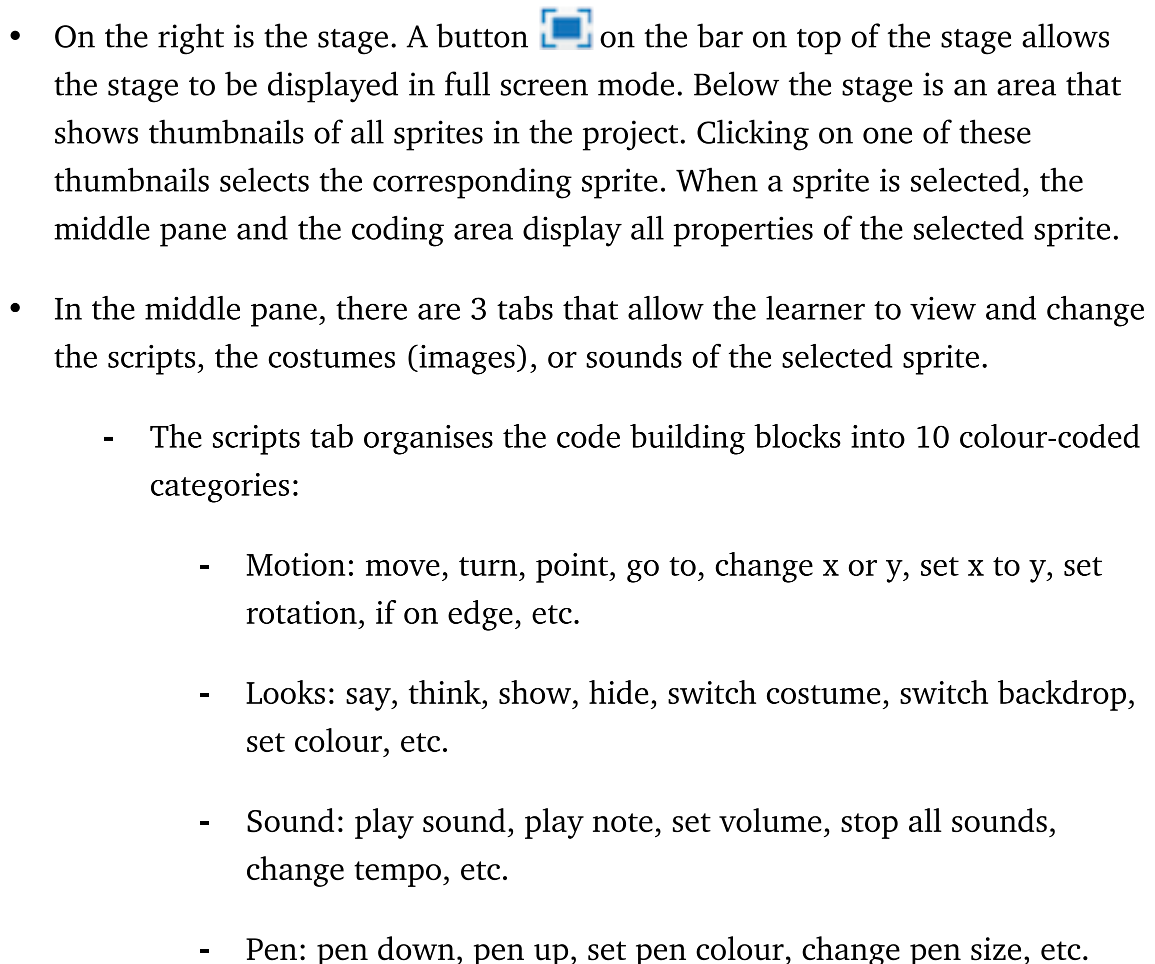


Figure 2.18: Scratch development area

The full Scratch development area (see Figure 2.18) is divided into four areas:



- On the right is the stage. A button on the bar on top of the stage allows the stage to be displayed in full screen mode. Below the stage is an area that shows thumbnails of all sprites in the project. Clicking on one of these thumbnails selects the corresponding sprite. When a sprite is selected, the middle pane and the coding area display all properties of the selected sprite.

- In the middle pane, there are 3 tabs that allow the learner to view and change the scripts, the costumes (images), or sounds of the selected sprite.

    - The scripts tab organises the code building blocks into 10 colour-coded categories:

        - Motion: move, turn, point, go to, change x or y, set x to y, set rotation, if on edge, etc.

        - Looks: say, think, show, hide, switch costume, switch backdrop, set colour, etc.

        - Sound: play sound, play note, set volume, stop all sounds, change tempo, etc.

        - Pen: pen down, pen up, set pen colour, change pen size, etc.

    - Data: create a variable, set a variable value, show/hide a variable value.
    - Events: When flag clicked (to start the program), when button pressed, when a sprite is clicked, etc.
    - Control: repeat, if… Then… Else, forever, wait, etc.
    - Sensing: is key pressed? Is sprite touching colour or other sprite? Is mouse down? Etc.
    - Operators: +,-, *, /, <, >, and, or, pick a random number, etc.
    - More Blocks - Extensions: create predefined blocks (procedures), etc.

- On the left-most pane is the scripting area. This is where the actual program is composed.

Having the command palette always visible encourages exploration. Any individual block of code or a stack of blocks can be executed immediately (even before the program is complete) to preview its functionality just by double-clicking on it. This immediate feedback reduces the novice programmers' fear of the unknown. The fact that not all blocks fit together makes writing code less error prone. The area where a block can be dropped is highlighted, and a block cannot be placed at a point where it does not make sense, program-wise.

Furthermore, when a Scratch program executes, by clicking on the green flag, the code that is built inside every sprite executes at the same time. Scratch code is not executed in a serial manner; some run-time events, such as a key being pressed, or a mouse click on a sprite, can change the flow of the program. Scratch also provides the users with another valuable visualisation: it highlights those blocks of code which are currently being executed. Thus, Scratch users experience an event-driven, multi-threaded runtime environment without even realising it and are exposed to advanced programming concepts in a tangible manner. Hopefully, this will help them develop a more solid process model of how a computer program works.

2.3.4.4 APP Inventor

MIT App Inventor is an open source visual programming language which utilises ready-made code blocks for building Android Apps and was initially developed by

Google's Mark Friedman and MIT Professor Hal Abelson in 2009; the MIT Version was released in 2012.

APP inventor aims to introduce to inexperienced novice programmers the basic programming constructs, while focusing on application creation concepts using drag-and-drop visual building blocks. The perceived ease of use of the simple graphical interface transforms the complex language syntax of a text-based coding environment to plugging puzzle pieces together. The graphical programming user interface of APP inventor is based on Open Blocks visual programming (see Figure 2.19, Figure 2.20) and resembles Scratch, the Hour of Code and StarLogo TNG. Block-based programming environments are widely used in lower and upper schools to introduce basic programming ideas (Maloney *et al.*, 2007; Nikou & Economides, 2014; Panselinas *et al.*, 2018; Papadakis & Orfanakis, 2018).

Applications created using APP Inventor can be easily deployed on Android mobile devices and enable students to easily share their work with their family and peers.

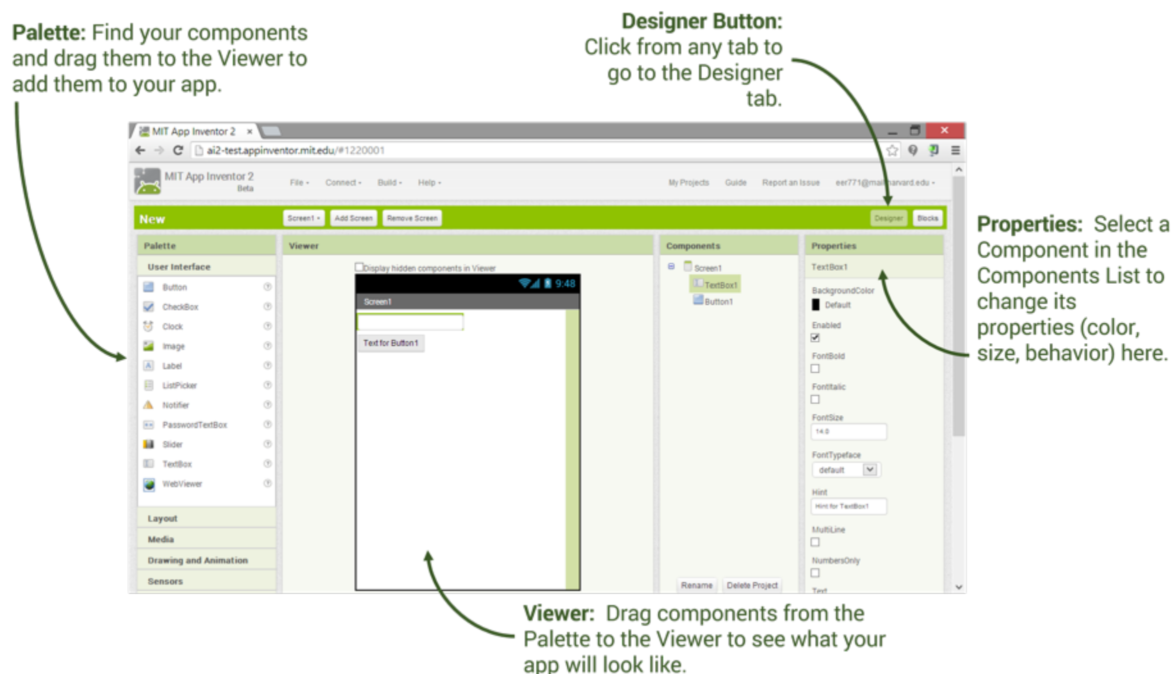


Figure 2.19: AppInventor: Program design

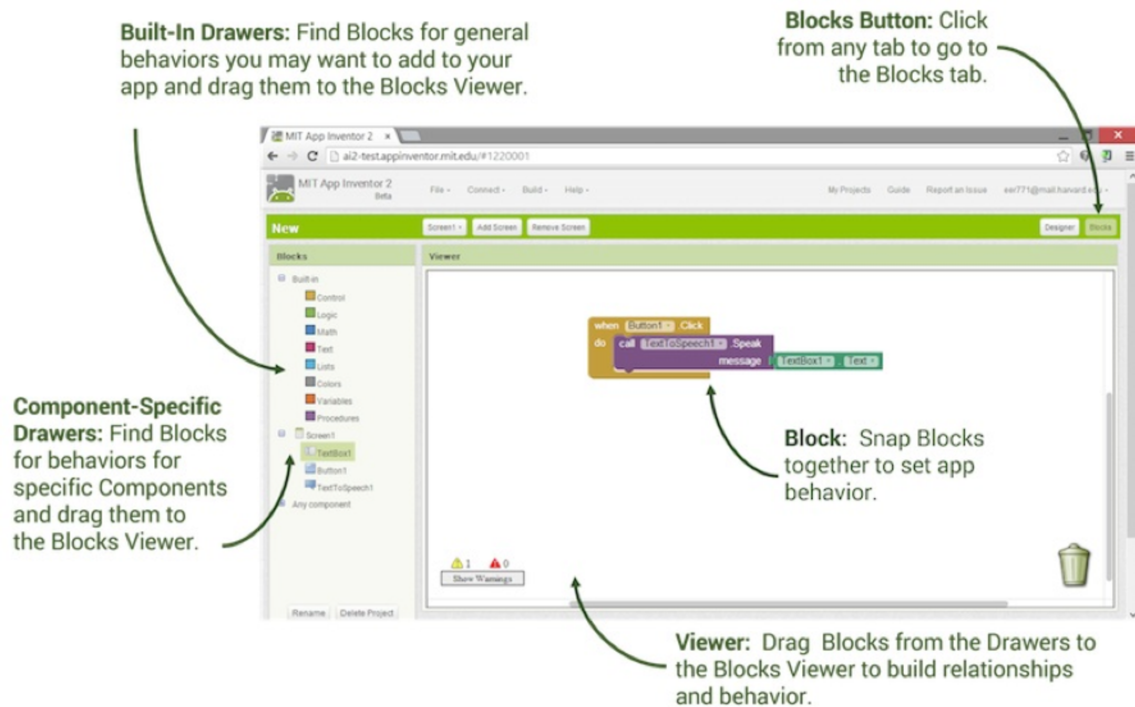


Figure 2.20: AppInventor: Block-based code editor

2.4 Conclusion

This chapter provides the relevant context and prepares the ground for justifying the choice of tools used in this research project. To this end, it explores the field of computer programming, programming paradigms and programming environments, with an emphasis on the description of the functionality of four well-known visual programming environments (Alice, Greenfoot, AppInventor and Scratch).

These VPEs have as a common characteristic the use of predefined code blocks, but also have some identified differences regarding the types of programs they can create, the level of programming experience they require and the programming concepts they can demonstrate. Table 2.2 summarises the main characteristics of the four visual programming environments discussed above.

VPE	Editor	Mode of user interaction	Purpose (types of programs they can create)	Requires previous programming experience	Can be used to demonstrate	
					OOP	Multithreading
Greenfoot	Frame-based or text-editor	Desktop Application	Create 2-D games	some	X	
Alice	Block-based editor	Desktop Application	Create 3-D interactive stories, games or animations	none	X	
AppInventor	Block-based editor	Online	Build mobile applications of various types for android devices	some understanding of user interaction with application elements (buttons, input text, etc)		X
Scratch	Block-based editor	Online and desktop	Create 2-D interactive stories, games or animations	none		X

Table 2.2: Comparison of the main characteristics of Greenfoot, Alice, AppInventor, Scratch

It should be pointed out that Greenfoot and Alice are strongly based on object-oriented programming concepts where actors are instances of pre-defined objects and the concept of inheritance and sub-classing is fundamental in program development, while the ability to demonstrate multithreading concepts can best be demonstrated by Scratch and AppInventor. Alice and Scratch do not require previous programming experience, while Greenfoot and AppInventor (based on my teaching experience) will be more suitable for students who have some prior experience with coding and application development respectively.

The four VPEs addressed in this chapter have been selected for this study because they have a long history of serving as focal programming tools in introductory programming courses. They were all designed to support teaching and learning how to program by making things easier and more pleasurable; support a "motivational" approach to learning; were designed to make conceptualisations visible to learners thus reducing cognitive load; have textbooks to support teaching; and have a vast online presence with active educator communities. Finally, they have all been researched in the past for their educational effectiveness (see Section 4.5).

Chapter 3 The Theoretical Framework

Chapter 2 introduced the concepts of computer programming and presented different programming paradigms, as well as types of programming environments that exist in the market today. This chapter examines cognitive and educational theories and their application to teaching programming, the concept of approaches to learning and theories of motivation. Finally, through a presentation and comparison of assessment tools (for learning approaches and motivation), the discussion advances justification for the selection of the ones used in the study.

3.1 Educational Theories

Cognitive psychologists and educators have long been interested in understanding: a) the nature of learning as an active process (perception, thinking and knowledge representation); b) knowledge organisation in memory (rote memorisation versus comprehension); c) how learning evolves towards problem-solving (Mayer, 1981; Wertheimer, 1983) and d) the importance of prior knowledge in assimilating new material (Shuell, 1986).

"Meaningful learning" and retention, according to Ausubel (1963), are facilitated when the learner has a meaningful cognitive framework within which to organise, process and assimilate newly-presented material. Michael (2001) contrasts meaningful with rote learning and stresses the importance of the ability of the learner to actively do something with all the memorised information. He elaborates on Ausubel's definition that meaningful learning results in knowledge that is well integrated with everything else that one knows and that can be accessed from many different starting points.

Cognitive psychologists have identified three conditions (comprising an information processing model), for meaningful learning to occur:

- Reception: the learner should pay attention to the information he/she receives to register this within short-term memory.
- Availability: the learner needs to recognise or identify connections of similar context within long-term memory, or "appropriate anchoring ideas", as Ausubel terms them.
- Activation: the learner must use existing knowledge and establish connections between this knowledge and new material at hand.

A number of practical studies have concentrated on applying cognitive and educational theories to the teaching of programming to improve student learning:

- Mayer (1981) applied Ausubel's (1960) idea of "advance organisers" to provide a framework for the reception and availability conditions mentioned above and to define the process of meaningful learning (or assimilation to the schema) of technical information.

- Du Boulay (1986) made use of metaphors and analogies in teaching programming and based his studies on the development of a "concrete model" for teaching LOGO and argued that there are two approaches to teaching and learning how to interact with the computer; the "black box" approach and the "glass box" approach. The first approach is based on the idea that the internal operations of the machine are not visible and not even necessarily of any interest - like a true black box - leaving the learner to focus exclusively on inputs and outputs. The second approach is based on the idea that the learner should attempt to understand how the computer operates internally - hence like a glass box.

- Chalk, Boyle, and Fisher (2003) tested the application of "learning objects" in an attempt to improve student performance.

- Hadjerrouit (1999) presented a teaching approach based on the principles of constructivist epistemology.

The concern for learning focuses on the way in which people acquire new knowledge and develop skills and the way in which existing knowledge and skills are modified (Shuell, 1986). There are many different approaches to teaching and learning. Historically, psychologists tried to develop a hypothesis of how individuals acquire, retain, and recall knowledge. Although a number of definitions of learning appear in the literature, Shuell (1986) provided one which is broad enough to incorporate the views of different paradigms: "Learning is an enduring change in behaviour, or in the capacity to behave in a given fashion, which results from practice or other forms of experience", as interpreted by Schunk (2012).

By studying different learning theories, we as teachers can better understand how learning occurs and how to apply underlying principles to effectively identify appropriate instructional tools, techniques and strategies that promote learning in the field. A literature review on learning theories suggests that there are three widely accepted paradigms of learning: behaviourism, cognitivism and constructivism.

Schunk (2012), in his book, considers the following critical issues in the study of learning and tries to compare the manner in which each paradigm addresses them: a) how learning occurs; b) what the role of memory is; c) how transfer occurs; and d) which learning tasks are more appropriate for each paradigm.

The behaviourist learning paradigm is based on the view that learning occurs when the learner presents a recognised response as a reaction to an external stimulus. Thus, a primary focus is on how to form strong and lasting associations between stimuli and responses. In Skinner's (1953) view, a response to a stimulus is more likely to re-occur in the future as a function of the consequences of prior responses and is promoted by repetition and positive reinforcement. The learner is characterised as being reactive to conditions in the environment. As a result, learning strategies that follow the behaviourist approach can be applied to specific learning tasks such as recalling facts, defining and illustrating concepts, applying explanations and automatically performing a specified procedure (Ertmer & Newby, 2013).

The cognitive learning paradigm is based upon the view that the learner, rather than being a passive receptor of information or knowledge, is an assimilator of knowledge, which is actively constructed based on pre-existing cognitive structures. The understanding of cognitive processes is essential to this paradigm, as related mental activities must be identified and targeted to promote the most effective learning. The learner's knowledge schema is viewed as an organised hierarchical structure (Bruner, 1964; Gagné *et al.*, 1993) and the emphasis is not on human behaviour but on the mental processes that take place in order for learning to occur. The mental processes include perception, thinking, knowledge representation, memory and transfer. The learner is characterised as being an active participant in the learning process (Shuell, 1986). As a result, learning strategies that follow the cognitivists' approach can be applied to more complex learning tasks that involve reasoning, problem-solving and information-processing (Ertmer & Newby, 2013).

The constructivist learning paradigm is based on the view that a learner is capable of constructing his or her own knowledge, though within the framework of a subjective model of representation. Piaget (1977) asserts that learning occurs by an active construction of meaning, rather than by passive recipiency. This paradigm approaches learning as a process in which one integrates new information with previous knowledge and experiences (Duffy *et al.*, 1993) in order to actively construct an extended knowledge schema in a piece-wise fashion (Steffe & Gale, 1995).

A number of researchers consider constructivism as a branch of cognitivism because both theories view learning as a mental activity, but there is a fundamental difference

between the two in the way that knowledge is assimilated, whether or not transferred into the memory. Some behaviourists and cognitivists have argued that knowledge can be “mapped” onto a learner (to acquire meaning), while constructivists have argued that a learner builds personal interpretations (to create meaning) based on his/her unique experiences and interactions with the world. Jonassen (1991) has described three stages of knowledge acquisition (introductory, advanced, and expert) and argues that constructive learning environments are most effective for the stage of advanced knowledge acquisition, in which learners are called upon to deal with more complex, and unstructured problems, whereas, for the introductory phase, behavioural or cognitive approaches are more appropriate.

As a result, learning strategies that follow the constructivists’ approach can be applied to problem-solving activities within loosely structured realms to promote self-realisation and allow learners to adapt their mental models to newly-discovered knowledge. A more detailed discussion on the development of mental models required to understand and apply a programming language to solve problems follows in Chapter 4.

Nonetheless, no traditional learning theory can be deemed absolute and all-encompassing. For example, the possible social dimension to learning was missing from the learning paradigms mentioned above. Learning, as a human behaviour, is such a complex and multi-faceted process that it would be considered limiting to describe it using only cognitive or behavioural factors. Bandura (2001), in his social learning theory, argued that there is a continuous interplay of both, along with the inevitable influence of the social environment and the subsequently observed modelled activities. The social factor is also encountered in Vygotsky’s social development theory, which is based on the idea that social interaction, culture and language play a major role in the development of cognition. His “zone of proximal development” defines a higher level of cognitive development which can be reached with guidance by adults and interaction with peers (Vygotsky, 1978).

Situated learning theory (Lave & Wenger, 1991) is also based upon the view that a social practice dimension is intrinsic to the learning process. This paradigm considers the social environment to be that in which knowledge exists and throughout which it can be disseminated efficiently. As such, learners enhance, challenge, validate, and ultimately deepen their understanding within the context of peer- or group-related activities involving communication, synergy, sharing, and overall interaction with others in communities of practice.

Although the fundamental processes of learning have not changed, the world around us has. As such, learning theories of the past have not considered the impact that technology advancements would bear on the learning process (Siemens, 2005). During the past 20 years, with the advent of the Internet, there has been a dramatic change regarding how, where and with whom people learn. According to Prensky, *"More and more young people are now deeply and permanently technologically enhanced, connected to their peers and the world in ways no generation has ever been before"* (Prensky, 2010, p.2). In respect to these fundamental changes in the learning space and the need to develop the required skills of today's learners, significant consideration is given to the theoretical perspectives of constructionism (Papert & Harel, 1991) and social constructivism (Vygotsky, 1978) and their corresponding teaching pedagogies.

Papert, influenced by Piaget, actively supported the idea of learning-by-making. In his essay "Situating Constructionism", he refrains from providing a definition for constructionism; instead, he encourages readers to construct their own meaning of the term from the examples he provides. Papert was inspired by observing children create "soap" sculptures throughout a semester and noticed how this process provided them the time, opportunity, and environment to think, try out their ideas, talk about them, and see other people's work. He envisioned "soap-sculpture math" and learners as designers and builders of meaningful "public-entities" with the use of technology empowered learning tools (Papert, 1980; Papert & Harel, 1991).

In accordance with the constructionist and social constructivist approach, the social element is highly present in professional software development. Both theories emphasise the use of peer collaboration and problem-based learning as an instructional method (Savery & Duffy, 2001), as opposed to social learning theory which emphasises observation of modelled behaviour of others. In the same context, Lave and Wenger (1991) stress the importance of collaboration among learners and the exchange of ideas within and even across communities of practice.

Problem-based learning as an instructional method is based on Dewey's philosophical view that practical experience plays a significant role in learning (Dewey, 1938). Problem-based learning involves contextualising learning, given a "real-world" problem that requires a solution. Students work in small groups to solve a problem provided by their teacher. Problem-based instruction aims to promote students' critical thinking, enhance their problem-solving skills, and prepare them for their future practice or professional endeavours.

Last but not least, I should refer to the alternative theory of connectivism, characterised by Siemens as being the learning theory of the digital age (Siemens, 2005). According to connectivism, learning is no longer viewed as an individualistic activity, but rather as a process, not entirely controlled by the learner, that occurs in vague environments (clouds) with interconnected yet continuously changing nodes (information sources). New information can be acquired almost instantly, but it becomes critical that the learner is capable of differentiating between important and unimportant information. Creating up-to-date and accurate knowledge is a key element of all connectivist learning activities, along with the ability to create meaningful connections between concepts and ideas. On the other hand, my own personal experience has shown that the vast amount of information which resides on the Internet might unwantingly lead novice learners to consume it as is, without first trying to understand it, make connections to prior knowledge and critically evaluate it.

In terms of software development, this is not merely an individual task. Programming, at the professional level, requires individuals to work in teams, collaborate and share knowledge to ensure the success of a project. Such real-world programming requires extensive communication and collaboration amongst a plethora of people (customers, end users, system analysts, database designers, network architects and many other specialists) with the primary goal of creating a solution to a real-world problem (Kotsovoulou & Stefanou, 2016).

My teaching methodology in programming modules has been influenced thus far by the social constructivist philosophical view and the constructionist instructional method of problem-based learning; constructivism, because it advocates that teaching and learning should involve hands-on activities and practical sessions through which knowledge can be built; social constructivism, because it emphasises the use of peer collaboration (Prawat & Folden, 1994), applied in class with peer programming; and problem-based learning because I value the importance of creative experimentation in the construction of software as a "public entity".

A large body of researchers in computing education is also considering instruction of novice programmers from a constructivist viewpoint (Ben-Ari, 1998; Van Gorp & Grissom, 2001; Wulf, 2005). Dewey's inquiry-based education, Piaget's constructivism, Vygotsky's social constructivism, Papert's constructionism, Bruner's discovery learning, Pask's conversation theory, Schank's problem-based learning, Marton's deep learning and Lave's socio-cultural learning are all in accordance with

Tyler's views that *"learning takes place through the active behaviour of the student: it is what he does that he learns, not what the teacher does"* (Tyler, 1949).

Laurillard (2006) points out that the most influential writers on learning have emphasised the importance of active learning and, based on that, she argues that the promotion of active learning in a social context should be the focus for the design of teaching-learning environments. Laurillard's conversational framework is based on this view, but also underlines the importance of the iterative dialogue between the teacher and the student. This dialogue is an interplay between theory and practice and is essential for "making the abstract concepts concrete," as Resnick (2007) states. Goodman has also coined the concept of the dialectical interplay when he described mathematics as a social product that is "*created and developed by the dialectical interplay of many minds, not just one mind*" (Goodman, 1979, p. 545).

Teaching computer programming is an endeavour that goes far beyond the traditional lecture format, which was prevalent in the past. It requires the combination of a variable set of teaching methodologies and hands-on problem-solving activities, including partial code completion, code walkthroughs, testing and debugging, use of rich instruction environments including animations and visualisations, group-work and collaboration. Most of the activities involved in computer programming education align with the social constructivist pedagogy and have a practical application in the design of instructional material.

Interestingly, Alesandrini and Larson (2002) specified ten activities grouped into five phases which can provide the foundation for a constructivist approach to instruction: investigation, invention, implementation, evaluation and celebration.

- The investigation phase includes contextualisation and clarification of the task as well as research on how to approach the solution.
- The invention phase includes planning, designing or building a model.
- The implementation phase includes the realisation of the solution but sometimes overlaps with the invention phase during modification of the initial design.
- The evaluation phase includes testing, modifying, interpreting and reflecting.
- Finally, the celebration phase includes the presentation of the results in a larger group.

Drawing from my experience as a software developer and an IT educator, I cannot overlook the fact that all of the activities mentioned above have a great resemblance to the software development lifecycle steps, which include: the preliminary analysis and definition of the requirements (investigation); systems design (design and modelling); development (implementation); evaluation (testing and modification); and deployment and presentation of the solution (celebration). Thus, it can be argued that following the software development lifecycle itself utilises a constructivist approach to problem-solving.

I also believe that teaching computer programming to novices requires a continuous refinement of the understanding of the concepts and that each new concept should be built upon a solid foundation. Mayes and Fowler (1999) proposed a learning model of gradual refinement of understanding and conceptualised teaching and learning as an iterative process, which repetitively cycles through its three discrete stages: conceptualisation, construction, and dialogue. It can be argued that these three stages of the learning model actually follow the three learning theories (cognitive, constructivist and socially-situated learning). More specifically, the conceptualisation phase, because of its focus on organising concepts and forming relationships between pre-existing knowledge and new information, can be viewed as being based on cognitive theory. Next, the construction phase, because it targets the creation of new knowledge through practice and problem-solving, can be seen as illustrating the constructivist theory. Finally, the dialogue phase, concerned with peer collaboration and group discussion, can be taken as being aligned with socially-situated learning theory.

Given the above, a successful teaching methodology for introductory computer programming is, therefore, likely to be one that builds on, and extends, useful features from all of the theories mentioned above and aims to provide the students with appropriate feedback and support. The role of evaluation is crucial in this respect for the success of such a teaching methodology.

To summarise, becoming a computer programmer requires mastering a number of diverse skills, ranging from analytical reasoning and problem-solving, to critical thinking and research. Social skills (such as communication and collaboration) are also imminently important for successful programmers. In order for learners to accumulate all of these assorted and complementary skills, the instructor should create an educational setting where diverse and constructive real-world activities take place through repetitive and collaborative practice. This last point was critical in the

formulation of the teaching methodology and implementation framework I employed in carrying out this study.

3.2 Learning Approaches, Learning Styles and Assessment Tools

"Learning is the process whereby knowledge is created through the transformation of experience" (Kolb, 1984)

One of the research questions of this study seeks to explore possible relationships between students' learning approaches with the perceived enjoyment, ease of use, usability and usefulness of visual programming environments. The purpose of this section is to review the literature on learning approaches in the field of computer programming, and to identify the tool to be used later in the study.

Many studies have explored the complexity of learning how to program and the associated difficulties that students face during this process. A number of factors have been identified as directly or indirectly contributing to this complexity. Among these factors are: approaches to learning; learning styles; and motivation. Since one of the research questions of the present study is to identify possible correlations between students' learning styles and approaches and their preference of visual programming environments, as well as the possible effects of these environments on their motivation to learn, there is a need to further investigate relevant background that supports this possible relationship. Felder and Brent (2005) categorised student diversity with regard to approaches to learning, learning styles and intellectual development, based on the fact that students inevitably have different backgrounds, strengths and weaknesses, levels of motivation, attitudes about teaching and learning, approaches to studying, responses to specific classroom environments and instructional practices. They argued that if teachers could identify key differences in these three diversity domains and design a variety of instructional methods and learning tasks, they could possibly conceivably address students' learning goals and promote intellectual development more effectively. Thus, I will commence by exploring these diversity domains and reviewing the literature on possible findings of their relationship to learning computer programming, with a goal of selecting the most appropriate instrument for my investigation.

Marton and Saljo (1976) introduced the term "levels of processing", based on the idea that university students, when assigned a task, would adopt either a surface or a deep level of processing information. Later on, the researchers reconsidered the term

and changed it to “approaches to learning”, in order to stress the element of intentionality and awareness, along with the cognitive memory processing that takes place during learning. Later, Pask (1976) introduced the term “learning strategy” and identified a third approach to learning - a so-called strategic one. Ramsden (1981) also supported the term “strategic approach”, which at a later date Biggs (1987) called “achieving approach”.

Students who adopt a surface approach focus on memorising and reproducing facts with the intention of satisfying course or assignment requirements, without attempting however to reflect on or fit information into a larger context. On the other hand, students who adopt a deep approach focus on transforming and relating ideas to previous knowledge with the intention of understanding the facts by critically evaluating concepts and becoming actively involved in the process. Finally, students who adopt a strategic approach focus on organising the concepts, putting targeted effort into their studying, managing their time and relating the ideas to assessment criteria, all with the intention of achieving the highest grades possible. Biggs (1987) refers to this last kind of learning approach as “model student behaviour”. Entwistle (2005) summarised the defining features of each approach to learning that have emerged from relative research, in the following list:

Deep Approach (Transforming) with the intention - to understand ideas for yourself by

- Relating ideas to previous knowledge and experience
- Looking for patterns and underlying principles
- Checking evidence and relating it to conclusions
- Examining logic and argument cautiously and critically
- Becoming actively interested in the course content

Surface Approach (Reproducing) with the intention - to cope with course requirements by

- Studying without reflecting on either purpose or strategy
- Treating the course as unrelated bits of knowledge
- Memorising facts and procedures as a matter of routine
- Finding difficulty in making sense of new ideas presented
- Feeling undue pressure and worrying about work

Strategic Approach (Organising) with the intention - to maximise grade potential by

- Putting consistent effort into studying
- Finding the right conditions and materials for studying
- Managing time and effort effectively
- Being alert to assessment requirements and criteria
- Gearing work to the perceived preferences of lecturers

These approaches have become central to subsequent research on studying and the development of more effective teaching (Laurillard, 1979; Entwistle, 1991; Gibbs & Awards, 1992).

Related research in the field of teaching computer programming and approaches to learning has shown that learning a programming language requires a student to employ both a "deep" and a "surface" approach (Bruce *et al.*, 2004; Pea & Kurland, 1983; Winslow, 1996). Students that focus exclusively on coding and syntax rules employ an inadequate "surface" approach to learning how to program, as opposed to students that focus on problem-solving using the programming language syntax rules as a means to reach their goal. These students employ a "deep" approach to learning. Students that follow the "strategic" approach will use all the skills mentioned above. The fact that a programming language can be rotely memorised (as a vocabulary and a set of syntax rules) does not imply that a student can construct programs solely based on that skill. In order to be efficient and proficient in programming, the student should learn how to think in computer terms, implement abstraction and modularity, construct algorithms and know where to look for "surface" information such as syntax rules.

Entwistle, in his study in 1990, identified four study orientations associated with approaches to learning:

"Thus, the deep approach was associated with a holistic style (…making use of a wide variety of information…) and intrinsic motivation (interest in the subject matter itself) to form a meaning orientation. Surface approach went with serialist style (a narrow, cautious stance relying on evidence and logical analysis) and fear of failure within a reproducing orientation, while strategic approach indicated a use of both deep and surface approach supported by a competitive form of motivation (need for achievement) combined with vocational motivation within an achieving orientation." (Entwistle & Ramsden, 1983, p.49)

Research also shows that students' approaches to learning can be influenced by the learning context. Intentionality, stemming from the specifics of a given learning situation, is a strong motivator that may determine the approach that the student opts to follow in that particular case. Although the internal cognitive processes that take place during learning are more "fixed" (Rayner & Riding, 2010) than the approaches to learning themselves, learner predispositions about the learning subject and motivation to learn might act as a bridge for the formation of a learning strategy. This is also supported by Laurillard (2005) and Entwistle and Tait (1990): if the origin of the approach is the student's intention, then, as the student may have different intentions within different learning situations, the same student may use either approach, on different occasions.

Apart from intention, which, as discussed above, affects the way they approach a learning task, students further differ in how they receive, absorb and process information. These differences in student preferences and traits are generally referred to as learning styles. Learning styles represent a rather broad concept which can be viewed through a number of approach angles, which accounts for the variety of related definitions, models and measures that can be found in associated literature. Some representative but very similar definitions include:

- *"…the ways in which an individual characteristically acquires, retains, and retrieves information"* (Felder & Henriques, 1995, p.21).
- *"… traits that refer to how individuals approach learning tasks and process informatio*n" (Morrison *et al.*, 2011, p.58).
- *"… the way individuals begin to concentrate on, process, internalize, and retain new and difficult information."* (Dunn *et al.*, 2009, p.136).

A more elaborate definition is provided by Keefe and Languis (1983): "learning style is the composite of characteristic cognitive, affective, and physiological factors that serve as relatively stable indicators of how a learner perceives, interacts with, and responds to the learning environment. It is demonstrated in that pattern of behaviour and performance by which an individual approach of educational experiences" (p.140-141) as cited in Keefe (1985).

Curry (1983) proposed a layered model (using the onion layer metaphor) in an attempt to categorise the numerous learning style viewpoints, models and their respective measurement instruments. She initially identified three general areas of research on learning styles, which shared common characteristics: instructional

preference (which is considered to be the most observable, but also the most influenceable and, as such, the least stable for measurement); information processing; and cognitive personality (which is considered the most stable one). Later on, Curry (1987) expanded that model further by including the social interaction layer.

In a short description of Curry's extended onion model for the classification of the concepts of cognitive or learning styles, the outermost and most inclusive layer, instructional preference, encompassed individual preferences concerning the physical environment (sound, light, temperature, space), emotion (motivation, persistence, responsibility, structure), sociology (learning alone or in a group), physiology (auditory, visual, tactual, and kinaesthetic) and psychology in the aspect of processing inclinations (global/analytic, impulsive/reflective) (Dunn, Dunn, & Price, 1979). The social interaction layer involves individual preferences regarding independence, participation and collaboration with others (Dunn *et al.*, 1979; Riechmann & Grasha, 1974). The information processing (or cognitive style) layer includes individual preferences for the ideal intellectual approach to assimilating information: holistic/analytic, verbal/imagery, sensing/intuitive (Dunn *et al.*, 1979; Kolb, 1984; Felder & Silverman, 1988; Riding & Sadler-Smith, 1997), as well as for approaches to learning: surface/deep/strategic (Entwistle, 1991; Biggs, Kember, & Leung, 2001). Finally, the innermost layer of cognitive personality involves individual differences in observed behaviours across different learning situations (Riechmann & Grasha, 1974; Myers, 1998; Grasha, 2002).

Having identified the diversity of learning styles' dimensions and measurement instruments, Curry (1990) points out a major concern in the academic field - that being the failure to identify and agree upon those characteristics which are most relevant to learners in a given learning situation. On the other hand, following Felder's recommendations on the usefulness of identifying students' learning styles, approaches to learning and levels of intellectual development, as well as the correlations among them, the next step for this research is to identify an appropriate learning style assessment instrument.

While there are a number of learning style assessment tools and methodologies (Allert, 2004; Coffield *et al*, 2004; Zualkernan *et al.*, 2006), two similar assessment instruments are predominant in science and engineering education— Kolb's Learning Styles Inventory (LSI) (Kolb & Kolb, 2014), which is based on Kolb's (Kolb, 1984) experiential learning theory, and the Soloman–Felder (Felder & Soloman, 1993) Index of Learning Styles Questionnaire (ILS), which is based on a learning styles model developed by Felder and Silverman. Both instruments have been validated and

have been used in computer education research (Zualkernan *et al.*, 2006; Da Silva Carmo *et al.*, 2007; Chen & Lin, 2011; Li & Yang, 2016).

Kolb's experiential learning style theory (see Figure 3.1) is based on the assumption that effective learning happens when the learner progresses through four stages in a cyclic fashion: Concrete Experience (CE), Reflective Observation (RE), Abstract Conceptualisation (AC), and Active Experimentation (AE). Answers to questions like 'What? How? Why? and What if?' are involved in the process of learning from knowledge comprehension to knowledge transformation. He identified two dimensions in which learning takes place: processing (doing or reflecting) and perception (experiencing or thinking) and created a matrix to present this continuum. He argued that "*learning arises from the resolution of creative tension among these four learning modes*" (Kolb & Kolb, 2014). Concrete experiences are the basis for observations and reflections. These reflections are grouped and refined into abstract concepts. These abstract concepts may drive new actions which can be tested further to initiate new experiences. Kolb also believed that a single person cannot perform both variables at the same time, for example, doing and reflecting, or experiencing and thinking.

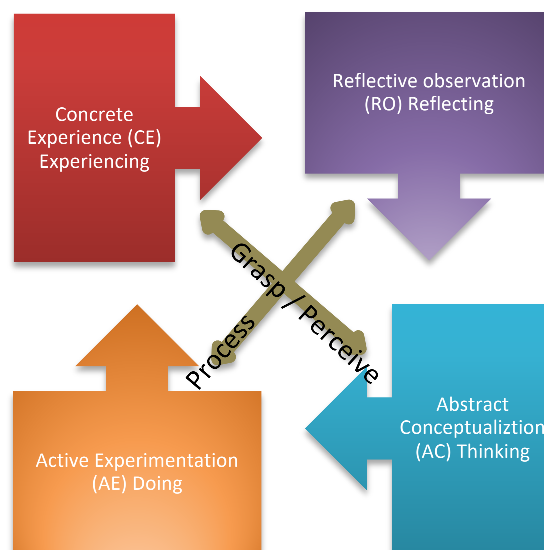


Figure 3.1: The experiential learning cycle

Hence, in Kolb's Learning Style refined Inventory (KLSI 4.0), each person's unique learning style is defined by the combination of his/her preferences for these 4 stages in the learning process. This combination can be charted into a unique kite-like shape created from the learner's degree of preference for each of the stages. The nine-style

typology, along with a description for each style is shown in Figure 3.2 (Kolb & Kolb, 2014).

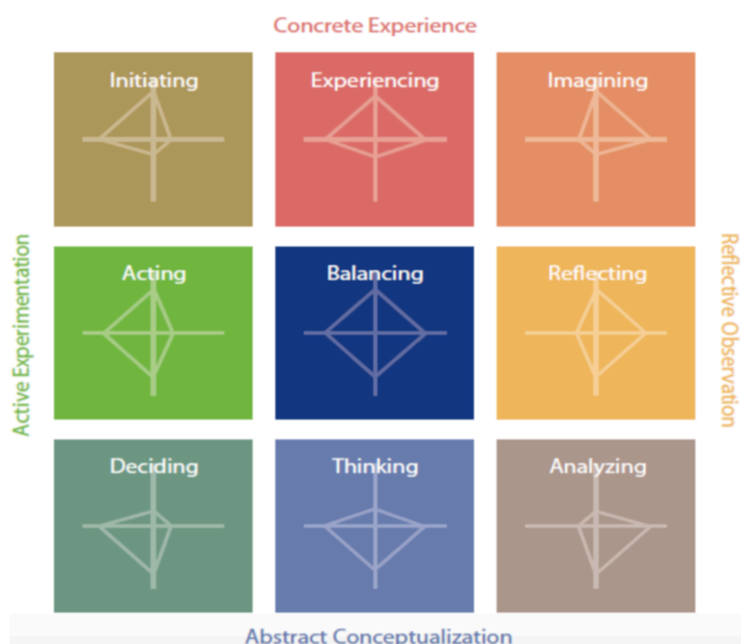


Figure 3.2: The nine learning styles in the KLSI 4.0

In 1988, Felder and Silverman developed a model to classify engineering students and professors according to where they fitted on a number of four scales with respect to the way they prefer to receive, perceive, process and understand information (see Figure 3.3). In their model, they included four dimensions extracted from previous research. The sensing/intuition dimension was based on Jung's theory of psychological types, is used in Myers-Briggs Personality Type Indicator (MBTI) and is closely related to Kolb's concrete experience and abstract conceptualisation stages of learning. The active/reflective processing dimension was based on Kolb's active experimentation/reflective observation stages, while the global/sequential dimension was based on Pask's holist/serialist learning strategies (Pask, 2010). The Visual-Verbal dimension is proposed by Felder and Silverman and is based on cognitive psychology research on how people receive sensory information (Felder & Henriques, 1995). The Index of Learning Styles (ILS) instrument was developed and validated by Felder and Soloman, in order to assess learner preferences on the four dimensions (see Figure 3.3).

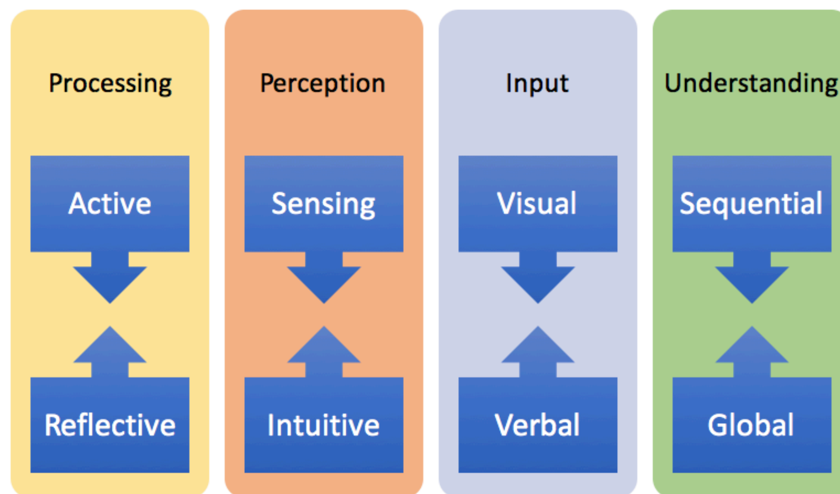


Figure 3.3: The four scales/dimensions of the Felder-Silverman Model and their respective learning style continuum

In the information processing dimension, active learners tend to retain and understand information best by engaging with it in something active - discussing it, applying it or explaining it to others. Learners with a reflective learning style preference, in comparison, prefer to think things through and work alone.

In the perception dimension, learners with a sensing learning style preference tend to like learning facts and procedures and are more practical. Conversely, learners with an intuitive learning style preference often prefer discovering possibilities and relationships and are more conceptual and oriented towards theories and meanings.

In the input dimension, learners with a visual learning style preference remember best what they see—pictures, diagrams, flow charts, timelines, films, demonstrations, etc. In contrast, learners with a verbal learning style preference get more out of words - written and spoken explanations.

Lastly, in the understanding dimension, sequential learners tend to gain understanding in a linear, stepwise fashion, with each step following logically from the previous one, and to learn in an incremental manner. Global learners, on the other hand, are holistic system thinkers who tend to learn in large jumps and absorb material almost randomly without seeing connections, and then suddenly “get it”.

Each of the instruments discussed above classifies learning style preferences based on opinion surveys, but Kolb’s model does not address Felder’s Visual-Verbal dimension or the Sequential-Global dimension.

Both tools have been used extensively in education, while a number of studies have identified weaknesses and limitations (Psaltidou 2009, van Zwanenberg *et al.,* 2000). Much criticism regarding the use of tools for the identification of students' learning style preferences is in line with Reylond's quote: *"Even using learning style instruments as a convenient way of introducing the subject [of learning] generally is hazardous because of the superficial attractions of labelling and categorizing in a world suffused with uncertainties"* (1997, p.128). Conversely, it is supported that identifying learning style preferences can be beneficial for students' self-development through self-awareness of their natural learning strengths (Kozhevnikov, 2007, Felder 2010).

Additionally, within the learning styles' literature, there is a commonly accepted view that although a pedagogy can foster or impede a style, different learners can adopt different strategies and styles in different tasks (Hartley, 1998). In the case of visual programming, pedagogy can foster instruction in the visual dimension. While the linkage among this type of instruction and a student's learning preference may appear logical, it is of high importance to find out to what extent their enjoyment of the specific programming environment correlates to students' possible tendencies to prefer learning visually.

In this research, I am particularly interested in whether or not there is a correlation between the likeability/enjoyment and preference to use a visual programming environment, such as Scratch, and the learning style preference of students, especially in the input and the processing dimensions which are assessed only in the Felder and Soloman's Index of Learning Styles (ILS) instrument.

Furthermore, the ILS has been utilised in several computer science studies (Chamillard & Karolick, 1999; Thomas *et al.*, 2002; Allert, 2004; Zualkernan *et al.*, 2006; Da Silva Carmo *et al.*, 2007; Chen & Lin, 2011) in order to identify possible correlation between students' learning styles and their performance.

Given the above, the Index of Learning Styles (ILS) will be the instrument of choice for this research project.

Relevant previous research on learning how to program in relation to learning styles revealed that most students have visual learning styles (Kuri *et al.*, 2002; Ratcliffe *et al.*, 2002; Allert, 2004; Gomes & Mendes, 2008; Santos *et al.*, 2010; Tsai *et al.*, 2011) but there is no current research examining the relationship of learning styles with the preference of using visual programming environments. Therefore, considering the learning styles of students in the context of their perceived preference for visual

technologies in computer programming education is one of the two main subjects of this research.

In the following section, I will introduce the second subject of this research, which is student motivation in the context of visual technologies.

3.3 Motivation and Self-Determination

As one of the research questions of this study is to identify whether or not visual programming environments affect student motivation to learn to programme, I will review the fundamental concepts and theories of motivation in order to determine what motivation is, which its primary determinants are, and how educational activities can cultivate it. Furthermore, I will explore what has been written about the motivational process itself and how it links to programming education outcomes. Lastly, I will review existing assessment tools that measure student motivation within the context of visual programming environments and identify which motivational aspects to target.

A theory of motivation is concerned with those factors that affect people to initiate behaviour (Dweck, 1999). The first theories of motivation were based on the idea of motives being driven by the need for achievement or affiliation, or by rejection, and explored how each behaviour is initiated and crafted by these motives. Motives, in this respect, are seen as driving forces influenced by interest and ability and shaped by experiences (Murray, 1938).

Dweck (1999) argued that motives alone are not enough to ensure outcome. Goals that people set out to pursue also affect the degree and the intensity of their behaviour towards their attainment. In that respect, Elliot and Harackiewicz (1996) related the motives with the goals, and argued that, within the learning process, the "achievement motive" can be used to predict learner orientation towards setting "achievement goals". Achievement motivation is defined as behaviour in which the goal is to develop a high ability and/or demonstrate it to one's self or to others (Nicholls, 1984). High ability in this sense can be judged either against the learner's own past performance or compared to the performance of others in the same task. Achievement motivation is affected by goal orientation (why the learners engage in the task) and self-efficacy (personal judgements of ability to perform), and is related to perceptions of task difficulty and task value (perceptions about task importance, relevance and utility) (Pintrich, 2000).

Goal orientation refers to the learner's general attitude towards the task as a whole, and it manifests itself in two complementary forms: intrinsic and extrinsic, which are based on the traditional views of motivation (DeCharms, 1968; White, 1959). Intrinsic goal orientation concerns the degree to which the student possesses a real interest in the task with the aspiration to increase his/her knowledge in the subject for reasons such as challenge, curiosity, control and fantasy (Malone & Lepper, 1987). Extrinsic goal orientation complements intrinsic goal orientation, and *"concerns the degree to which the student perceives herself to be participating in a task for reasons such as grades, rewards, performance, evaluation by others, and competition"* (Pintrich, Smith, Garcia, & Mckeachie, 1991, p.10). In early childhood, people have the freedom to engage in more intrinsically directed tasks, whereas, when social responsibilities increase with age, they increasingly engage with less intrinsically interesting tasks.

Gottfried (1985) developed the Children's Academic Intrinsic Motivation Inventory (CAIMI) of 122 questions to relate intrinsic academic motivation to academic achievement, in terms of mastery, curiosity, task persistence and learning of challenging topics, based on the works of Deci (1975; 1978), Harter (1981), Pittman, Emery and Boggiano (1982), Nicholls (1984), and others. The results showed that, although there is a high correlation between academic achievement and intrinsic motivation, there is an even higher correlation between academic achievement and perceptions of self-efficacy. Other studies also showed that there is a limit of achievement that motivated students will reach, as other factors that affect academic achievement come into play, such as ability, quality of instruction, the educational environment itself, and educationally relevant aspects of the home environment (Uguroglu & Walberg, 1979).

Self-efficacy of learning and performance refers to a learner's ability to properly gauge his own capabilities of successfully performing an academic task (Schunk, 1991). *"Perceived self-efficacy is defined as people's beliefs about their capabilities to produce designated levels of performance that exercise influence over events that affect their lives. Self-efficacy beliefs determine how people feel, think, motivate themselves and behave. Such beliefs produce these diverse effects through four major processes. They include cognitive, motivational, affective and selection processes"* (Bandura & Wessels, 1994, p.1).

Control of learning beliefs refers to the degree of control that learners believe they possess regarding their learning ability and learning outcomes. Bandura and Wessels (1994) defined control of learning beliefs as the ability to actively affect one's

motivation, cognition, affect, and behaviours. In his study of the role of reinforcement in student performance, and thus in the amount of knowledge gained, he distinguished the control on reinforcement into two categories: internal or external. Belief in internal control concerns the degree to which a learner perceives that all outcomes depend on his own actions, whereas external control concerns the degree to which a learner thinks that "powerful others", such as luck, fate and chance can affect the outcome of his actions and his ability to complete a task (Rotter, 1966; 1990).

Task value refers to the learner's personal interest in a given task and is driving his/her own beliefs about how interesting, important, useful, valuable and meaningful the task is. Pintrich (2000) also argues that a task value viewed as being high leads to greater levels of involvement towards the completion of the task.

Findings from the literature have demonstrated that students' intrinsic motivation and beliefs of their self-efficacy, as well as the perceived value of a topic or an activity, generally constitute good predictors of performance and achievement (Zusho, Pintrich, & Coppola, 2003). A primary concern shared by educators, myself included, is how to enable students to value and self-regulate activities which are not designed to be intrinsically motivating, and to carry them out on their own (Ryan & Deci, 2000) or how to modify educational activities so as to be intrinsically motivating. In my experience, the most successful students seem to employ self-regulated strategies to direct their learning.

Pintrich (2000) provides an overall definition of self-regulated learning as "an active, constructive process whereby learners set goals for their learning and then attempt to monitor, regulate, and control their cognition, motivation, and behaviour, guided and constrained by their goals and the contextual features in the environment." Zimmerman (1990), in his attempt to define self-regulated learning, identified three main characteristics that self-regulated learners typically display. First, they use "self-regulated learning strategies" in the areas of metacognition, motivation and behaviour and are active participants in their own learning. Second, they use "self-oriented feedback", which enables them to keep track of their learning effectiveness (self-monitoring) and discover their problem areas (self-evaluation). Metacognitive theories focus on understanding the processes of self-monitoring and self-evaluation towards the selection of appropriate strategies for learning (Borkowski *et al.*, 1990). As a result, learners can take necessary actions to select, structure and create their learning environment so as to match their learning style. In that sense, self-regulated learners can be considered as self-motivated. On the other hand, this self-regulation

process is activated from a number of self-motivational constructs both intrinsic and extrinsic (Zimmerman, 2008).

Malone and Lepper, after reviewing the literature, presented a logical taxonomy of four kinds of intrinsic motivations which are present in any learning situation: challenge, curiosity, fantasy and control (Malone & Lepper, 1987). These motivations were used in testing learning environments and especially those that incorporate games. Student perceptions about task enjoyment, interest, involvement and self-efficacy were used to assess the effectiveness of instruction using educational games (possibly fun) by Lepper and Cordova (1992) and Garris *et al.* (2016). Findings from their studies suggest that motivational and cognitive benefits can be gained from the use of relatively small motivational embellishments in educational activities, aiming to increase students' intrinsic interest. Lepper and Cordova (1992) pointed out, that adding motivational embellishments to an activity will possibly have positive effects on learning if there is a "match" between those actions required by learners to assimilate the material and those required to enjoy an activity. For example, the goal to win the game should be supportive of the goal to learn the material. They also identified areas in which adding more "seductive" details to learning activities might draw the attention of learners away from the main concepts being taught and have a negative effect on learning: "it appeared to have pursued motivation in expense of learning" (Lepper & Cordova, 1992).

Keller (1987) developed a model with guidelines on how to create instruction that stimulates motivation, based on the theory of motivation and instructional design. The Attention, Relevance, Confidence and Satisfaction (ARCS) model is based on four major conditions that need to be met in order for learners to become, and remain, motivated. This model provides instructional designers with a set of strategies that target the four conditions. The most basic motivational concern is attention. Gaining learner attention is considered a relatively easy task, but sustaining that attention seems to be the challenging component. Keller proposes inquiry and the use of games or simulations to target learner participation. The second condition within the ARCS model is perceived relevance. Strategies to stimulate relevance include relation to student interests, presentation of worth and usefulness of the activity and relation to past and future skills. Confidence, or expectancy for success, is the third condition. Confidence relates to control of learning beliefs mentioned above. Strategies to stimulate confidence include the presentation of material using an increasing degree of difficulty, clear and realistic goals and association of effort with success. Finally, the satisfaction condition targets a learner's intrinsic motivation. Strategies to invoke

feelings of satisfaction include praise, personal attention, avoidance of threats, regular informative feedback and frequent reinforcements.

Rieber *et al.* (1992; 1998) focused on the evaluation of micro-worlds, in which learning is achieved through exploration and discovery, and on the contribution of "serious-play" to student motivation and performance. They found that when educational activities are designed in such a way that serious play can be incorporated, there is an increase in intrinsic motivation and reflective knowledge construction. When learners receive a great amount of enjoyment from their experience, they are willing to engage with the task and spend time and energy. On the other hand, Bloom and Hanych (2002) argued that approaching learning only from the "fun" perspective may result in trivialising the learning process.

Learning computer programming requires a high degree of self-regulated learning, prolonged motivation, sustained willingness to practice and even enjoyment, to a degree, of the whole process in order to overcome the complexity and the abstraction of computer concepts (which will be discussed in the following chapter). Visual programming environments, such as Alice, Scratch, APP-Inventor and other block-based programming tools, seem to be a promising alternative to more traditional text-based programming. Their creators claim that these visual programming environments can provide students with a more fun and learner-friendly approach to programming, eliminating the mundanities of syntactical errors and decreasing the overall layer of complexity.

Based on research findings mentioned above, there seems to be a connection between intrinsic motivation, self-regulated learning and "fun" educational activities (based on the concepts of exploration, play and learn), which I will explore further in this context in the area of teaching and learning computer programming. Unfortunately, since motivation is such a complex and multi-dimensional concept, it is very difficult to measure (Ball, 1977) and indeed be absolutely certain about the results. Associating results from a validated tool with data collected from interviews and class observations is more likely to help researchers identify the general categories of student motivation. To explore and qualitatively determine - to the degree possible - the current level of motivation of information technology students at XYZ college, it is essential to use a reliable and validated instrument.

One widely used and validated instrument to assess student achievement motivation is the Motivated Strategies for Learning Questionnaire (MSLQ), developed by Pintrich (2004). MSLQ is based on the cognitive view of motivation. Pintrich's instrument, based on theoretical, empirical and statistical analysis following a 10-year research

period and continuous refinement of the questionnaire, included 81 items in the instrument's final version. Items and scales have been tested for internal consistency, with coefficient computation and factor analysis, and for predictive validity through correlations with course performance, producing statistically-validated results (Pintrich *et al.*, 1991). MSLQ is comprised of two sections. The first part assesses motivation through components of value (intrinsic and extrinsic goal orientation, task value), expectancy (control of learning beliefs, self-efficacy of learning and performance) and affectiveness (test anxiety). The second part assesses learning strategies through components of cognitive and meta-cognitive strategies (rehearsal, elaboration, organisation, critical thinking and self-regulation) and resource management (time, effort, peer support and help-seeking). MSLQ has also been used in computer education research to assess the effect of various programming environments on student motivation (Bergin & Reilly, 2005; Dillon, 2012; Nikou & Economides, 2014; Erol & Kurt, 2017).

Another instrument which has been used to assess student motivation, specifically targeting science students, is the Science Motivation Questionnaire (SMQ-II), developed by Glynn *et al.* (2009). Science Motivation Questionnaire II, based on the social cognitive theory, assesses motivation in the components of value (intrinsic and extrinsic grade and career motivation), expectancy (self-efficacy) and self-determination (Glynn, 2011). Glynn *et al.* (2011) have also created a discipline-specific version of the questionnaire for chemistry, biology and physics, simply substituting the word science with chemistry or biology or physics. In all its interchangeable adaptations, SMQ-II has been tested for reliability, internal consistency, and construct validity, and findings indicate that it validly provides a profile of the components that contribute to a student's motivation (Glynn *et al.*, 2011).

Computer science is also concerned with the ability to define models, to make predictions about the behaviour and vulnerabilities of these models, implement them and validate the performance of computer systems and software. In that sense, I believe that SMQ-II can also be administered to assess students' motivation in computer programming courses.

Despite their differences, there are many similar inquiries between the two instruments described above (MSLQ and SMQ-II) and especially in those questions which involve intrinsic and extrinsic motivation, expectancy and self-determination/regulation (see Figure 3.4). It is noted that, in regard to prior research

on motivation in academic college settings, the majority of the studies have been conducted using the MSLQ tool.

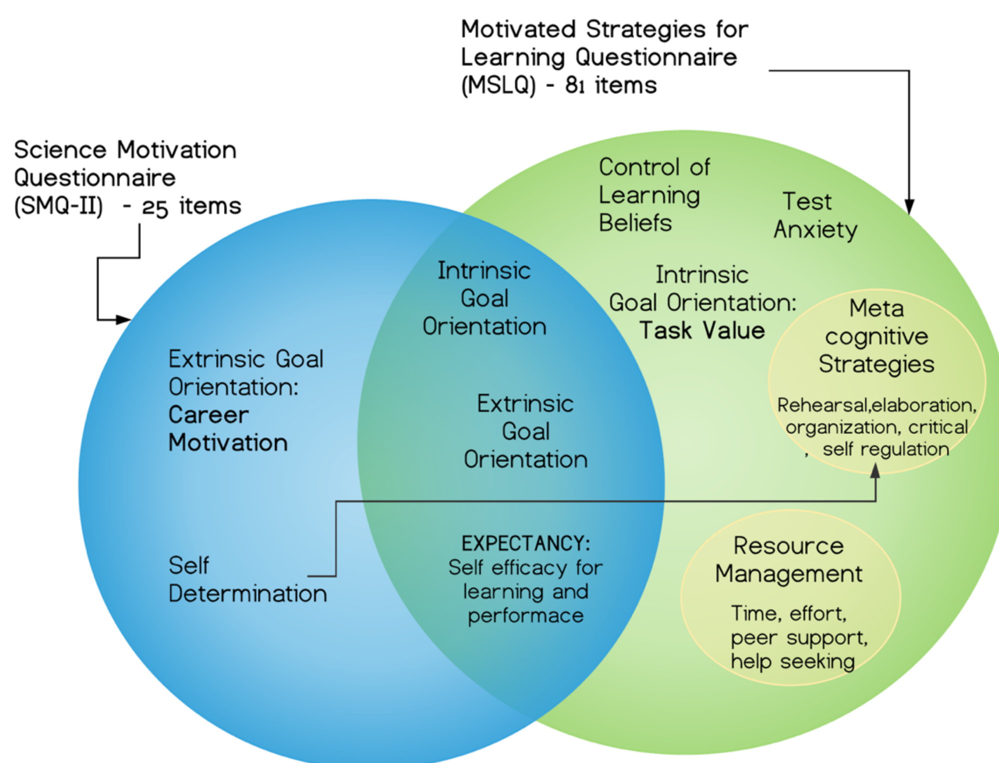


Figure 3.4: Differences and similarities between SMQ-II and MSLQ

The choice of approaches and concepts discussed in this chapter were made on the basis that they align closely to teaching and learning computer programming. There are more recent approaches, that could have been considered, such as embodied cognition, distributed cognition, multimodality, knowledge development and socio-materiality, but these approaches do not interplay the role of visual design, technology mediation and interpretative processes of engagement with technology or computing programming specifically. For example, Bergin & Reilly (2005) used the intrinsic and extrinsic goal orientation scales of MSLQ, to analyse the relationship between student motivation and programming performance. Nikou & Economides (2014) used the intrinsic, extrinsic goal orientation, task value, control of learning beliefs and self-efficacy scales of MSLQ, to examine the effects of VPEs (Scratch and APPInventor) on students' motivation. Erol & Kurt (2017), examined the effect of programming instruction with Scratch on the motivation and programming achievement, using an adapted version of MSLQ with 2 subscales: motivation and learning strategies.

3.4 Conclusion

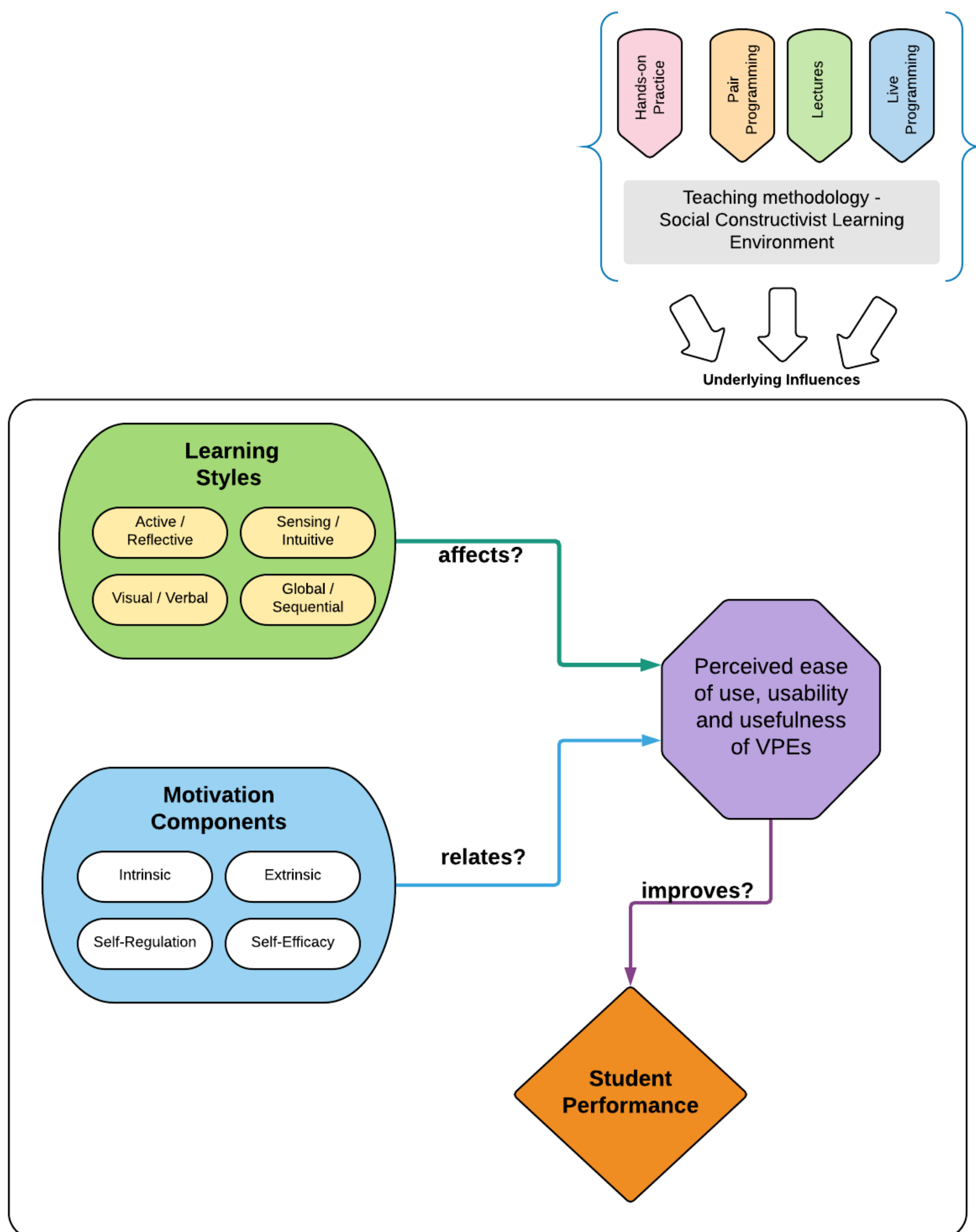
In this research study, in order to address the research question "How do students' motivations for learning programming relate to their perceptions about programming and their attitudes about visual programming environments?", students' overall motivation towards programming needs to be assessed. A gap in the literature has been identified – and a tool to assess student motivation in learning how to program has been identified. To that end, questions from both instruments (MSLQ and SMQ-II) were selected and adapted to match a programming course and a new survey instrument was developed, tested and administered. The new instrument targets the components of value (intrinsic and extrinsic motivation), self-efficacy, self-determination and self-regulation. There is already a history of studies that have used customised instruments to assess motivation, but again targeted the same motivational components: intrinsic, extrinsic, and achievement (Jenkins, 2001; Zainal *et al.*, 2012).

In Chapter 6, Research Design and Methodology and more specifically in the section "Development of the Questionnaire Survey Tool", I will analyse the process and the rationale behind the selection and the adaptation of questions from both MSLQ and SMQ-II instruments.

The purpose of this study is to find whether or not there is a correlation between the students' learning preference (using four scales/dimensions of the Felder-Silverman model) with their perceived preference for a visual block-based programming environment and whether or not this preference influences their motivation towards the achievement of the module's learning outcomes. Creating a learning environment by incorporating "fun" educational activities, where instruction might trigger intrinsic motivation and self-regulated learning, is one of my personal aims as an educator.

As discussed earlier in this chapter, literature supports the theory that motivation, learning styles and strategies are associated with achievement, but is there a relationship between a student's learning style preference and the perceived acceptance of a VPE? Does motivation to learn programming relate to the acceptance of a VPE? And finally, does the use of a VPE improve student performance?

As such, the conceptual framework of this research is summarised in Figure 3.5.



Research conceptual framework

Figure 3.5: Conceptual research framework

Chapter 4 Teaching and Learning Computer Programming

In Chapter 2 , I discussed programming and programming paradigms, as well as different types of programming environments. In this chapter, I explore the cognitive aspects of programming and the basic programming constructs typically taught to novices. Then, I focus this literature review on the difficulties that students face while learning how to program and I compare them with findings obtained from programming modules which I have personally taught in the past. A short history of educational programming environments follows, and the chapter concludes with the exploration of the possible power of visualisations in learning facilitation, in accordance with the overall theme of this study.

4.1 Cognitive Aspects of Programming

Pennington (1987b) defines computer programming as "a complex cognitive task composed of a variety of subtasks and involving several kinds of specialized knowledge" and Du Boulay adds that the "ability to see a program as a whole, understand its main parts and their relation is a skill which grows only gradually" (Du Boulay, 1986).

Teaching computer programming has traditionally been considered a challenging endeavour, since mastering the subject matter requires a combination of different skills (syntactic, conceptual, problem solving and strategic), as well as considerable engagement and persistence. Kolling (1999) partially attributes this difficulty to the introduction of numerous abstract concepts from the very beginning of the teaching process.

In order to understand the process of learning how to program, we must first consider the skills required to cognitively structure programming knowledge and apply it in practice. Linn and Dalbey (1985), Fay and Mayer (1988) and Pears *et al.* (2007) have identified a chain of cognitive accomplishments required to achieve the learning outcomes expected in programming courses: a) learn the language syntax and rules; b) comprehend what an existing program does and be able to modify, extend and/or debug it; c) understand

and conceptualise a given problem and learn to design programs (problem-solving skills); d) translate a solution into code, execute it and check for correctness; e) build upon prior knowledge and/or experience and acquire new programming skills; f) make appropriate generalisations and be able to apply them to other programming languages and environments; and g) create strategies and a "catalogue" of ready-made solutions to problems solved in the past. All of the aforementioned aspects of computer programming require not only knowledge of the specific programming language syntax rules, semantics and programming conventions, but also: relative degrees of experience in real-world problem domains, such as accounting, finance, sales, statistics, banking, or even physics; familiarity with numerous design strategies and re-applicable components and solutions; knowledge of computer features that impact program performance and implementation; and awareness of the personas of a computer program's intended users.

In investigating the difficulties that novices face when learning how to program, troublesome activities can be grouped using the three distinct categories of programming skills identified in the literature, those relating to: programming knowledge, mental models and strategy (programming plans) (Bonar & Soloway, 1985; Soloway, 1986; Norman, 1987).

Programming Knowledge: represents the knowledge that *"allows novices to write some parts of a program correctly"* (Bonar & Soloway, 1985). The minimum skills required to complete a basic programming task is to know the syntax of a programming language (syntactic knowledge), to understand the semantics behind the syntax and to comprehend a program (program comprehension).

Syntactic knowledge is precise, very detailed, and rigid, and pertains to a given programming language. Semantic knowledge, on the other hand, is independent of the specifics of a programming language and may range from low-level basic notions, for example what an assignment command does, to high-level strategies, for example how to use recursion. Higher levels and broader degrees of semantic knowledge can be created from building upon and anchoring concepts via experience and is required to create more complex programs (Shneiderman & Mayer, 1979).

Program comprehension is the skill which enables programmers to read and analyse the source code of a program, understand its intent and implementation approach, and formulate an overall description of what it does. According to Soloway and other authors program comprehension is the programmer's ability to recognise plans in the code, reverse engineer these plans to identify subgoals or system components and finally create a high-level representation of the system's functionality (goals) by locating their inter-relationships (Soloway *et al.*, 1983; Soloway & Ehrlich, 1984). In line with Soloway, Pennington (1987a) identified two main categories of program comprehension: procedural (language structure); and functional (goals of the program). The first category is relevant to the program text-base (commands) and the second category relates to the domain model (the goals of the program).

- Procedural comprehension includes the knowledge of operations, control flow and data flow. "Operations" include all the actions the program performs at source code level, such as declaration of variables, assignment of values to variables, comparison of variable values. "Control flow" involves the sequence (often conditional and dynamic) of command execution in a program, while "data flow" relates to all the intermediate transformations and manipulations that data undergo from their initial state through to the final program output.

- Functional comprehension involves the understanding of the program state and function. "State" reflects the relationships between the execution of an action and the state of the program at a specific point in time. "Function" involves the relation between the main goal of the program and the hierarchy of sub-goals necessary to achieve it. Functional comprehension thus relates closely to the semantic knowledge mentioned above.

A complete computer program is formed by purposefully combining numerous advanced language constructs to deal with abstract entities (i.e. pointers, iterators, arrays). Novice programmers, having little or no past experience to draw from, typically face difficulties relating to both procedural and functional comprehension (Sajaniemi, 2002).

Similarities often noted between programming language commands and natural (spoken) language have been found to be another reason contributing to a novice programmer's difficulty. Bonar and Soloway (1985) refer to these similarities as **bug generators,** because novices use *"pre-programming"* natural language knowledge to create "patches" to their fragmented programming knowledge. Two kinds of such similarities are identified: functional; and surface.

Functional similarities exist because both the natural language and programming commands are concerned with repeated actions, choice between conditions and counting, for example. Surface similarities exist because most programming languages share many words with natural language. There are many common lexical entities in the two plan-sets which can generate confusion between surface and functional links. For example, the word "while" in natural language, can be used:

a. as a conjunction with the meaning of "during the time that; or at the same time as"

b. with the meaning of "despite the fact that; although"

c. with the meaning of "length of time"

This kind of semantical difference is unusual in a programming language. A more typical use of the word "while" in a programming language is one in which a loop condition gets discretely tested once per loop iteration. For example, in the following piece of code:

```
while (counter < 10) {
      print "Hello";
      Counter++
      }
```

the condition will be tested first, then the code inside the brackets will be executed, the counter will increase, and the condition will be tested again, and so on.

The surface link between the divergent uses of the word "while" in natural and programming language might lead a novice programmer to infer similar

semantics. Similarity in semantics might “block” his/her ability to write correct pieces of code.

Another statement that has been found to cause programming misconceptions is “if” (Pea, 1986). For example, in the sentence *“if you want to have lunch, tell me so…”* there is an assumption of a duration, whereas the programming construct “if” singularly evaluates a condition only at the point of execution of the specific statement. I have personally noticed that students often use the “if” statement to repetitively validate user input, thinking that the computer will keep asking for input as long as the user-entered value does not satisfy the stated condition.

To summarise, novice programmers tend to:

- concentrate more on the syntax of the language rather than the process and the semantics;
- not fully comprehend what a program does;
- mix up natural language (often referred to as pre-programming knowledge) with programming language commands.

Mental Models: Gentner and Stevens argue that *“A mental model is a representation of some domain or situation that supports understanding, reasoning, and prediction”* (Gentner & Stevens, 2002, p.9683). Holt and Schultz add that the basic components of a mental model structure are the fundamental elements of knowledge and the relationships formed between them (Holt & Schultz, 1987). Numerous researchers have argued that the formation of “valid” mental models is crucial to understanding the functionality of computers. The main purpose of a mental model, according to Norman (1987), is to enable a person to predict the operation of a target system. He also supports that mental models could be used to explain human reasoning about physical systems, such as the interaction of people with computers and other devices. More specifically, people create mental representations of objects, situations and information in the world in general, and then they use these internal representations to understand, explain, and predict the behaviour of external systems.

Mental models play a significant role in program development, program comprehension, program modification and debugging and can be affected by program structure and content. Holt and Schultz support that mental models of experienced programmers, when viewed as hierarchical structures, may vary in quality, depth, width and complexity from those of their less experienced counterparts, due to their increased knowledge base (Holt & Schultz, 1987). As mental models are naturally evolving, expert programmers tend to form abstract and more conceptual representations, which consequently enable them to make useful and valid generalisations.

On the other hand, novice programmers face significant challenges in constructing a mental model of how the programming language commands interact with the physical computer system. The term "notional machine" was introduced by Benedict du Boulay to describe *"the general properties of the machine that one is learning to control*" (Du Boulay, 1986, p.57). The notional machine is *"an idealised, conceptual computer whose properties are implied by the constructs in the programming language employed"* (Du Boulay & O'Shea, 1981, p.237) and serves the purpose of helping novices understand what is going on inside the computer during program execution. The concept of a notional machine has nothing to do with an accurate model of computer hardware functions, but with an abstraction, or rather a simplification, of how a particular programming language stores and processes information. Du Boulay associated student difficulties in learning how to program with an inability to understand and describe the machine which they are learning to control, and he proposed that teachers follow the notional machine strategy to help tackle this issue. In that case, the notional machine should satisfy two basic principles: it should be conceptually simple - both functionally and syntactically; and should provide ways for the learner to observe some processes as they happen (Du Boulay & O'Shea, 1981). An incomplete model of the relationship between the behaviour of the physical machine and the properties of the notional machine will result in an incorrect and insufficient understanding of programming concepts and vice versa.

To summarise, the main troublesome areas for novices in this category are:

a. Lack of a detailed mental model of what the computer does when a program executes (Adelson, 1984; Winslow, 1996);

b. Unclear understanding of how the underlying physical machine hardware relates to the properties of the programming language's notional machine.

Strategy: represents the set of tactics and plans that allow a programmer to break down a problem into smaller parts and understand the importance of each one, as well as their interaction, based on a higher-level plan or goal. A skilled programmer has developed strategic skills, goals (intentions) and plans (techniques for realising these intentions) (Letovsky & Soloway, 1986), which allow for efficient planning, problem decomposition, algorithmic design and debugging. Soloway refers to these plans as libraries of "*stereotypical, canned solutions*" (Soloway, 1986), composed from reusable patterns of data flow and control flow which follow rules of programming discourse. These rules - analogous to discourse rules in a human conversation - specify conventions that create expectations in the minds of expert programmers, which other programmers are "expected" to follow. Novice programmers therefore should first master following simple coding rules and master simple plans (for example: how to obtain input, how to print the elements of an array, and how to create a method or procedure), before they are able to move on to more complex coding endeavours.

A major research debate can be found in the area of mastery learning - that is, can a complex skill be decomposed into smaller component skills which can be learned and addressed separately? In relation to computer programming, Anderson and Corbett (1995) and McCane *et al.* (2017) found that there is some correlation between mastering isolable coding skills and an increase in programming performance, but Anderson and Corbett failed to mention the complexity of the programming problem and the type and complexity of the isolable skills. Carpenter *et al.* (1990) performed a test to measure intelligent behaviour and postulated that, according to their findings, students that performed well in the test showcased the ability to induce a correct strategy in order to decompose problems into smaller manageable sections; the ability to manage a hierarchy of goals and sub goals which resulted from problem decomposition; and the ability to form generalisations. All of these abilities are also recognised as vital skills of expert programmers (Anderson & Corbett, 1995).

To summarise, the main troublesome areas for novices in this category are:

a. Incomplete libraries of "stereotypical solutions" and limited use of rules of programming discourse (Soloway, 1986).

b. Limited ability for problem decomposition, modularisation and generalisation (Carpenter *et al.*, 1990) which can be related to a not-yet-established systematic methodology (Rugaber, 2007).

c. Difficulty in the formation of algorithms to solve a given problem.

d. Limited debugging skills (Soloway, 1986).

Relevant research on teaching and learning how to program has shown that learning a programming language requires a student to deploy both a "deep" and a "surface" approach to learning (refer to Chapter 3 on learning styles and approaches).

The fact that a programming language can be memorised does not imply that a student can thus construct programs. In order for a student to be efficient and proficient in programming, a student should learn how to think in computer terms, implement abstraction and modularity, construct algorithms and know where to look for "surface" information such as syntax rules. Associating this fact about programming to learning approaches, and the definitions provided by Entwistle and Tait (1990) are representative: "*deep approach is associated with a holistic style and intrinsic motivation (interest in the subject matter itself) to form a meaning orientation. Surface approach goes with a serialist style (a narrow, cautious stance relying on evidence and logical analysis) and fear of failure within are producing orientation, while strategic approach indicates a use of both deep and surface approach supported by a competitive form of motivation (need for achievement) combined with vocational motivation within an achieving orientation*" (p.171).

Applying the findings mentioned above, the following conclusion can be reached: students that focus on coding and syntax rules employ a surface approach to learning how to program, as opposed to students that focus on problem-solving using the programming language syntax rules only as a means to reach their goal. These students employ a deep approach to learning.

Computer programming involves so much more that learning a programming language and producing lines of code. Programming is about producing digital artefacts; it incorporates abstraction and creativity; involves implementing ideas, understanding human behaviour and solving problems. If programming is viewed in this broader sense, then Papert's (1980) view that programming can enhance students' thinking skills which can be applied to other disciplines as well provides a first definition of computational thinking (CT).

Wing (2006) stated that "Computational thinking is reformulating a seemingly difficult problem into one we know how to solve, perhaps by reduction, embedding, transformation, or simulation" (Wing, 2006, p.1). She also argued that CT means to be able to engage in five cognitive processes: problem reformulation; recursion; decomposition; abstraction; and testing, with the goal to solve problems efficiently.

Taking into consideration that the majority of students attending XYZ college come from Greek high schools, I should stress that at the time of this writing, the Greek high school curriculum did not include Computational Thinking (CT) as a subject, although it forms the basis for formulating solid problem-solving techniques.

My teaching methodology for the 'Introduction to Programming' module aims to provide a structured context for student learning, which commences with an introduction to CT using problem-solving techniques (decomposition and abstraction) using pseudocode and Scratch. Then the module advances to Java programming language syntax and rules. In this module, teaching programming with Java focuses first on how to code smaller tasks (create a class with a main method, produce simple output, declare variables) and finally proceeds with the process of creating a complete program: understanding the inputs and the outputs; outlining the processing requirements; and finally creating a program by using reusable pieces of code.

4.2 Troublesome Programming Constructs and Skills

The fact that students face academic difficulties when learning how to program has long been identified and is one of the major concerns amongst computer science educators. To date, numerous studies have tried to identify and categorise types of difficulties, errors and misconceptions of students learning

computer programming, aiming to improve instruction and learning (Kaczmarczyk *et al.*, 2010; McCall & Kölling, 2015; Veerasamy *et al.*, 2016; Bosse & Gerosa, 2017).

Throughout my own teaching career, I have always been interested in exploring which programming concepts are considered by students at my college to be more challenging. To do so, from Spring Semester 2013 until Spring Semester 2018, I conducted a web-based survey on the various programming concepts that students registered in the Introduction to Programming module at XYZ College found more challenging. Students were asked to rate each concept on a scale from 1 to 5, according to their perceived difficulty level surrounding each concept (1=extremely easy, 2=somewhat easy, 3=neither easy not difficult, 4=somewhat difficult, and 5=extremely difficult). The results of the study are presented in Table 4.1, ordered by the most commonly reported troublesome concept.

Rank	Concept Ranked By 105 Students	Mean Score [in a Scale 1 - 5]
1	Using Arrays	3.4
2	Defining Methods	3.0
3	Displaying Formatted Output	2.7
4	Understanding the steps required to solve a programming problem and writing the pseudocode	2.7
5	Validating User Input	2.3
6	Using Exceptions	2.3
7	Reading from and Writing to Files	2.2
8	Tracing a program (finding out what is the value of a variable at a given time in a program)	2.1
9	Using the WHILE loop	2.1
10	Transferring the pseudocode into a program	2.1
11	Using the FOR loop	2.0
12	Obtaining Input from the User	1.9
13	Writing IF statements	1.7
14	Declaring variables with correct naming standards and datatypes	1.6

Table 4.1: Java programming: Difficult concepts – student perceptions
(105 undergraduate students, June 2013-May 2018)

The same web-based questionnaire was administered to 34 professors teaching introduction to programming modules, during the same period, in a number of universities in Greece. Of the participating professors, seven (7) were from XYZ college and twenty-seven (27) from other universities. The results are presented in Table 4.2.

Rank	Concept Ranked By 34 Professors Mean Score [Scale 1 - 5]	27 Professors from other universities	7 Professors from XYZ College
1	Understanding the steps required to solve a programming problem and writing the pseudocode (problem solving)	3.79	3.86
2	Using Arrays	3.74	3.71
3	Transferring the pseudocode to a program	3.71	3.71
4	Defining Methods	3.65	3.57
5	Using Exceptions	3.62	3.71
6	Reading from and Writing to Files	3.38	3.43
7	Validation Input from the User	3.35	3.57
8	Using the FOR loop	3.29	3.29
9	Using the WHILE loop	3.26	3.29
10	Tracing a program (finding out what is the value of a variable at a given time in a program)	2.91	2.86
11	Writing IF statements	2.56	2.57
12	Displaying Formatted Output	2.44	2.43
13	Obtaining Input from the User	2.29	2.43
14	Declaring variables with correct datatype	2.12	2.14

Table 4.2: Java programming: Difficult concepts – professor perceptions
(34 professors, June 2013-May 2018)

I should also point out that the mean score (3.18) of perceptions of the 7 professors from XYZ College is not found to be statistically different from the mean score of the perceptions of the professors of other Universities (3.15). A t-test derived a p-value of 0.8821 with a 99% significance interval, which leads to acceptance of the null hypothesis that the means are equal. The resulting ranking was generally the same with two slight exceptions around the

difficulty of using exceptions and validating user input, which were ranked a bit higher by XYZ College's professors.

From the rankings obtained from this survey, I found some differences but also some similarities between the concepts that students find as more difficult and the ones that professors consider as more troublesome. An example of such a difference is "transferring the pseudocode to a program", which professors consider a rather difficult concept with a mean score of 3.71, whereas students rate the same concept with a mean score of 2.1 (10th in the Rank). An example of a similarity is in the "use of arrays", which both students (3.4) and professors (3.7) rate as a difficult concept. Declaring variables and datatypes is another example of a similarity in the perceptions. Both students and professors perceive it as being a rather easy concept.

Another very interesting finding is that students and professors have quite different perceptions on how difficult a concept is. Students' mean difficulty score for all concepts is 2.29 on a scale from 1 to 5 (1=extremely easy, 2= somewhat easy, 3=neither easy not difficult, 4=somewhat difficult, and 5=extremely difficult), whereas the professors' mean score is 3.15. See Figure 4.1 for the difference of students' and teachers' responses concerning the programming concepts.

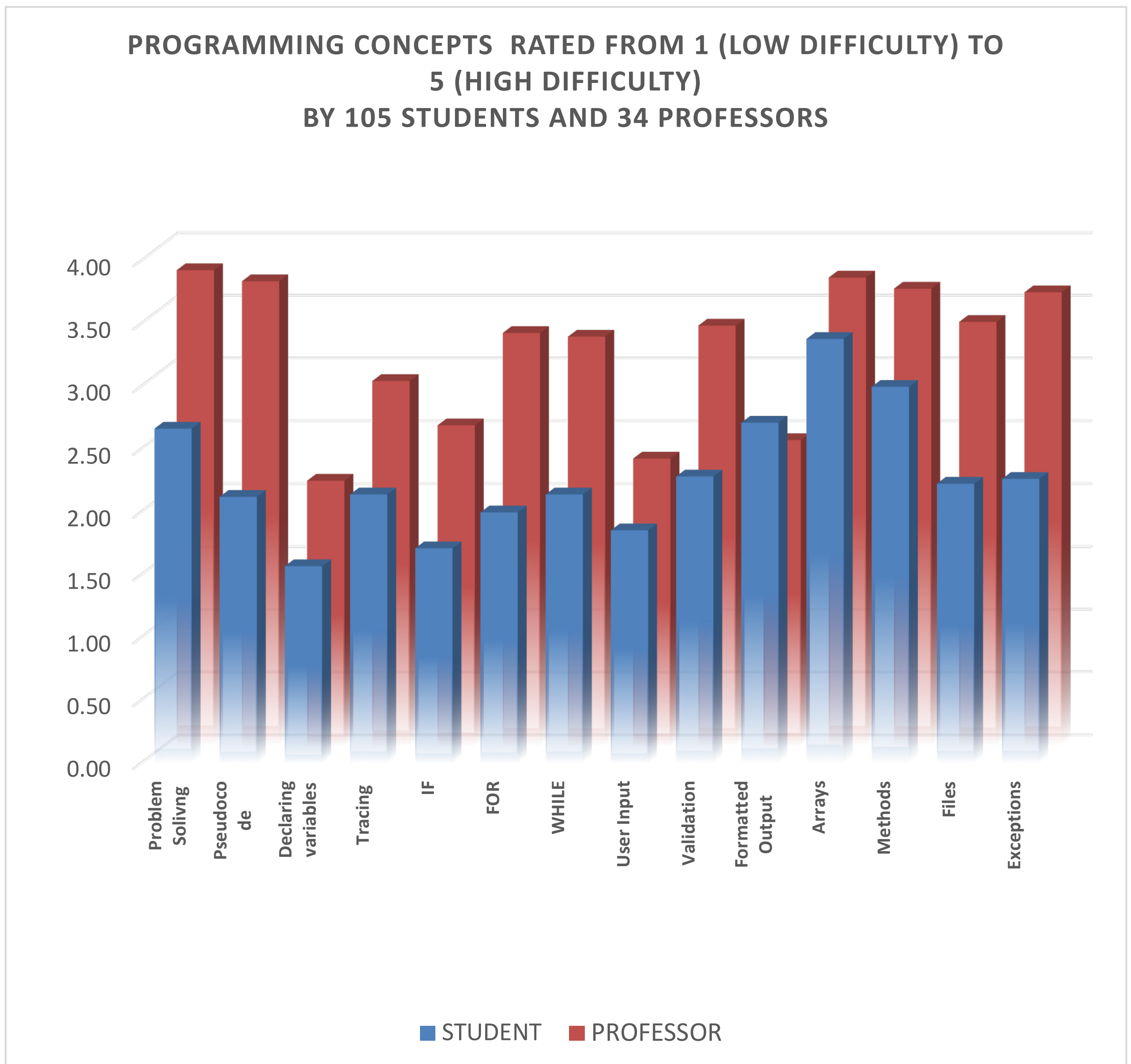


Figure 4.1: Programming concepts rated by 105 students and 34 professors

A Welch two-sample t-test in R produces a p-value of 0.0002852, which shows that there is a statistically significant difference between the two means. This leads us to think that students in this sample tend to find programming concepts as being easier than how the professors would regard them. This underestimation of their understanding might be one reason for not performing very well in their examinations. For example, while professors think that transferring a pseudocode to a program is at the top of the list of the perceived difficulties, students rank it as a “somewhat easy” concept.

The results obtained from this study align with results obtained from a similar study performed by Lahtinen *et al.* (2005), where the average student perception about the difficulty of programming concepts (mean 2.8) was smaller than that of instructors (mean 3.5). In another study in 2002, Milne and Rowe also noticed a difference in the mean scores of students and tutors. They claimed that students *"may believe they understand a topic, but upon detailed examination or one-to-one querying from a tutor it turns out that they are often wrong in their belief"* (Milne & Rowe, 2002, p. 58).

The last question in the study administrated to students was to select from a list of predefined statements the one that best matched their ability to understand the concepts taught in the course and produce correct code by implementing them. The results are presented in Table 4.3.

Ability to understand and write correct code	Student PCT
1. understand the concepts and manage to write correct code	6%
2. understand the concepts and usually manage to write correct code	10%
3. understand the concepts but only sometimes manage to write correct code	47%
4. somewhat understand the concepts but do not know how to write correct code	34%
5. do not understand the concepts and do not know how to write correct code	3%

Table 4.3: Java Programming: Ability to understand and write code – student perceptions (105 undergraduate students, June 2013-May 2018)

A finding from this study is that 47% of the students claim that they "understand the concepts but only sometimes manage to write correct code" and 37% of students feel that they "somewhat understand" these concepts "but do not know how to write correct code" to implement them. This is also supported by the literature (Sanders *et al.*, 2012) and is generally an accepted fact in computer science education. Students need first to understand the theory and then develop practical skills in order to become successful IT professionals. Unfortunately, novice programmers lack those problem-solving strategies that will enable them to design and code functional programs.

Interestingly enough, although students in the initial evaluation considered programming concepts to be rather easy, most of them were not very confident

in their ability to code correctly. To further explore the rationale behind why they thought they could not write correct code, students that selected choices 3 and 4 were asked to fill in an open text area describing the main reason why they could not produce correct code. Of the eighty-five (85) participating students, there were only twenty-five (25) responses to this open-ended question. Some characteristic responses included:

- I get frustrated with the error messages.
- I do not know how to fix compiler errors.
- I do not understand compiler syntax error messages.
- I get lost with the brackets.
- I think that my code is correct, I just do not know why it does not compile.
- I do not understand why my program is not doing what I think it is supposed to do.
- I know what I want to do, but I do not know how to put the commands together.
- Even when I have a correct pseudocode, I do not know where to start coding.

After performing coding on the given 25 responses, three general themes were identified: syntax errors; logic/semantic errors; and translation to code.

Effectively resolving syntax errors requires a very detailed knowledge of the programming language's syntax rules and experience in understanding the meaning or the implications of the error message(s) produced by the compiler. This task can sometimes be particularly challenging - for example, a single curly bracket (extra or missing) can cause a misleading compile-time error message which points to a different and completely unrelated line in the program and can also be affected by the programming environment itself. As Freund and Roberts claim in their research: *"student frustration is less a function of the language than of the programming environment"* (Freund & Roberts, 1996).

On the other hand, locating logical or semantic errors requires enhanced debugging skills, a detailed understanding of the effects of each command, and proper appreciation of the algorithm being employed.

Finally, the last theme of *"translation of pseudocode to code"* shows a student inability to construct an actual program despite having a clear idea of the requirements or steps involved. In order to put it in context in terms of the difficulties explored in the previous section of the questionnaire, this theme is similar to the rated difficulty *"transfer the algorithm to program code"*. A number of reasons are found in the literature that attribute to the difficulty of transferring an algorithm to a programming language and relate closely to the cognitive aspects of programming mentioned in the previous section:

- Lack of "one-by-one" translation rule from a pseudocode to code. This statement is also supported by Sanders *et al.* (2012).
- Inadequate/incomplete mental models of the process (Kessler & Anderson, 1986; Freund & Roberts, 1996; Winslow, 1996).
- Abstraction of the underlying notional machine that the students should learn to understand and manipulate (Xinogalos, 2014)

However, the introduction to coding using a visual programming environment could assist students to overcome at least some of the difficulties mentioned above, due to their inherent design and purpose: to prevent syntax errors and make the process of developing a program more intuitive and creative, without compromising the development of computational thinking skills.

Since 1986, professor have made attempts to overcome these difficulties with the integration of visual technologies into their teaching. Their main focus was (and still is) to motivate students by cultivating positive attitudes (less frustration) towards learning computer programming (Myers, 1986) .

In response to the last question in this study, to investigate professor perceptions as to whether students should be introduced to programming via the usage of visual programming environments, twenty-six professors (75%) answered yes, six professors (19%) answered that they were not sure and only two (6%) answered no (see Figure 4.2).

Educator perceptions: should students be introduced to programming using visual (block-based) programming environments prior to text-based programming?

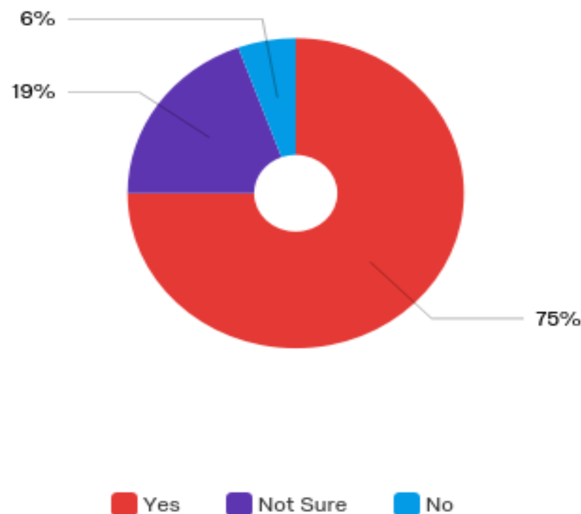


Figure 4.2: Educator perceptions about VPEs

A number of visual programming environments are available today, used by teachers with the intention to overcome at least some of the learning difficulties, discussed above.

To this end, a short review of the development and use of educational programming environments and software visualisation tools follows in the next section.

4.3 A Short History of Educational Programming Environments

As stated before, programming is a highly cognitive activity that requires acquiring new reasoning skills, understanding unfamiliar technical information and developing abstract representations of a process (Cañas, Bajo, & Gonzalvo, 1994; Ramalingam, LaBelle, & Wiedenbeck, 2004). An accurate framework or a so-called mental model of how the computer works is required to be formed in order to incorporate programming domain-specific knowledge. Norman (1987; 1988) defines mental models as the internal representations that people have about themselves, others, the environment and the things they interact with. He also argues that people use these mental representations to

reason about, explain, and predict the behaviour of external systems. The mental model of a system is formed through experience, training and instruction and by interpreting its perceived actions and its visible structure but *"People's mental models are apt to be deficient in a number of ways, perhaps including contradictory, erroneous, and unnecessary concepts"* (Norman, 1987). However, the internal components of the computer, where all data storage and processing take place do not have a visible structure. As a result, it is very important for novice programmers to develop an accurate mental model of how a program works.

Educators in the programming discipline have long faced the complexity of teaching programming, and as a result numerous educational programming languages and tools have been developed (from as early as the beginning of the discipline) that aim to enrich students' learning experience and reduce the obstacles imposed by the complex cognitive activities required by the process. Many educators believe that using a higher conceptual level of simplicity makes it easier for students to comprehend how a program works and thus learn programming more effectively (Du Boulay & O'Shea, 1981).

Initially, the purpose of educational or so-called pedagogical programming environments (PPEs) was to simplify the programming language - they later evolved to allow students to construct programs using graphical objects aimed at preventing syntax errors, to provide visualisations to assist in the formation of a solid model of the "notional machine", and to enhance the social learning dimension in order to motivate and engage students.

The first simplified programming language, "B.A.S.I.C." (Beginner's All-purpose Symbolic Instruction Code), was designed in 1964 by Kemeny, Kurtz and Keller aspired to provide an easier environment for non-science students to create computer programs (Figure 4.3).

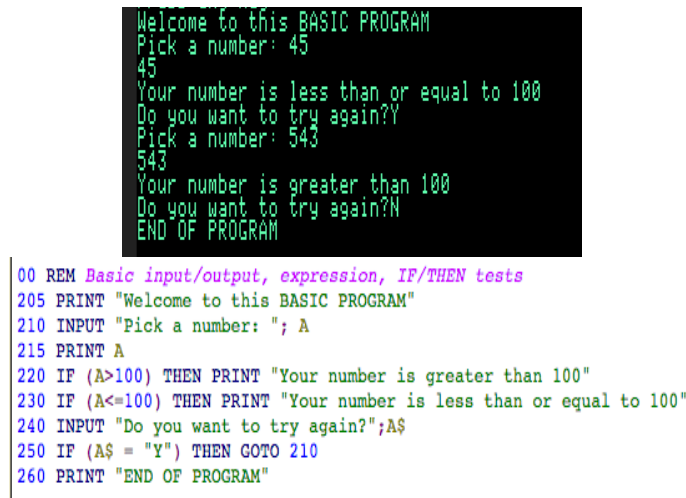


Figure 4.3: Example of a BASIC program source code and runtime.

Following BASIC, in 1967, Feurzeig, Papert and Solomon designed another educational programming language named Logo. Logo is widely known by a small robot called the *turtle*, which sat on the floor and which novice programmers learned to move around by typing English language commands on the computer, such as forward, left, etc. Soon the turtle was migrated to the computer screen using graphics. Flexibility, easy to remember commands, friendly error messages and immediate visual feedback were some of the main advantages of Logo (Figure 4.4).

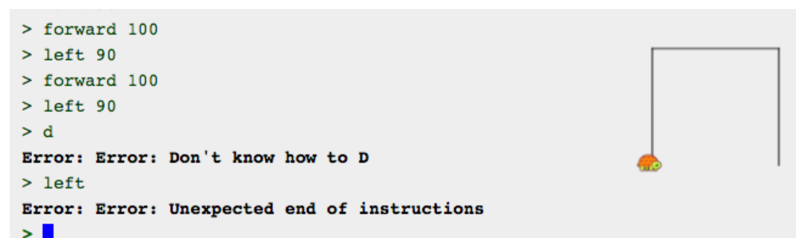


Figure 4.4: Logo programming environment with a virtual turtle

In 1980, Papert, based on the philosophy of *"constructionism"*, introduced the concept of Microworlds, a larger set of Logo-based implementations (yet having a limited scope) where children could actively experiment with "powerful" ideas by developing meaningful software projects (Papert, 1980; 1987). Microworlds allowed students to gain fundamental programming knowledge and experience without the barriers imposed by programming complexity (Figure 4.5).

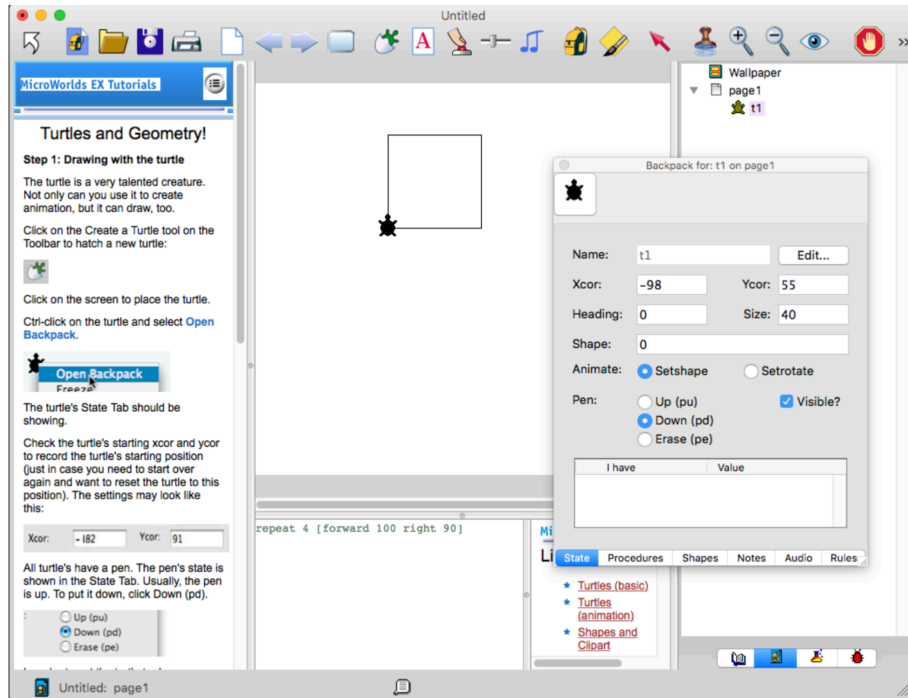


Figure 4.5: MicroworldsEX

In 1988, the Logo language was interfaced with traditional LEGO Bricks with the addition of newer components, such as motors and gears to create an "intelligent Brick". This new enhancement allowed children to construct and control their own mechanical toys. Resnick and Ocko mention that "*students rarely get the opportunity to design and invent things*" (1990, p.1) and describe how LEGO/Logo could provide them with this opportunity. LEGO/Logo has been evolved since then with the latest version LEGO Mindstorms EV3 released in 2013 (Figure 4.6).

Figure 4.6: A Lego car construction controlled by Logo programming language

The first visual programming language, Logo Blocks, was developed in 1996, at the MIT Media Lab, and served as the basis for all block-based visual programming environments, including Alice, Crickets, Scratch, Code.org, AppInventor and others. The purpose of the visual programming languages is to shift the focus of the novice programmer from syntax to problem solving research. Rigby and Thompson (2005) and Vogts, Calitz and Greyling (2008) have also shown that students face more difficulties when they try to learn programming using professional programming environments due to the complexity of the interface.

Block-based programming utilises ready-made blocks of commands, organised in palettes, that the user assembles to create a program. Since the programmer neither has to type nor memorise the instructions, there is no possibility of syntax-errors. The most widely known block-based programming environment is Scratch. The first version of Scratch was released in 2007. Scratch has gained great popularity in teaching programming over the past 5 years and has been used to introduce programming to students (from lower schools to universities) all over the world (Malan & Leitner, 2007; Resnick *et al.*, 2009). Currently, there are more than 27 million registered users in the Scratch website, with a continuing growing trend (see Figure 4.7).

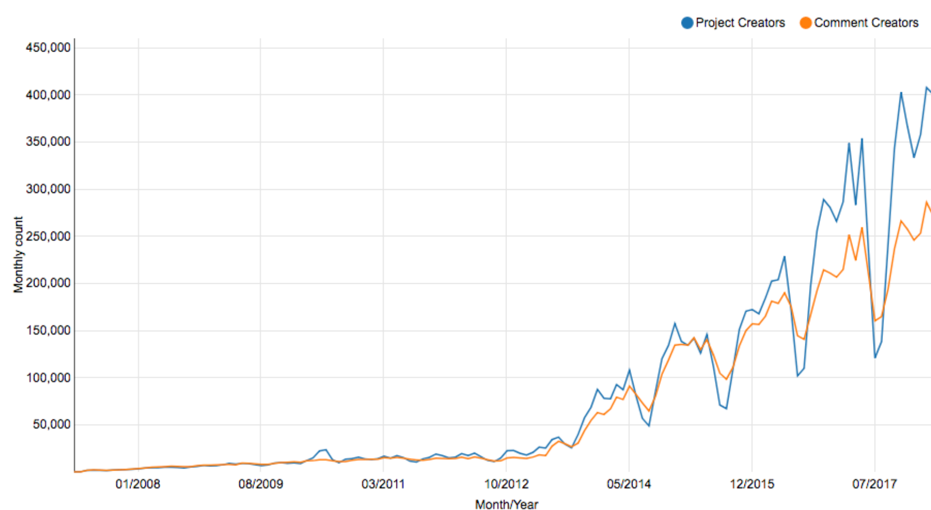


Figure 4.7: Scratch active users
(Retrieved from: https://scratch.mit.edu/statistics/ June 2018)

In the next section, I will discuss the various aspects of software visualisation tools and programming environments which are used in introductory

programming modules and attempt a categorisation based on their features and characteristics.

4.4 A Classification of Educational Programming Environments

Based on a literature review on programming environments classification (Myers, 1990; Burnett & Baker, 1993; Price *et al.*, 1993; Kelleher & Pausch, 2005; Sorva *et al.*, 2013; Xinogalos *et al.*, 2015) from a personal evaluation of the related characteristics/features, and the uses of the educational tools in the discipline, I have adapted and extended the existing classifications to include currently-used tools. The following classification in Figure 4.8 includes the main categories: type of editor; runtime environment (desktop/online); features; use; and type of visualisation each programming environment provides.

The primary distinction in this classification is between pedagogical/novice/educational programming environments, professional integrated development environments (IDEs) and command-line compilers (appearing in purple). At the second level of categorisation, in this study's pedagogical area of interest, most tools provide some kind of software visualisation (appearing in green): visual programming and algorithm/program visualisation. In a further breakdown of program visualisations as proposed by Price *et al.* (1993), visualisation of memory contents and program tracing are included as features in the proposed classification. The third level (appearing in yellow) includes the type of editor each tool provides to the user: block-based, icon-based, frame-based and text-based. The fourth level of classification relates to the type of execution environment: online or standalone. The last level (appearing in white) displays the name of the tool in the subcategory. Finally, features (appearing in purple), uses (appearing in light yellow) and types of visualisation provided (appearing in mauve) are linked with each tool using dotted lines.

The main purpose of this classification is to enable readers to understand where each of the numerous educational programming tools discussed in the study stands, as well as their similarities and differences. A limitation of this classification is that it is not fully comprehensive (due to the very large number

of educational programming tools in the market) and that it has been reviewed only by one other researcher in the area.

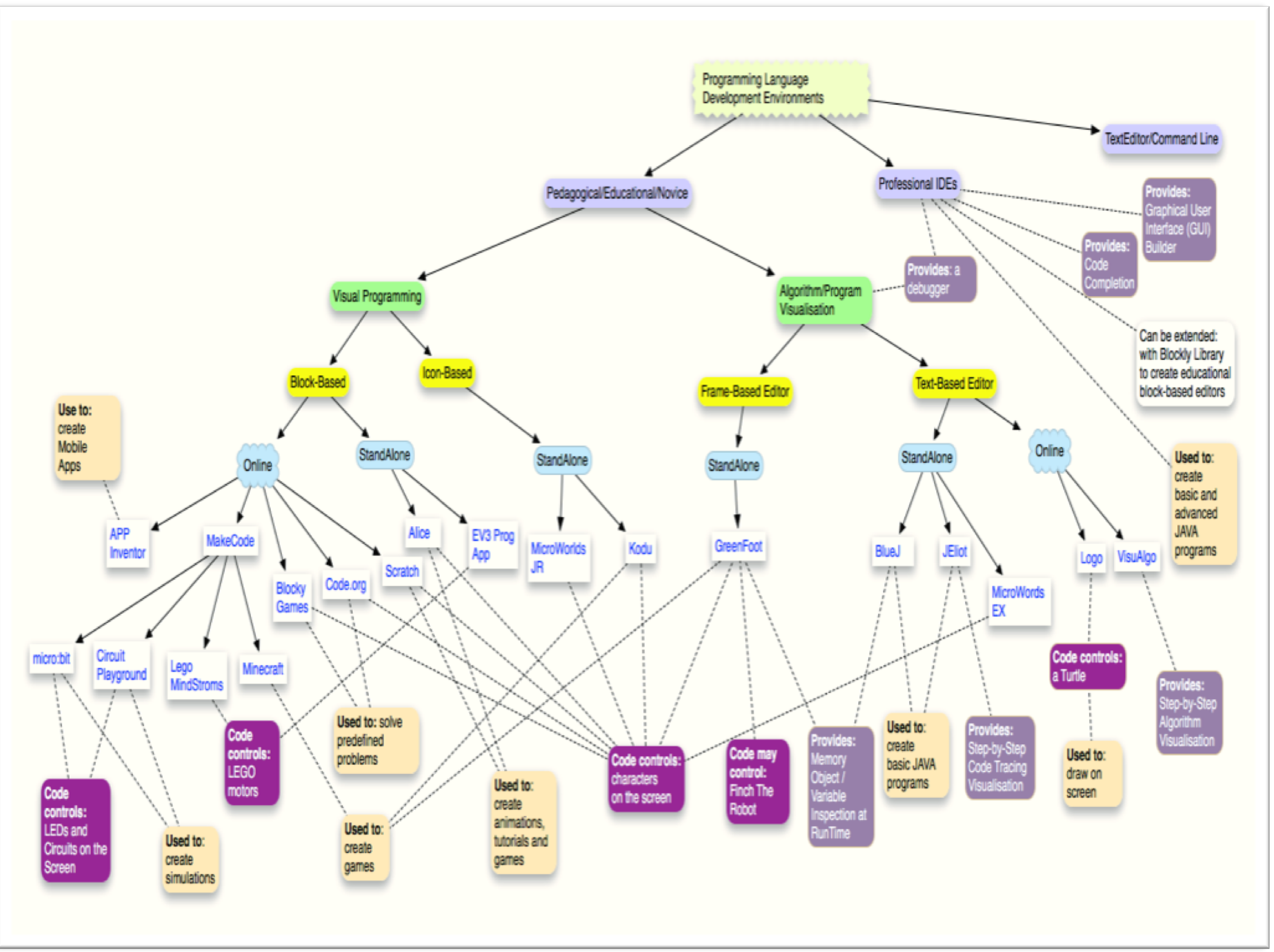

Programming Language Development Environments
TextEditor/Command Line
Professional IDEs
Pedagogical/Educational/Novice
Provides: Graphical User Interface (GUI) Builder
Provides: Code Completion
Provides: a debugger
Can be extended: with Blockly Library to create educational block-based editors
Used to: create basic and advanced JAVA programs
Visual Programming
Algorithm/Program Visualisation
Block-Based
Icon-Based
Frame-Based Editor
Text-Based Editor
Online
StandAlone
StandAlone
StandAlone
StandAlone
Online
Use to: create Mobile Apps
APP Inventor
MakeCode
Blocky Games
Code.org
Scratch
Alice
EV3 Prog App
MicroWorlds JR
Kodu
GreenFoot
BlueJ
JEliot
MicroWords EX
Logo
VisuAlgo
micro:bit
Circuit Playground
Lego MindStroms
Minecraft
Code controls: LEGO motors
Used to: solve predefined problems
Used to: create games
Used to: create animations, tutorials and games
Code controls: characters on the screen
Code may control: Finch The Robot
Provides: Memory Object / Variable Inspection at RunTime
Used to: create basic JAVA programs
Provides: Step-by-Step Code Tracing Visualisation
Code controls: a Turtle
Used to: draw on screen
Provides: Step-by-Step Algorithm Visualisation
Code controls: LEDs and Circuits on the Screen
Used to: create simulations

Figure 4.8: Programming development environment classification

4.5 Using Visualisations to Teach Programming

Many claims have been made that support the power of visualisations in learning facilitation (Shu, 1989; Pattis, 1993; Bergin *et al.*, 1996; Naps, 1997). Visualisations enable the learner to understand what happens inside the computer. Traditionally, teachers, including myself, have used graphical external representations to address the concept of visualisations (Gries *et al.*, 2005; Mselle, 2010; Hertz & Jump, 2013). One such technique is to draw boxes on the chalkboard/whiteboard (see Figure 4.9) to represent the contents of variables in computer memory handled by the program and attempt to perform a step-by-step program tracing.

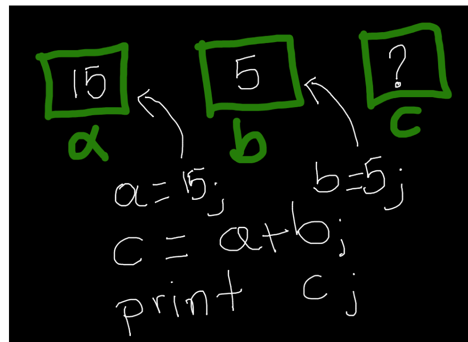


Figure 4.9: Traditional chalkboard visualisation

Another traditional visualisation technique involves using actual physical boxes like file cabinets. These boxes can be labelled with variable names, in which the learner can place a piece of paper with the written value to be assigned to the box, while hand-tracing the program code. I have found both of these techniques to be an excellent initial introduction of the variable concepts in my teaching. The main disadvantage, however, of such traditional approaches is that, as a program grows longer or more complex, teachers find it extremely time-consuming to draw and redraw the memory contents on the board or introduce more physical boxes to the class presentation. To assist in this process, researchers and educators have created software tools which provide computerised ways to create these visualisations.

Du Boulay and O'Shea (1981) used the following metaphor to describe software visualisation environments: *"A black-box inside the glass-box"*. Burnett (1999) defines visual programming as programming in which more than one

dimension is used to convey semantics. Dimensions include, but are not limited to, diagrams, relationships, time dependencies (before-after), sketches, icons, or even demonstrations of performed actions.

Software visualisation and visual programming environments were proven to be successful with students and to impart a positive impact on students' understanding, organisation of the concepts (Du Boulay & O'Shea, 1981; Eisenstadt,1992; Cañas *et al.*, 1994; Dann *et al.*, 2001; Boyle *et al.*, 2003; Čisar *et al.*, 2011) and student motivation. On the other hand, researchers in the area have found that experienced programmers consider novice programming languages as being overly simplified and even distasteful and in a sense, not telling the complete "truth" (Du Boulay & O'Shea, 1981).

Myers (1990) provided a classification of types of visualisations: program visualisation and visual programming. A more recent classification by Sorva *et al.* (2013) (Figure 4.10) provides a more detailed classification.

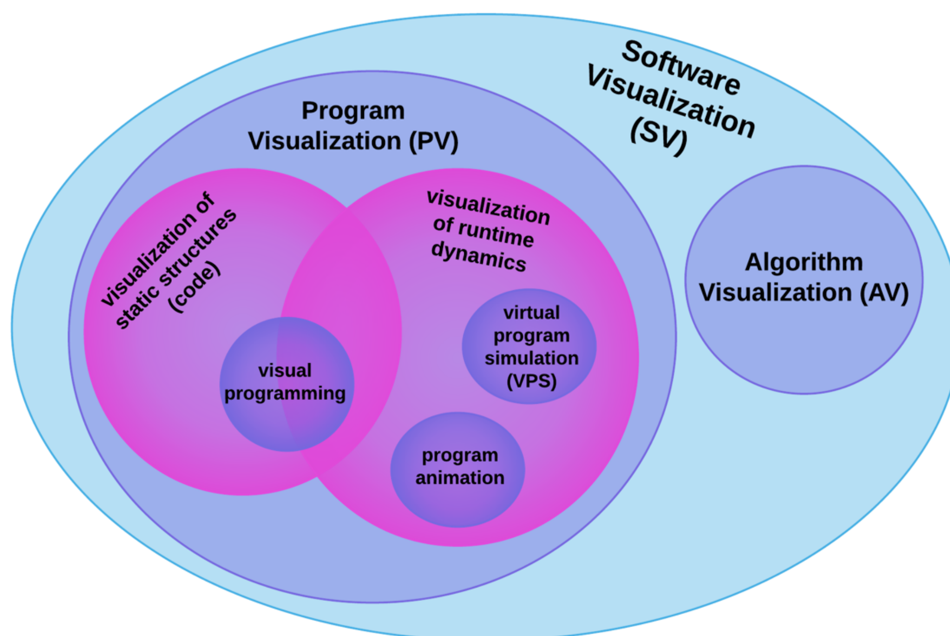


Figure 4.10: Forms of software visualisation (Sorva *et al.*, 2013)

The formal definition of **Software Visualisation** (SV) comes from Price (1993): *"Software visualisation is the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction*

to facilitate both the human understanding and effective use of computer software" (p.213).

The two main subcategories of software visualisation are program visualisation and algorithm visualisation.

Program Visualisation (PV) is the ability of an environment to provide and utilise graphics in order to illustrate some aspects of a program or its runtime execution, while the actual program code is written in text. Gershon *et al.* (1998) consider visualisation as the *"link between the two most powerful information processing systems: the human mind and the modern computer"* (p.29) and provides a definition outside the boundaries of computing, as the process of transforming data and information into a visual form enabling people to observe, explore and manipulate data more effectively. Cañas *et al.* (1994) tested program visualisation by utilising automatic code tracing, that demonstrated the status of all program variables during code execution. Their study showed that students in the tracing group developed semantically-oriented mental representations, as opposed to students in the non-tracing group, who developed syntactically-oriented mental representations, while their performance was not related to the way their mental representations were formed.

BlueJ is a representative example of a pedagogical development environment, classified in the subcategory of program visualisation, which specifically provides learners with visualisation of object instantiation, as well as direct observation and manipulation of memory contents (Figure 4.11). BlueJ enables students to view which values exist inside each variable at any given point in execution, thus supporting the notion of *"a glass box"*.

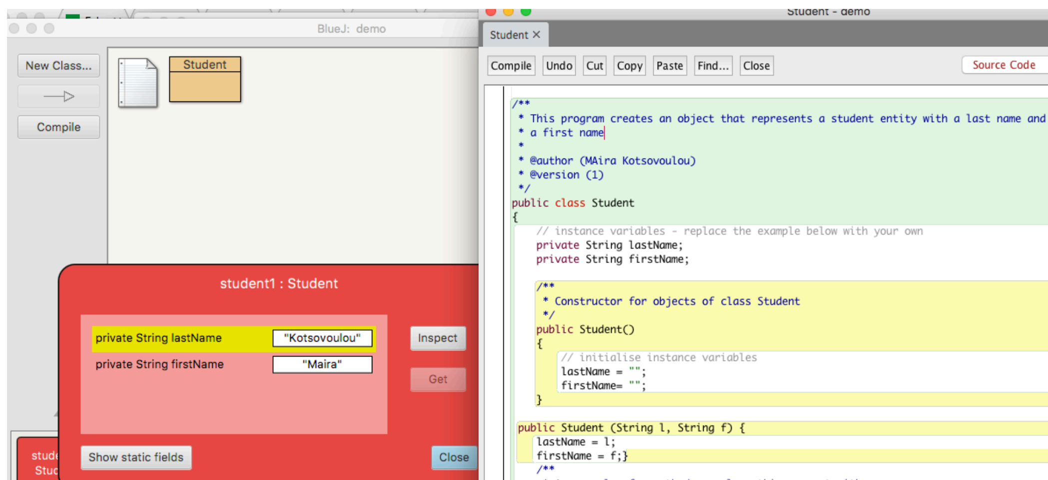


Figure 4.11: BlueJ variable inspection feature and the text-based code editor

Algorithm visualisation (AV) is the ability of an environment to use discrete images or animations to depict the execution of an algorithm and how it affects the data, while the user controls its execution (play/replay/pause/stop). Using algorithm visualisations, students can actively compare and contrast algorithms in terms of speed and efficiency. On the other hand, tools in this category operate at a high level of abstraction and their purpose is not to demonstrate the fundamentals of the program execution, but to provide concrete representations of the abstract notions of algorithm methodologies (Kehoe, Stasko, & Taylor, 2001). Grissom, McNally and Naps (2003), in a study measuring the effects of algorithm visualisation, found that learning increases with a rise in the level of student engagement. Simply viewing an animated algorithm does not necessarily demonstrate a noticeable gain in learning, while responding to questions during algorithm execution and provoking and engaging in additional exploration activities does improves learning. Hundhausen *et al.* (2002) performed a meta-study based on 24 research projects about the effectiveness of algorithm visualisation, of which 11 showed a positive impact, 10 showed no significant difference, 1 showed a negative effect and 1 showed a positive effect not directly related to algorithmic visualisation. VisuAlgo (Figure 4.12) is such an online tool which enables students to observe the step-by-step animated execution of an algorithm.

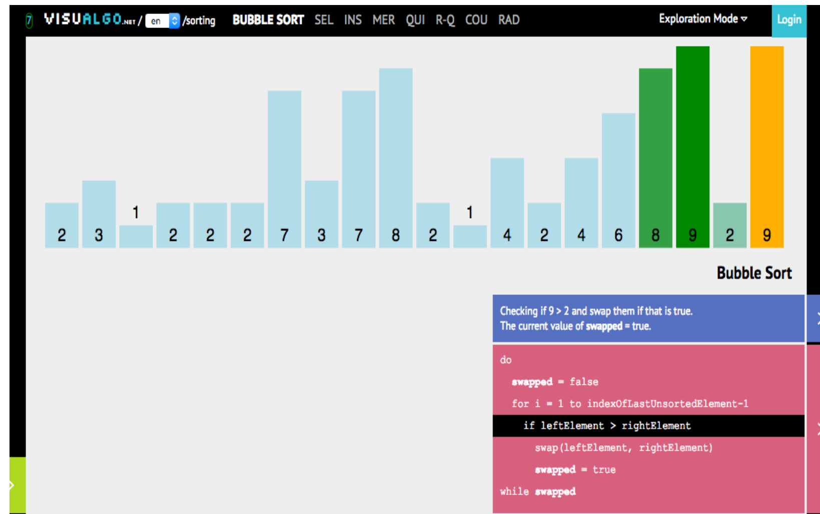


Figure 4.12: Sample algorithm execution in VisuAlgo

Visual programming, a subcategory of program visualisation, is the ability of the programming environment to specify the code by letting the user spatially arrange ready-make blocks of code, such as Scratch, Alice, App Inventor, Code.org, etc. (see Figure 4.13). Visual programming can assist users to reduce or even completely eliminate the potential of making syntax errors.

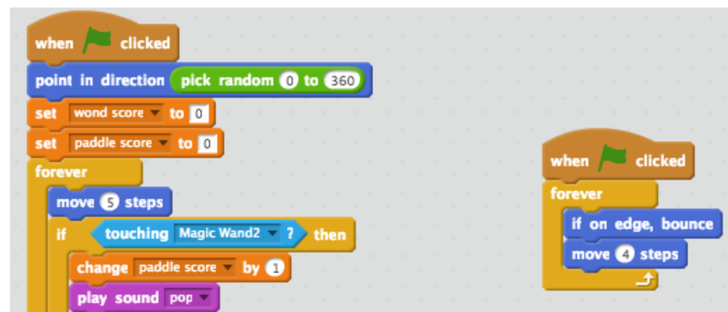


Figure 4.13: Scratch visual programming code editor

Furthermore, creating code with visual programming environments enables students to build multimodal artefacts (incorporating text, sound, graphics, animation and user interactions) while interacting with multimodal interfaces. The multimodalities imposed by the use of visual programming environments especially within the context of game creation have been shown to be a great

motivation tool (Gee, 2003; Jewitt, 2005) and found to enhance through programming a sense of accomplishment and self-esteem (Muraina *et al.*, 2019).

Visual programming environments could possibly provide a framework for novices to learn programming, by targeting all three conditions of meaningful learning: reception, availability and activation, without the frustration caused by syntax errors. All commands (depicted in the form of blocks) are illustrated by the professor during the reception stage. This can be performed using examples and live-coding. During the availability stage, the learner - when faced with a programming assignment - must process the requirements and identify which are the most appropriate commands to use to solve the problem. Availability could be enhanced with VPEs because they provide the learner with a full list of all available commands, categorised according to their functionality (motion, sound, control, events, variables, etc.). Finally, the learner must connect new commands to the ones previously learned to create a project. Using problem-decomposition skills and a step-by-step development approach, activation could be enhanced.

As stated in the previous chapter, students use motivational strategies to drive and inspire them to accomplish academic tasks (Pintrich & de Groot, 1990; Wolters, 1999; Pintrich, 2004; Code *et al.*, 2006). Understanding how students are motivated to explore, discover, learn and set their personal achievement goals could have a significant impact on choosing the most appropriate learning environment and teaching pedagogy.

In the next section, I intend to justify the selection of Greenfoot, Alice, AppInventor and Scratch as the pedagogical programming environments used for the preliminary investigation of this research study.

4.6 Related Research on Greenfoot, Alice, AppInventor and Scratch

As mentioned previously in Chapter 2 , there are numerous programming languages and many different programming environments including the ones which aim to teach object oriented programming by creating computer games like Greenfoot, the ones which employ visual elements (blocks) to replace the typing of programming instructions like Scratch and AppInventor, and a later

addition to this large list of tools, the ones that “restrict” the typing of code in pre-determined frames, making coding less error prone.

The third research question of this study considers primarily the exploration of students’ motivations for learning programming and how these relate to their perceptions about programming and more specifically to their perceptions about visual programming environments. To support this aim, I seek to discover whether or not visual programming environments affect student motivation to learn programming.

The need to assess the assumption that a visual programming environment might affect student motivation imposed the selection of an appropriate visual programming tool but also created another challenge for this project. Most visual programming environments mentioned previously satisfy the requirements that form the basis of this research, which is to promote ‘fun’ and engaging learning experiences. First of all, the selected tool should be able to fulfil the educational goals and learning outcomes of the module (refer to Chapter 1, contextual information). Then the selected tool should conform with the underlying constructivist philosophical view and the constructionist instructional method of problem-based learning by engaging learners in the learning process and setting a game-like context for the programming assignments. Finally, the main focus of the tool should be on promoting the understanding of programming logic by eliminating the burden of syntax errors.

Greenfoot with the frame-based editor, Alice, AppInventor and Scratch with their block-based code building blocks, all satisfied the above requirements. They are very popular and widely used for the introduction of programming concepts around the world and are also extensively used in related research because they all address the need to reduce complexity and at the same time enhance students’ motivation to learn how to program (Malan & Leitner, 2007; Leitner *et al.*, 2009; Maloney, *et al.*, 2010; Nikou & Economides, 2014).

Greenfoot has been used in studies and workshops aiming at teaching computational thinking by creating two-dimensional board games and simulations using the Object-Oriented Programming approach (Henriksen & Kölling, 2004; Gallant & Mahmoud, 2008; Hijon-Neira *et al.* 2013; University

of Kent, 2014). Findings from these studies have shown increased student engagement and motivation. Furthermore, the students' subjective opinion about Greenfoot from a key study was overall positive and they enjoyed all activities they successfully completed (Gallant & Mahmoud, 2008). Greenfoot's characteristics such as interactivity and visualisation supported active experimentation and exploration while providing immediate feedback, leading to more than 60% success in learning the concepts taught (Begosso & Begosso, 2012). One main difference of Greenfoot in relation to the other three pedagogical programming environments, discussed in this section, is that coding tasks are completed either by typing the commands in the Java programming language or using frames.

Alice's main similarity with Greenfoot is that it is based on the Object-Oriented Programming (OOP) approach but its main difference from the rest of the visual programming environments is that it uses three-dimensional animated actors and scenes for the construction of virtual worlds. Coding tasks are completed by using blocks. Research studies have reported improvement in student performance, enjoyment and confidence in understanding programming concepts when using it (Cooper *et al.*, 2004; Moskal *et al.*, 2004; Bishop-Clark *et al.*, 2007; Sykes, 2007). More specifically, Bishop-Clark *et al.* (2007) reported a significant decrease in creativity and overall attitude towards programming for students that did not use Alice. On the other hand, Cliburn (2008), in his study about student opinions of Alice, reported that 40.5% of students were not convinced that Alice contributed to their learning of Java.

Choice of program can also be considered in terms of the adoption rate of mobile devices among students, which is exponentially growing. APPInventor's driving force is on "what is being built" (Wolber, 2011; Wolber *et al.* 2015) with emphasis on exploring how to solve real-world problems, using applications for mobile devices; again by "hiding" code complexity, these are reducing syntax errors with the use of blocks. Taking advantage of mobile devices to motivate and expose students to problem-solving and computational thinking is the main target for AppInventor. Research studies again report increase in engagement, intrinsic goal orientation, self-efficacy and task value (Wagner *et al.*, 2013; Nikou & Economides, 2014) for students who are exposed to programming using AppInventor.

Scratch, with the media-rich block-based programming environment, has also been extensively used in research relative to teaching introductory programming modules for lower school (Calder, 2010; Wilson B., 2010; Tsai & Chen, 2011; López *et al.*, 2016; Chiang & Qin, 2018), middle school (Meerbaum-Salant *et al.*, 2011; Fields *et al.*, 2013; Nikou & Economides, 2014), upper school (Moreno-León, Robles 2015; Weintrop, 2015; Pellas & Peroutseas, 2016) and universities (Malan & Leitner, 2007; Malan, 2010; Ozoran *et al.*, 2012; Saltan & Kara, 2016; Yukselturk & Altiok, 2016; Erol & Kurt, 2017) aiming to examine its effects on the students' motivation, achievement, self-efficacy and overall attitude towards programming. In a recent study by Erol and Kurt (2017), it was revealed that their participants' programming achievements increased for the Scratch group, but also demonstrated skill transferability to C# which was the programming language used after Scratch.

4.7 Conclusion

It seems apparent that all the VPEs discussed above have active experimentation and exploration as a common underpinning pedagogy. They all conform with an underlying constructivist philosophical view and a constructionist instructional method of problem-based learning by engaging learners in learning processes and setting a game-like context for the programming assignments. They have all been used in a range of studies to examine their effects in student motivation to learn programming, with positive results.

The question that then arises is which VPE is the most appropriate to be used in this study; the answer to this question is sought in the next chapter, which presents the detailed methodology leading to the selection process.

Chapter 5 The Pilot Study

5.1 Purpose

The purpose of this chapter is to present and outline the research design and present the rationale as to why participatory action research has been identified as the most appropriate methodological practice for the preliminary investigation into selection of the visual programming environment to be used for the main study.

5.2 Participatory Action Research

Action research is an iterative process and is sometimes referred to as an "iterative case study". It involves researchers and practitioners acting together on a cycle of tasks, including problem diagnosis, action intervention and reflective learning (Avison *et al.*, 1999). Action research focuses on a change process (Runeson, 2012) and on the outcomes of interventions and aims, and improvement, reflection, monitoring and evaluation of the outcomes (Cohen *et al.*, 2013).

The aim of the action research cycles is to investigate the possible effects of a particular change before considering it for my main research. As a researcher, I attempt to solve a real-world problem (how to motivate students to learn programming) while simultaneously studying the experience of solving that problem (Davison *et al.*, 2004).

According to Stringer *et al.* (2010), action research works through three basic phases:

- Look: build a picture and gather information, define and describe the problem to be investigated and the context in which it is set.
- Think: interpret, analyse and explain the situation.
- Act: judge the worth, effectiveness, appropriateness, and outcomes of the activities.

Action research can be incorporated into all phases of instruction and works in cycles, where each cycle is informed by the previous one:

- phase 1 during lesson planning and preparation;
- phase 2 during instruction; and
- phase 3 during assessment and evaluation.

Action research is neither quantitative nor qualitative in nature, but it may use data collection techniques that involve either one or both of these approaches, such as collection of quantitative data (student performance examination results) and qualitative data (student opinions), by conducting experimental case studies. Action research focuses on the outcomes of interventions and aims and improvement, reflection, monitoring and evaluation of the outcomes (Cohen *et al.*, 2013).

Action research, and participatory action research in particular, has been considered a desirable tool for educators, in that it helps them to search for better ways to meet their students' needs, monitor and evaluate the impact of changes, reflect on the process, and thus promote positive change in educational settings. Critical participatory action research brings together the *"self-reflective collective self-study of practice, and transformational action to improve things"* (Kemmis *et al.*, 2013). Carr and Kemmis (1986) criticised the idea that the researcher should remain an "objective" and "disinterested" observer, but rather should engage in active self-reflection of the conduct and the consequences of his/her practices.

Based on the definition of the aims of action research and the previously mentioned criticism of detached researchers, I found participatory action research to be a suitable research methodology for the preliminary investigation regarding selection of the visual programming environment, because my aim is to find more desirable and interesting ways to introduce students to the art of programming.

Action research, in this context, aligns with the postulation of Stenhouse (1985), referred to in Bassey (1999), that it "is concerned with contributing to

the development of the case under investigation by feedback of information which can guide revision and refinement of the action" (p.28).

Having identified the problem area of the study, which is the difficulty novice programmers face when they learn how to program, and having conducted a literature review on relevant educational theories, learning approaches and motivation, I selected Greenfoot, Alice, AppInventor and Scratch as the visual programming tools to be used for this pilot research for the reasons mentioned at the end of Section 2.3.4.

I planned to go through 4 action research cycles (one for each of the tools mentioned). Each action research cycle aims to investigate student perceptions about the tools' enjoyment, usability and suitability towards the achievement of the specific module's learning objectives and, secondly, to observe how each of these tools affected students' motivation to learn programming (see Figure 5.1).

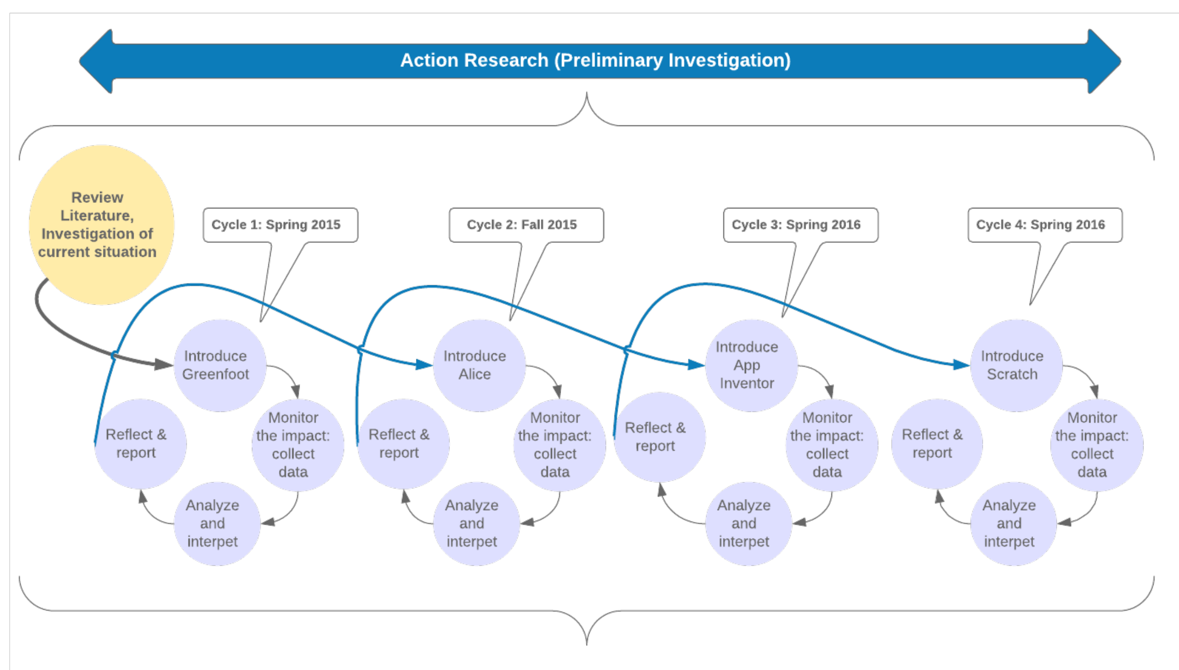


Figure 5.1: Action research cycles

To this end, I prepared lesson materials, trained fellow professors and jointly introduced each tool to our students. Upon completion of each cycle, data were collected from student assessment scores (from a homework exercise), a short survey and via in-class discussions.

All participants, in all cycles, were given the same exercise to complete at the end of the instruction.

The homework exercise, which required students to create a hangman game, was designed in such a way that it could be completed using any of the four participating Visual Programming Environments. Grading was performed using the same rubric (see Table 5.1) based on the following criteria: functionality, complexity, use of graphics and animation, use of sound, scoring, levels, player mode and use of word dictionaries).

Criterion	**Evaluation and Points**			
Is the game functional?	No	Small Bugs	Yes	
	0	5	10	
Code complexity	No code delivered	Easy (Only Basic Structures)	Most programming constructs are correctly used	Complex - Advanced & modular code
	0	5	10	15
Use of graphics and animation	No graphics	Simple (from the existing Library)	Simple graphics (from the existing Library)	Advanced (use of graphics and animation)
	0	5	10	15
Use of sound	No Sound	Simple sounds (from existing library)	Advanced (custom recorded sounds or many sounds for different events)	
	0	5	10	
Code to keep scores	No Scores	Simple Scores (just display Score)	Keep Ranks and Store Past Scores	
	0	5	10	
Single player or two player mode	Single Player (one player inputs the word)	Two Players (alternating turns)	One Player against the computer	Two Players (alternating turns) against the computer
	0	5	10	15
Increasing level of difficulty	There is not increasing level of difficulty	2 levels	more than two levels	
	0	5	10	
Words Dictionary	no dictionary used	static dictionary (seeded into the program)	user can upload a dictionary	program downloads the dictionary from the web
	0	5	10	15

Table 5.1: Grading rubric for the formative assessment used in all action research cycles

5.2.1 **The Survey Tool**

The survey questionnaire contained 2 demographic questions (regarding gender and age), 5 general questions concerning their major course of study, pathway, year of study, level of programming experience, programming languages they knew, and previous experience with the tool.

The next section of the questionnaire contained 5 questions adapted from Pintrich *et al.*'s MSLQ (1991) concerning students' intrinsic motivation (Q9 - Q12), extrinsic motivation (Q13) to learn programming and 1 question concerning self-efficacy (Q14). The keyword "this tool" was replaced in each action research cycle with the name of the visual programming environment which was introduced as part of the intervention. Following Pintrich *et al.*'s (1991) recommendation, students were asked to read each question and rate how much they agreed or disagreed with the statement using a seven-point Likert Scale (1 = strongly disagree, 2 = moderately disagree, 3 = somewhat disagree, 4 = neutral (neither disagree nor agree), 5 = somewhat agree, 6 = moderately agree, 7 = strongly agree). For the evaluation of the results, scales were constructed by taking the mean of the items that made up each scale.

The last section contained 8 questions. Specifically, questions 15, 17, 19 informed the enjoyment factor, questions 16, 18 and 21 informed usefulness factor, question 20 provided an idea about the intention to use the tool outside the classroom environment and finally questions 22 and 23 acted as the final "vote" of the participants so that the "tool" could be adopted for the Introduction to Programming module.

Enjoyment factor adjective pairs were: boring/fun, unenjoyable/enjoyable and unpleasant/pleasant, while perceived usefulness adjective pairs were: ineffective/effective, useless/useful (adapted from Davis *et al.*'s questionnaire (1992)). The not beneficial/beneficial pair was not included in Davis's questionnaire, but was proposed by a focus group of 4 IT professors who teach programming in XYZ college, who studied the Davis *et al.* questionnaire and found that *"improve job performance"* and *"increase productivity"* questions did not fit in the case under investigation.

Students were asked to rate their perceptions utilising semantic differential (bipolar) rating scales from 1 (strongly disagree) to 7(strongly agree), based on Martin Fishbein and Icek Ajzen's theory of reasoned action (Fishbein & Ajzen, 1975).

Figure 5.2 shows the adaption of the technology acceptance model used for the evaluation of the perceived acceptance of each of the visual programming environments tested in this pilot study and the justification of the selection of the tool to be used for the main study.

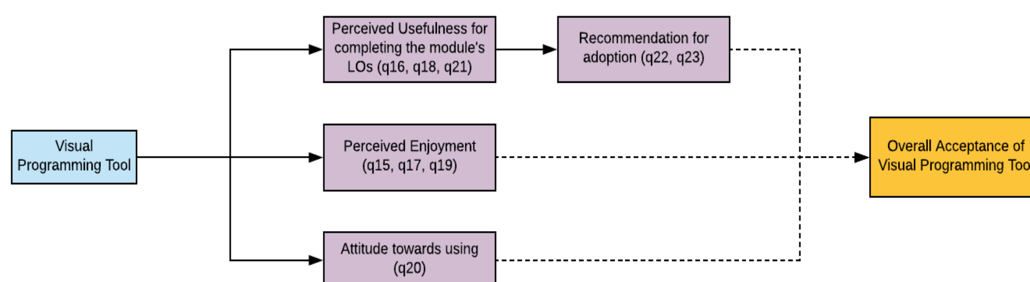


Figure 5.2: Adaption of the Technology Acceptance Model

5.2.2 Validity and Reliability for the Pilot Study Survey Tool

The final version the tool was tested with a sample of 127 student responses for construct, convergent and discriminant validity.

5.2.2.1 Construct Validity

Confirmatory Factor Analysis (CFA) was used to test whether or not the data collected from the questionnaire fit the hypothesized measurement model and as such to evaluate construct validity. Table 5.2 presents the factor loadings of the questionnaire and the three components extracted: Usefulness, Enjoyment and Intrinsic Motivation.

Rotated Component Matrix			
	1 Usefulness	**2 Enjoyment**	**3 Intrinsic Motivation**
VPE is Fun (q15)		0.947	
VPE is Enjoyable (q17)		0.930	
VPE is Pleasant (q19)		1.002	
VPE is Effective (q16)	0.906		
VPE is Beneficial (q18)	0.935		
Create Useful Programs (q21)	1.171		
Intent to Use (q20)	1.052		
Preferable over Java (q22)	1.101		
Interest in Programming (q9)			1.104
Prefer Challenging work (q10)			0.848
Enjoy module subject (q11)			1.065
Useful module subject (q12)			1.235
Extraction Method: Principal Component Analysis. Rotation Method: Varimax with Kaiser Normalization. a Rotation converged in 5 iterations.			

Table 5.2: CFA - Action Research Survey

Reliability analysis using SPSS was performed to analyse the internal consistency of the scales. The resulting Cronbach's alpha values were all above the recommended level of .70, thus indicating adequate internal consistency (Cronbach, 1951; Peterson, 1994; Tavakol & Dennick, 2011; Vogt, 2007) (see Table 5.3).

Intrinsic Motivation Scale (4 Items, Cronbach's alpha = 0.844)				
	Scale Mean if Item Deleted	Scale Variance if Item Deleted	Corrected Item-Total Correlation	Cronbach's Alpha if Item Deleted
Interest in Programming (q9)	15.9213	10.295	0.785	0.758
Prefer Challenging work (q10)	16.0236	12.055	0.633	0.824
Enjoy module subject (q11)	15.8504	10.557	0.744	0.776
Useful module subject (q12)	16.3858	10.112	0.599	0.852
Usefulness Scale (5 Items, Cronbach's alpha = 0.844)				
VPE is Effective (q16)	12.1654	19.393	0.648	0.821
VPE is Beneficial (q18)	12.4803	19.077	0.719	0.805
Create Useful Programs (q21)	12.4803	17.680	0.692	0.809
Intent to Use (q20)	12.7165	18.236	0.678	0.812
Preferable over Java (q22)	12.4567	18.520	0.578	0.842
Enjoyment Scale (3 Items, Cronbach's alpha = 0.838)				
VPE is Fun (q15)	7.4488	4.329	0.733	0.744
VPE is Enjoyable (q17)	7.0866	4.588	0.618	0.855
VPE is Pleasant (q19)	7.4016	4.099	0.757	0.719

Table 5.3: Cronbach's alpha – Action Research Survey

5.2.2.2 Convergent Validity

Convergent validity is the assessment to measure the level of correlation of multiple indicators of the same construct that are in agreement. According to Hair *et al.* (2016) to establish convergent validity, the factor loading of the indicator, composite reliability (CR) and the average variance extracted (AVE) should be considered. Table 5.4 presents the calculations for AVE and CR for the intrinsic motivation scale.

N=4	**λ Factor Loadings**	**λ^2**	**E Error Variance**
	0.881	0.77616	0.22384
	0.763	0.58217	0.41783
	0.850	0.72250	0.27750
	0.816	0.66586	0.33414
SUM	**3.310**	**2.74669**	**1.25331**
AVE	0.687	SQRT of AVE	0.82866
CR	0.897		

Table 5.4: AVE and CR calculations for Motivation Scale

The same calculations were performed for the usefulness and enjoyment scales and the results are presented in Table 5.5.

Scale	AVE	CR	SQRT(AVE)
Motivation	0.687000	0.897349	0.828660
Enjoyment	0.582410	0.874481	0.763158
Usefulness	0.714723	0.882392	0.845413

Table 5.5: AVE and CR values for all scales
AVE > 0.50 (Acceptable), AVE > 0.70 Very Good, CR > 0.70 Acceptable

The calculated AVE values exceed the recommended value of 0.50 and CR values exceed 0.70, so the questionnaire scales can be considered as adequate for convergent validity (Fornell & Larcker, 1981; Hair *et al.*, 2016).

5.2.2.3 Discriminant Validity

Discriminant validity is the extent to which a construct is truly distinct from other constructs by empirical standards. Thus, establishing discriminant validity implies that a construct is unique and captures phenomena not represented by other constructs in the model. According to the Fornell-Larcker testing system, discriminant validity can be assessed by comparing the square root of each AVE in the diagonal with the correlation coefficients (off-diagonal) for each construct in the relevant rows and columns (Fornell & Larcker, 1981).

For intrinsic motivation, the value obtained for the square root of AVE (0.828660) is greater than the correlation coefficients (see Table 5.6) which leads us to accept the discriminant validity of the scale.

Intrinsic Motivation Scale				
	q9	q10	q11	q12
Interest in Programming (q9)	1			
Prefer Challenging work (q10)	.598	1		
Enjoy module subject (q11)	.766	.602	1	
Useful module subject (q12)	.590	.460	.521	1

Table 5.6: Correlation matrix for motivation scale components

For enjoyment and usefulness scales, the value obtained for the square root of AVE (0.763158, 0.845413 respectively) is greater than the correlation

coefficients (see Table 5.7) which leads us to accept the discriminant validity of the scales.

Enjoyment Scale								
	q15	q17	q19	q16	q18	q21	q20	q22
q15	1							
q17	0.561	1						
q19	0.747	0.593	1					
Usefulness Scale								
q16	0.330	0.271	0.267	1				
q18	0.289	0.322	0.270	0.668	1			
q21	0.306	0.232	0.355	0.555	0.519	1		
q20	0.338	0.379	0.391	0.492	0.620	0.606	1	
q22	0.289	0.250	0.332	0.406	0.498	0.531	0.469	1

Table 5.7: Correlation coefficients for enjoyment and usefulness scales

5.2.3 Action Research Cycle 1 (Greenfoot)

During Spring Semester 2015, I performed the first action research cycle with the introduction and evaluation of Greenfoot.

To begin with, the literature review on Greenfoot as a visual programming environment reported intriguing research results (Gallant & Mahmoud, 2008; Decker & Trees, 2010; Begosso & Begosso, 2012). More specifically, as mentioned in an impact case study submitted by the University of Kent, students benefit from the use of Greenfoot *by "being able to achieve more tangible results more quickly, leading to increased motivation and satisfaction, as well as better understanding of programming concepts"* (University of Kent, 2014)

The test this theory in the context of the 'Introduction to Programming' module, Greenfoot was incorporated in the module's material. Students were first introduced to the Java programming language and, during the last 2 weeks (12-hours of instruction) of the module, students learned how to create games using Greenfoot. The material used for teaching Greenfoot was taken from Oracle Academy's Java Fundamentals course ("Java Fundamentals – Course Description"). A sample in-class activity is included in Appendix Five.

Thirty-five (35) students, consisting of two females and thirty-three males, participated in the study and were registered in 2 separate classes. One class was taught by me and the other by a fellow professor. Both of us had extensive teaching experience in teaching the 'Introduction to Programming' module (15-18 years).

At first glance, project grade statistics showed an improvement in the pass/fail rate. More specifically, 25 students passed the course and only 9 failed. This translates to a 74% pass rate, while the pass rate was as low as 51% during the previous two years (2013-2014). Although the increase in the pass/fail rate from previous semesters is obvious, feedback obtained from the post-instruction survey (see Appendix Two - Action Research Survey) and 5 semi-structured interviews showed that students not only deemed Greenfoot inappropriate for the course, but also found it confusing and difficult to program. One of the main disadvantages reported by students was that they were obligated to learn - along with the programming language - the use of Greenfoot's specific libraries of commands. Some negative comments of students in the last open-ended question of the survey included: *"Greenfoot did not help me in any way to finish my project"* and *"The reason I did not like Greenfoot was mainly because I would like to know what is the original code of that game we created for example, "main" code was locked by the creator"*. Another reported difficulty was that Greenfoot is heavily based on object-oriented programming concepts, whereas the module serves as an introduction to programming concepts. On the other hand, only one student reported that "*Greenfoot was a fun and creative way to learn programming*".

Table 5.8 shows the mean, median, standard deviation, minimum and maximum scores for intention to use, preferability over Java, overall enjoyment and tool usefulness as perceived by students who participated in the survey, as well as their recommendation for future adoption of the tool for the module.

VPE Visual Programming Environment - 1 Greenfoot						
	Median	Mean	Standard Deviation	Minimum	Maximum	Count
q20 Intent to Use	3.00	2.37	1.09	1.00	5.00	
q22 Preferable over Java	3.00	2.63	1.40	1.00	7.00	
Overall Enjoyable	3.00	3.22	0.88	1.00	5.00	
Overall Useful	3.00	3.00	0.61	1.00	4.00	
Assessment Grade	40.00	32.86	32.70	0.00	95.00	
Recommendation for use						
No						28
Maybe						3
Yes						4

Table 5.8: Student evaluation of Greenfoot programming environment
(Scale 1=Negative Opinion, 4=Neutral, 7=Positive Opinion)

Data obtained from the survey and student feedback from the interviews did not demonstrate a high student preference to use Greenfoot. Twenty-eight out of thirty-five students did not recommend the use of Greenfoot for the module. The average assessment grade of 40 also did not show optimal outcomes. These results from the first cycle initiated the second cycle, which took place during Fall Semester 2015. This cycle involved the evaluation of Alice as an instructional tool for the introduction to programming.

5.2.4 Action Research Cycle 2 (Alice)

A literature review on Alice as a visual programming environment also reported intriguing research results (Cooper *et al.*, 2000; Moskal *et al.*, 2004; Powers *et al.*, 2007; Al-Linjawi *et al.*, 2010; Dann *et al.*, 2012). For example, Moskal *et al.* (2004) in a two-year study which took place in two universities in order to examine the effectiveness of Alice for improving performance and retention, reported improved student performance, highly positive student experiences, as well as a stimulated interest for computer science in general. Dann *et al.* (2012) also reported that using Alice to introduce programming concepts before Java in a college first-year programming course (for two semesters) showed a significant positive impact on students' learning.

Inspired by these findings, Alice was incorporated in the material of the 'Introduction to Programming' module and its instruction cycle lasted for 2 weeks (12-hours of instruction).

The material used for teaching Alice was taken from Oracle Academy's Java Fundamentals course ("Java Fundamentals – Course Description") and a number of activities from the Alice.org website. A sample car race activity is included in Appendix Five.

Thirty-five (35) students participated in this study (registered in 2 classes). The first class was taught by me and the second by another professor. The results from the second preliminary investigation were not very promising either. Formative assessment scores and final course grades did not demonstrate an increase from past semesters, and feedback obtained from a survey and 4 semi-structured interviews showed that students overall found Alice very childish and not useful for the module.

Table 5.9 shows the mean, median, standard deviation, minimum and maximum scores for intention to use, preferability over Java, overall enjoyment, tool usefulness as perceived by students who participated in the survey, as well as their score in the homework exercise and their recommendation for a future adoption of the tool for the module.

VPE Visual Programming Environment – 2 Alice						
	Median	Mean	Standard Deviation	Minimum	Maximum	Count
q20 Intent to Use	3.00	2.26	1.01	1.00	4.00	
q22 Preferable over Java	3.00	2.74	1.20	1.00	5.00	
Overall Enjoyable	3.00	3.28	0.97	1.00	5.00	
Overall Useful	3.00	2.83	0.79	2.00	5.00	
Assessment Grade	40.00	33.71	28.24	0.00	90.00	
Recommendation for use						
No						25
Maybe						7
Yes						3

Table 5.9: Student evaluation of Alice programming environment
(Scale 1=Negative Opinion, 4=Neutral, 7=Positive Opinion)

Based on the above findings, which were not encouraging and with the intent on finding a Visual Programming Environment that could potentially increase student motivation to program, I found relative research on Scratch and AppInventor block-based educational programming environments that showed

positive results when used by students with no prior programming experience (Liu *et al.*, 2012; Nikou & Economides, 2014; Papadakis *et al.*, 2014).

5.2.5 Action Research Cycles 3 and 4 (Workshops on AppInventor and Scratch)

The third and fourth cycles were shortened in duration and did not involve changes in the content of the module or in the teaching methodology, but an introduction of two new programming environments in the form of workshops. Participation was voluntary and not formally assessed.

These two short cycles were designed in the form of workshops aiming to explore student experiences and whether there was a perceived increase in motivation from the viewpoint of students and instructors. Students who participated in the workshops filled out the same survey, assessing their motivation to participate in the workshop, their expectations and finally their opinion on the suitability of the tools as an entry-level teaching environment for the 'Introduction to Programming' module. Students were given a programming project to complete on their own after the end of the workshop, which they had to upload on a shared forum space on Blackboard. The rationale behind this formative assessment was to evaluate students' interest, motivation and capability to create their own game using the tool, after the end of the workshop. Also, my goal was to gauge the level of their involvement and whether they would take their training one step further, in terms of knowledge, beyond what they were taught in the workshop.

5.2.6 Action Research Cycle 3 (AppInventor)

The third action research short cycle took place during the Spring Semester 2016. I organised three short 2-hour workshops on AppInventor. One advantage of AppInventor over other visual programming environments is the possible increased motivation level which stems from creating applications that execute on a mobile device. The fact that students can create a game or an application which can be demonstrated and used by their friends and family might lead them to consider that they are not merely consumers of technology, but also producers of it (Wolber, 2011).

Twenty-five (25) students majoring in IT and two professors teaching introduction to programming attended the workshops. Although shorter in duration than the preceding cycles, student participants in this workshop showed greater involvement in the process. APPInventor utilises blocks as the basis for writing programs and students seemed to truly enjoy their interaction with the tool. During the workshop, students were introduced to the programming environment and created two mobile applications using tutorial resources from the official MIT APP Inventor website (see Appendix Five). Summative assessment scores were not available, since the workshop was not part of a module but instead open to all students that were interested in attending. The assessment grade mentioned in the table below was calculated from the optional hangman project which they were asked to complete. Data were collected from post-workshop surveys, in-class discussions at the end of the workshop and from the programming projects students completed after the workshops.

Table 5.10 shows the mean, median, standard deviation, minimum and maximum scores for intention to use, preferability over Java, overall enjoyment, tool usefulness as perceived by students who participated in the survey, as well as their score in the homework exercise and their recommendation for future adoption of the tool for the module.

VPE Visual Programming Environment - 3 App Inventor						
	Median	Mean	Standard Deviation	Minimum	Maximum	Count
q20 Intent to Use	3.00	3.27	1.34	1.00	5.00	
q22 Preferable over Java	3.00	2.88	1.31	1.00	5.00	
Overall Enjoyable	4.00	3.95	0.91	2.33	6.00	
Overall Useful	4.00	3.85	0.80	2.00	5.00	
Assessment Grade	65.00	57.88	26.80	0.00	100.00	
Recommendation for use						
No						16
Maybe						6
Yes						4

Table 5.10: Student evaluation of AppInventor programming environment
(Scale 1=Negative Opinion, 4=Neutral, 7=Positive Opinion)

The survey showed an increased motivation of students to get involved with mobile application development, but the students' recommendation to adopt the tool was still low.

Based on the fact that student perceptions about this specific kind of block-based programming was positive overall along with an assessment average score of 57.88% and in accordance with research findings, I decided to evaluate Scratch, despite the fact that its main target audience primarily spans the age group from 8 to 16 years.

5.2.7 **Action Research Cycle 4 (Scratch)**

The last cycle took place again during the Spring Semester 2016, when I organised another short 6-hour workshop on Scratch. Participation was even greater. Thirty-one (31) students from all major courses attended the workshop. During the workshop, students were introduced to the programming environment and created two programs: an IP packet switcher and a game (see Appendix Five). Data were collected from the short survey, an in-class group discussion, and scores from the formative assessment hangman project. Results demonstrated positive attitudes of students towards the usability of Scratch and a greater motivation to develop programs with it.

Table 5.11 shows the mean, median, standard deviation, minimum and maximum scores for intention to use, preferability over Java, overall enjoyment, tool usefulness as perceived by students who participated in the survey, as well as their score in the homework exercise and their recommendation for future adoption of the tool for the module.

VPE Visual Programming Environment - 4 Scratch						
	Median	Mean	Standard Deviation	Minimum	Maximum	Count
q20 Intention to Use	4.00	3.74	1.46	1.00	6.00	
q22 Preferable over Java	4.00	4.29	1.37	2.00	7.00	
Overall Enjoyable	4.33	4.33	0.80	3.00	6.33	
Overall Useful	4.33	4.38	0.74	3.00	6.33	
Assessment Grade	60.00	54.03	27.03	0.00	100.00	
Recommendation for use						
No						9
Maybe						7
Yes						15

Table 5.11: Student evaluation of Scratch programming environment
(Scale 1=Negative Opinion, 4=Neutral, 7=Positive Opinion)

5.3 Action Research Findings and Discussion

One hundred and twenty-seven (127) students in total participated in the study, of which only eighteen (18) were female and one hundred and nine (109) male, which is a representative sample of XYZ college's IT modules population (see Table 5.12 and 5.13).

Action Research Cycles – Demographics - Gender						
		Cycle 1	Cycle 2	Cycle 3	Cycle 4	
		Greenfoot	Alice	App Inventor	Scratch	Total
Gender	**Age**	**Count**				
Female	18-24	3	2	6	7	18
	24-34	0	0	0	0	0
Total Female		3	2	6	7	18
Male	18-24	28	30	18	20	96
	24-34	4	3	2	4	13
Total Male		32	33	20	24	109
Total		**35**	**35**	**26**	**31**	**127**

Table 5.12: Action research study – Demographics: Gender

Action Research Cycles – Demographics - Major					
Major	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Total
Communications	2	0	0	0	2
Economics	1	0	1	3	5
International Business	0	3	0	0	3
Information Technology	27	26	16	19	88
Management Information Systems	3	4	5	7	19
Marketing	0	2	1	0	3
Non-Degree	1	0	0	0	1
Undecided	1	0	3	2	6
All Majors	**34**	**35**	**23**	**29**	**127**

Table 5.13: Action research study – Demographics: Majors

A very interesting outcome of this research is that 72% of the students actually completed the homework (formative) assessment exercise. In the first action research cycle, 57% of the students submitted their work, as opposed to 63% for the second action research cycle. Even better results were demonstrated in the last two cycles, with 66% and 77% submission rates respectively (see Figure 5.3). At this point, I should stress the fact that attendance was "voluntary" for those last two cycles and the project was optional.

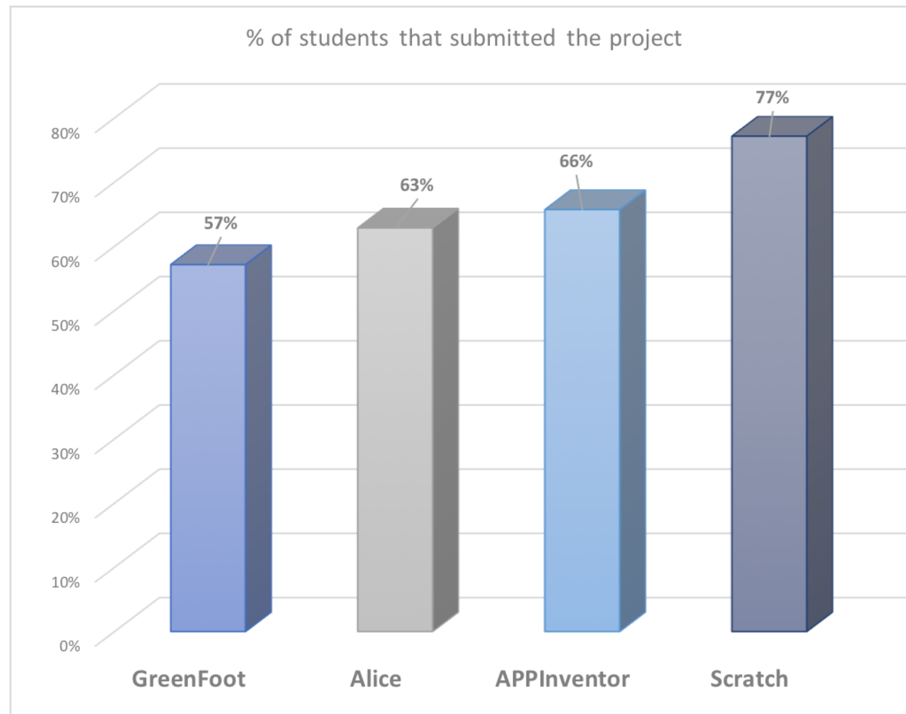


Figure 5.3: Percentage of students who submitted their project per VPE

One assumption for the analysis of the results is that the motivation to learn computer programming was not statistically different among groups. In order to test this hypothesis, I performed a non-parametric Kruskal-Wallis Test, which showed that the medians of motivation to learn programming (see Table 5.14) were the same across all four action research cycles (one per VPE).

Hypothesis Test Summary				
	Null Hypothesis	Test	Sig.	Decision
1	The medians of Motivation to learn programming are the same across categories of VPE Visual Programming Environment.	Independent-Samples Median Test	.956	Retain the null hypothesis.
2	The distribution of Motivation to learn programming is the same across categories of VPE Visual Programming Environment.	Independent-Samples Kruskal-Wallis Test	.804	Retain the null hypothesis.

Table 5.14: One-way ANOVA test for the equality of medians across VPEs

Before moving on with the analysis of the data collected from the questionnaires, Figure 5.4 shows the mean scores in each question per VPE.

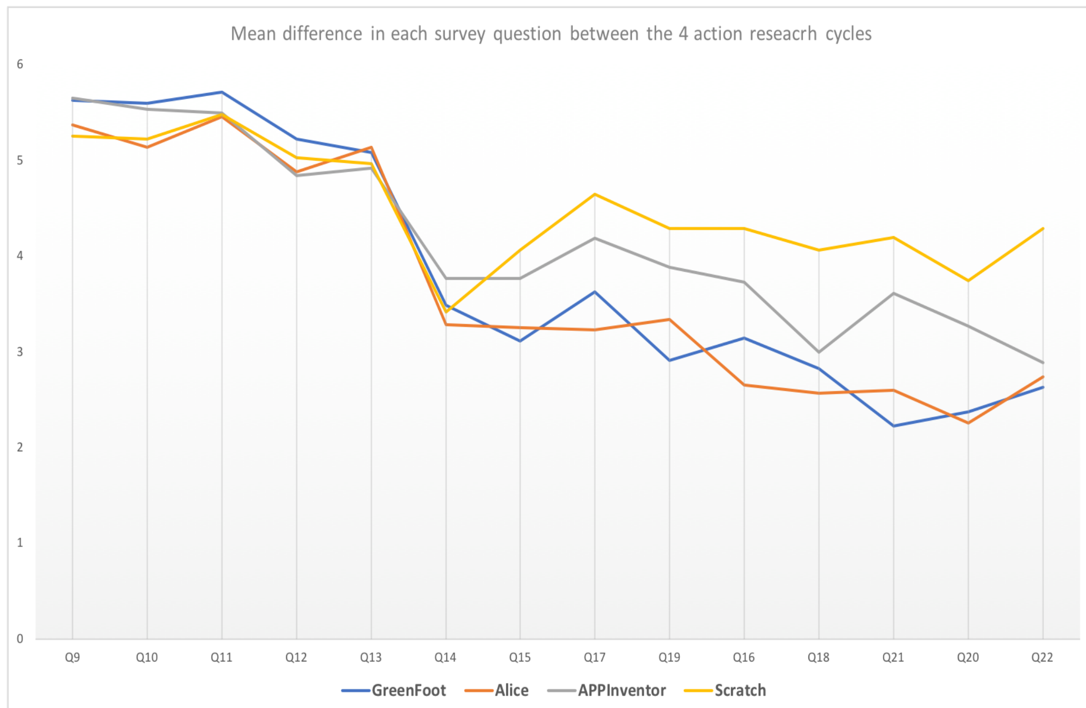


Figure 5.4: Mean score per question per VPE

By observing the line chart, we can see some differences in student opinions about each tool (q15–q22), while for the overall motivation to learn programming student opinions tend to converge (q9-q14). In order to test whether the observed difference of the means has a statistical significance and to decide on which is the most appropriate statistical test to perform, the following assumptions must be tested: a) that there are no significant outliers; b) data follows a normal distribution (Figure 5.5); and c) homogeneity variances are low.

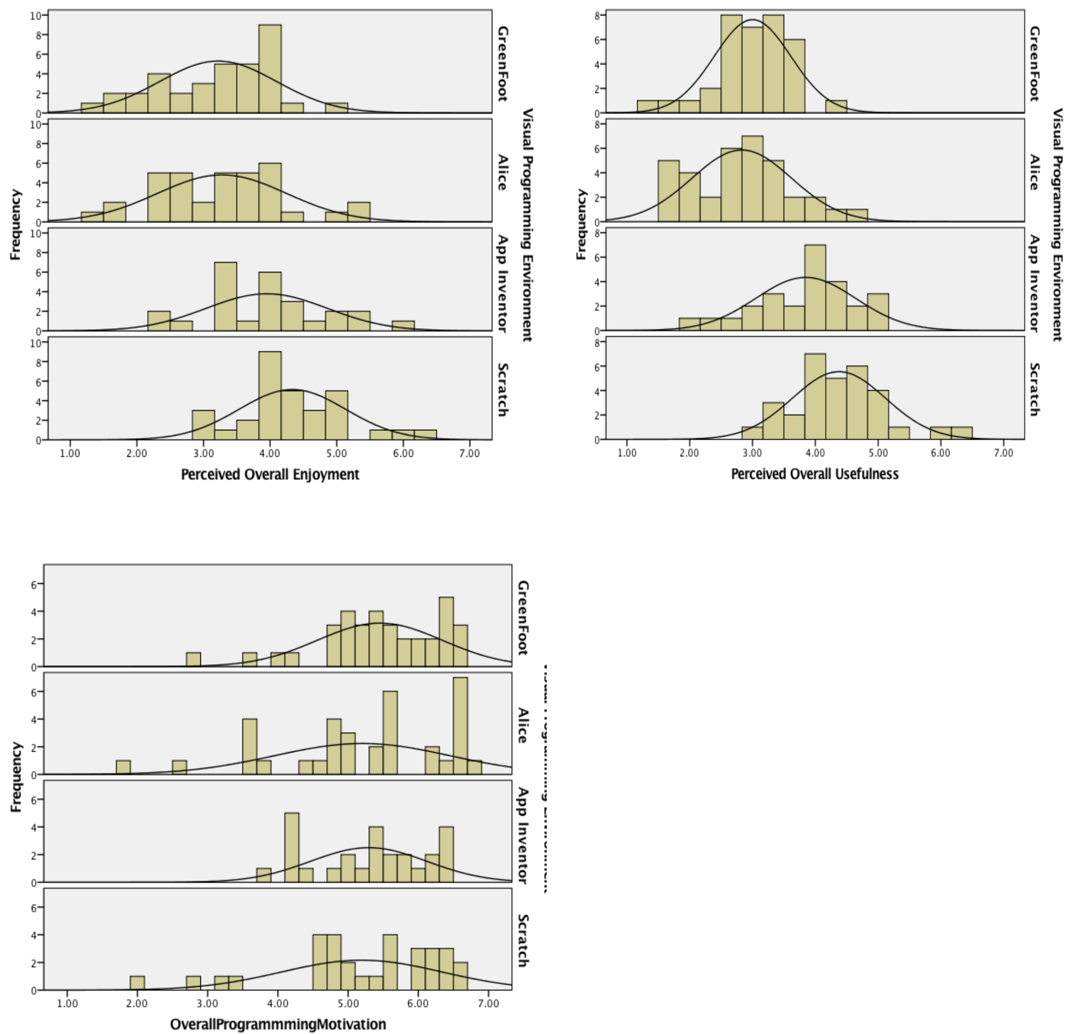


Figure 5.5: Comparison of distributions: Student rating for each VPE

The Shapiro-Wilk's test ($p > 0.05$) (Shapiro & Wilk, 1965) in Table 5.15 and a visual inspection of the histograms, normal QQ Plots and box plots, showed that mean scores for (b) perceived enjoyment and (c) perceived usefulness were approximately normally distributed for all VPEs; so, parametric tests can be employed for the comparison of their means. On the other hand, mean scores for (a) motivation to learn programming are not normally distributed, suggesting that non-parametric tests should be used to test for equality of means.

Shapiro-Wilk's Test of Normality						
	Kolmogorov-Smirnova			**Shapiro-Wilk**		
	Statistic	df	Sig.	Statistic	df	Sig.
A) Motivation to Learning Programming						
1 Greenfoot	.118	35	.200*	.925	35	.020
2 Alice	.117	35	.200*	.919	35	.013
3 App Inventor	.137	26	.200*	.920	26	.044
4 Scratch	.173	31	.019	.900	31	.007
B) Perceived Overall Enjoyment						
1 Greenfoot	.152	35	.039	.941	35	.058
2 Alice	.113	35	.200*	.966	35	.348
3 App Inventor	.135	26	.200*	.959	26	.374
4 Scratch	.145	31	.093	.943	31	.101
C) Perceived Overall Usefulness						
1 Greenfoot	.150	35	.046	.943	35	.068
2 Alice	.109	35	.200*	.953	35	.137
3 App Inventor	.192	26	.015	.946	26	.189
4 Scratch	.122	31	.200*	.955	31	.211
**. This is a lower bound of the true significance.*						
a. Lilliefors Significance Correction						

Table 5.15: Shapiro-Wilk's test of normality

The next step for the normally distributed dependent variables (b) and (c) is to test homogeneity of variances using Levene's test to check the assumption that the variances for the 4 groups are equal. The result of the Levene's Test was not significant (see Table 5.176).

(b) perceived enjoyment: $F(3/123) = 0.579$, $p=0.630$ at 0.95 alpha level.

(c) perceived usefulness: $F(3/123) = 1.088$, $p=0.357$ at 0.95 alpha level.

	Levene Statistic	df1	df2	Sig.
Perceived Enjoyment	.579	3	123	0.630
Perceived Usefulness	1.088	3	123	0.357

Table 5.16: Levene's test of homogeneity of variances

Thus, the assumption of homogeneity of variance is met and the one-way ANOVA test can be used to test the null hypothesis that the mean difference of enjoyment and usefulness across tools is not significant (see Table 5.17).

		Sum of Squares	df	Mean Square	F	Sig. (p)
Perceived Enjoyment	Between Groups	28181	3	9394	11760	.000
	Within Groups	98249	123	.799		
	Total	126430	126			
Perceived Usefulness	Between Groups	51267	3	17089	31599	.000
	Within Groups	66521	123	.541		
	Total	117788	126			

Table 5.17: One-way ANOVA test for the equality of means

Since $p < 0.001$ and thus < 0.05, which is the chosen level of significance, I can **reject** the null hypothesis that the means of perceived enjoyment and perceived usefulness between the 4 VPEs are equal.

Having understood that there is a mean difference between the four visual programming environments, the next step is to investigate which are those that cause the reported difference with a post-hoc multiple comparisons Turkey HSD test (see Table 5.18).

A Post-Hoc Multiple Comparisons - Tukey HSD Test							
Depe-ndent Variable	(I) VPE	(J) VPE	Mean Difference (I-J)	Std. Error	Sig.	95% Confidence Interval	
Perceived Enjoyment						Lower Bound	Upper Bound
	1 Greenfoot	2 Alice	-0.0571	0.2136	.993	-0.6136	0.4993
		3 App Inventor	-0.7296	0.2314	**.011**	-1.3323	-0.1270
		4 Scratch	-1.1142	0.2204	**.000**	-1.6884	-0.5402
	2 Alice	1 Greenfoot	0.0571	0.2136	.993	-0.4993	0.6136
		3 App Inventor	-0.6725	0.2314	**.022**	-1.2752	-0.0699
		4 Scratch	-1.0571	0.2204	**.000**	-1.6313	-0.4830
	3 App Inventor	1 Greenfoot	0.7296	0.2314	**.011**	0.1270	1.3323
		2 Alice	0.6725	0.2314	**.022**	0.0699	1.2752
		4 Scratch	-0.3846	0.2376	.372	-1.0036	0.2344
	4 Scratch	1 Greenfoot	1.1142	0.2204	**.000**	0.5402	1.6884
		2 Alice	1.0571	0.2204	**.000**	0.4830	1.6313
		3 App Inventor	0.3846	0.2376	.372	-0.2344	1.0036
Perceived Usefulness						Lower Bound	Upper Bound
	1 Greenfoot	2 Alice	0.1714	0.1758	.764	-0.2864	0.6293
		3 App Inventor	-0.8461	0.1904	**.000**	-1.3421	-0.3503
		4 Scratch	-1.3763	0.1813	**.000**	-1.8487	-0.9039
	2 Alice	1 Greenfoot	-0.1714	0.1758	.764	-0.6293	0.2864
		3 App Inventor	-1.0175	0.1904	**.000**	-1.5135	-0.5217
		4 Scratch	-1.5477	0.1813	**.000**	-2.0202	-1.0754
	3 App Inventor	1 Greenfoot	0.8461	0.1904	**.000**	0.3503	1.3421
		2 Alice	1.0175	0.1904	**.000**	0.5217	1.5135
		4 Scratch	-0.5301	0.1955	**.038**	-1.0395	-0.0208
	4 Scratch	1 Greenfoot	1.3763	0.1813	**.000**	0.9039	1.8487
		2 Alice	1.5477	0.1813	**.000**	1.0754	2.0202
		3 App Inventor	0.5301	.19557	**.038**	0.0208	1.0395

Table 5.18: Post-hoc multiple comparisons between VPEs - Tukey HSD test

The Turkey post-hoc test revealed that the overall perceived enjoyment of:

1) GreenFoot (M=3.219, SD=0.8776) was statistically significantly lower (-0.7296) compared to AppInventor (M=3.9487 SD=0.9125, p=0.11) and even lower (-1.1142) compared to Scratch (M=4.33, SD=0.8027, p=0.0001),

while there was no statistical difference compared to Alice (-0.0571) (M=3.2762, SD=0.9684, p=.993).

2) Alice (M=3.2762, SD=0.9684) was statistically significantly lower (-0.6725) compared to AppInventor (M=3.9487 SD=0.9125, p=0.001) and even lower (-1.0571) compared to Scratch (M=4.33, SD=0.8027, p<0.001), while there was no statistical difference compared to Greenfoot.

3) Scratch on the other hand (M=4.3333, SD=0.8027) was statistically significantly higher (1.1142) compared to Greenfoot and Alice (1.0571) but there was no statistically significant difference (0.3846) compared to App Inventor (M=3.9487, SD=0.7957, p=0.3720).

The Turkey post-hoc test revealed that the overall perceived usefulness of:

1) Greenfoot (M=3.000, SD=0.6103) was statistically significantly lower (-0.8461) compared to AppInventor (M=3.8462, SD=0.7957, p<0.001) and even lower (-1.3763) compared to Scratch (M=4.3763, SD=0.7441, p<0.001), while there was no statistical difference compared to Alice (0.1714) (M=2.8286, SD=0.7936, p=0.7640).

2) Alice (M=3.2762, SD=0.9684) was statistically significantly lower (-1.01758) compared to AppInventor (M=3.8462, SD=.7957, p<0.001) and even lower (-1.5477) compared to Scratch (M=4.3763, SD=0.7441, p<0.001), while there was no statistical difference compared to Greenfoot.

3) Scratch on the other hand (M=4.3763, SD=0.7441) was statistically significantly higher (1.3763) compared to Greenfoot (M=3.000, SD=0.6104, p<0.001), Alice (1.5477) and App Inventor (0.5302).

5.4 Conclusion

Based on the above, we can conclude that the specific groups of students did not enjoy programming using Greenfoot and Alice as much as the groups of students did using AppInventor and Scratch. As far as perceived usefulness is concerned, Scratch was deemed to be more useful for the 'Introduction to Programming' module than all the other 3 visual programming environments.

Feedback obtained from the in-class discussion indicated that, although students seemed to enjoy mobile application development using APPInventor, they did not find it appropriate for the introductory course.

The last question in the survey, *"would you recommend the addition of 'this tool' as part of the teaching material for the Introduction to Programming module?"* can be used to verify the results obtained from the statistical tests. Eighty per cent, 71% and 62% of the students would not recommend Greenfoot, Alice and AppInventor respectively for the introduction to programming module, while only 29% of the students were negative about Scratch. The results are depicted in Figure 5.6.

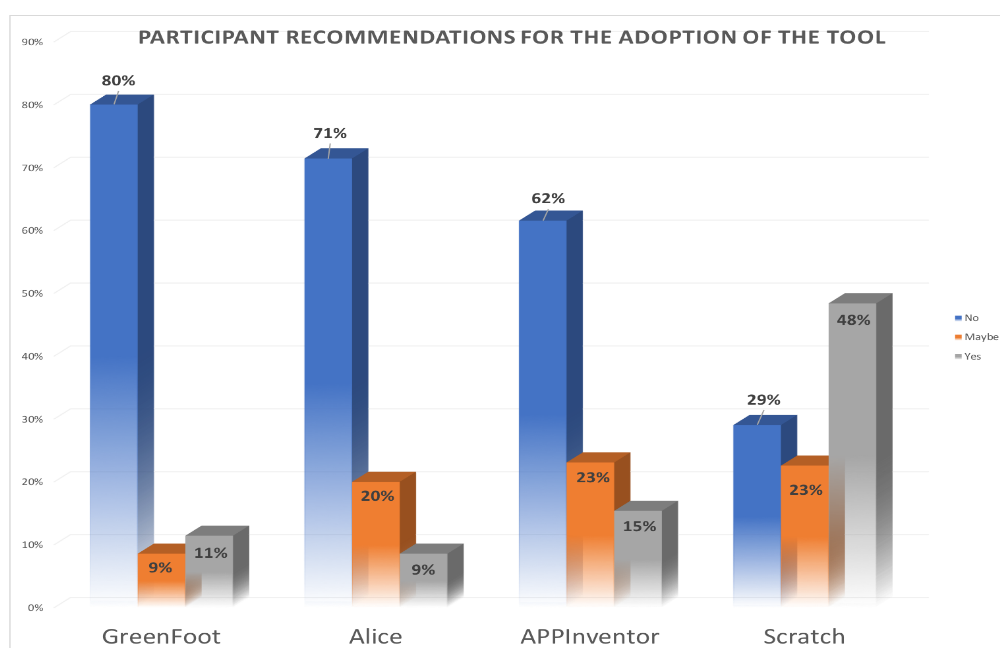


Figure 5.6: Participant recommendations for the adoption of each tool

The findings described above were instrumental in my decision to use Scratch as the Visual Programming Environment tool of choice for my main study.

It should be noted that a limitation of the data analysis carried out was that data obtained from the two "workshop" cycles, which informed the study about overall student perceptions around AppInventor and Scratch enjoyability and usefulness, cannot be accurately compared to that of the first and second action research cycles, in which the visual programming environment was actually incorporated into the material of the module.

Another limitation of this preliminary study might stem from the divergence across the in-class activities used in Alice, Greenfoot, APPInventor and Scratch. The nature of the programming activities performed during the workshops as well as the user interface and capabilities of each VPE might have affected student perceptions.

The "childish" interface of Alice (according to student comments) might have predisposed them to reject the tool, without carefully considering its capabilities to demonstrate and apply advanced programming concepts. On the other hand, the task developed in Scratch, using an Internet Protocol (IP) packet switching "computing concept" compared the "game" development activities demonstrated using Greenfoot, as well as the "fun" activities developed in APPInventor, might have altered learners' perceptions.

To help mitigate the bias resulting from the variation in the activities undertaken across action research cycles, students were introduced to the same concepts in all VPEs (see Appendix Five). Furthermore, students were given the same final assignment and graded using the same rubric (see Table 5.19). It thus seems unlikely that the variation in the learning activities would have led to a significant difference in the results.

Experience gained from the design, execution and data analysis of this preliminary research, in addition to the findings reported by existing literature, informed the case study design and formed the basis of my subsequent research.

Chapter 6 Research Design and Methodology

6.1 Purpose

The purpose of this chapter is to present and justify the research design and evaluate the suitability of the proposed methodology for conducting this type of research, as well as the reasoning by which case study was identified as the most appropriate methodology to evaluate and explain why the proposed innovation succeeded or failed to motivate students when learning how to program.

In addition, the chapter provides readers with background information and an explanation of the rationale for using mixed methods for data collection and the strategy behind the data collection process. It concludes with a detailed description of the steps taken to develop the questionnaires used to implement quantitative data collection, as well as the interview protocol used to perform qualitative data collection.

6.2 Case Study and Data Collection Approaches

Research methodology is defined by Leedy and Ormrod (2010) as *"the general approach the researcher takes in carrying out the research project"* (p. 14). The selection of a research methodology is based on the subject, the nature and the aims of the research questions being addressed, including the theoretical and philosophical assumptions upon which research is based and it will provide the general framework guiding the research project.

A case study methodology is considered to be appropriate when a researcher wishes to examine a unique issue or phenomenon in detail, as well as its real-life manifestation (Baxter & Jack, 2008). Additionally, a case study is a design of inquiry found in many fields, especially evaluation, in which the researcher develops an in-depth analysis of a case, often a program, event, activity, process, or one or more individuals (Stake, 1995).

Yin (2003) defines a case study as "an empirical inquiry that aims to investigate a contemporary phenomenon in-depth and within its real-life

context, especially when the boundaries between phenomenon and context are not clearly evident".

According to Yin, five components are crucial in a case study design:

1. The study's research questions;
2. Its propositions;
3. Its unit of analysis;
4. The logic linking the data to the propositions; and
5. The criteria for interpreting the findings.

Cases are bounded by time and activity, and researchers collect detailed information using a variety of data collection procedures over a sustained period of time (Stake, 1995; Yin, 2003).

Using case study as my methodological approach, I can study the complexity of learning programming as perceived by students (my unit of analysis) as well as their motivation to learn. Further, I can explore a possible connection between students' preference for visual programming environments with their learning styles, while observing their behaviour, and keeping track of their performance in this unique situation.

The phenomenon under investigation is unique as far as the group of individuals that will be studied, their age group, gender, ethnicity, their role in the class and XYZ college in which the study takes place is concerned. These independent variables cannot be controlled and might have an effect on the results obtained by the study.

Case study methodology has previously been used to explore topics including education and teaching of programming (Hadjerrouit, 2007; Jones, 2010; López *et al.*, 2016; Pellas & Peroutseas, 2016). While the focus of this research is mainly grounded in the IT field and more specifically in computer programming, the flexibility of the case study methodology will enable cross-disciplinary themes to be addressed such as educational and motivational theories and their implications. It could be argued that since human behaviour is a such a complex phenomenon, statistics alone cannot adequately describe it. As a result, blending both qualitative and quantitative methods can help

researchers enhance the understanding of technical and behavioural aspects (Seaman, 1999). Case study methodology allows for a mixed method data collection strategy as the exactness of quantitative, and 'richness' of qualitative, approaches can be combined (Runeson, 2012).

At the same time, according to Eisenhardt (1989), a major limitation of a case study design is that the results obtained, although very rich in detail, might lack the simplicity of a generalised perspective or may result in a very narrow and idiosyncratic theory.

However, a methodological debate is found in the literature, where different authors identify distinct themes which are used to categorise the direction, organisation and design of case studies. Thomas (2011) provides a table summarising the characteristics of most recent general themes in the methodological debate. I used Table 6.1 as a tool to identify the common categories and I highlighted and emboldened the ones that fit my methodological approach.

Merriam (1988)	Bassey (1999)	de Vaus (2001)	Yin (2009)	Creswell (2011)
Descriptive	Educational theory seeking	Descriptive / **Explanatory**	Critical	Convergent
Interpretative	Theory testing	Theory testing / Theory building	Extreme/ **unique**	**Explanatory**
Evaluative	Storytelling	**Single Case** / Multiple case	Longitudinal	Exploratory
Concrete and Contextual	Picture drawing	Holistic/embedded	Representative	Embedded
	Evaluative	Parallel / **Sequential**	Revelatory	Transformative
		Retrospective/ prospective		Multiphase

Table 6.1: Case study designs/themes

In the context of this research, I share Merriam's (1998) view that a case study is particular (concrete and contextual), descriptive and heuristic, and as such, it can be used to evaluate and explain why an innovation worked or failed to work, as well as to summarise and make conclusions.

The evaluative nature of case studies in educational settings is also discussed by Bassey (1999) referring to Stenhouse's (1978) views, who claims that the purpose of an evaluative case study is to provide teachers (and other educational actors) with information that will help them judge the worth of a program (or a policy or even an institution).

As far as the data collection and analysis is concerned, three generalised categories or so-called strategies of inquiry are found in the literature: quantitative, qualitative and mixed methods.

One simplified distinction between quantitative and qualitative informing results is that, to explore and understand a case, the quantitative data rely on numbers while their qualitative counterpart rely on words. Creswell (2014) notes that quantitative research is an approach for testing objective theories by examining the relationship among variables, while qualitative research is an approach for exploring and understanding the meaning individuals or groups use to describe to a social or human problem, while mixed (hybrid) methods reside in between the two approaches by incorporating elements from both.

Mixed methods data techniques involve collecting both quantitative and qualitative data, integrating the two forms of information, and using distinct designs that may involve philosophical assumptions and theoretical frameworks. The core assumption of this form of inquiry is that the combination of qualitative and quantitative data provides a more complete understanding of a research problem than either approach could provide if applied alone. More specifically, Creswell (2014) mentions that *"mixed methods involve combining or integration of qualitative and quantitative research and data in a research study. Qualitative data tends to be open-ended without predetermined responses while quantitative data usually includes closed-ended responses such as found on questionnaires or psychological instruments" (p.43).*

A literature review on the quantitative/qualitative debate shows that a researcher can use mixed methods as a means to attain meaningful and valid

results and to answer pertinent research questions. Any quantitative measure can be expressed qualitatively, and any qualitative measure can be expressed in a quantitative manner. Krauss (2005), Creswell and Clark (2011) and Robson and McCartan (2016), support the argument that mixed forms of evidence will lead us to a comprehensive understanding of the problem and extract meaning from “the real world”. Merton and Kendall (1946) express the same sentiment that social scientists have come to abandon the spurious choice between qualitative and quantitative data: they are rather concerned with the combination of both that makes use of the most valuable features of each. The problem becomes one of determining at which points they should adopt the one, and at which the other, approach.

This research study follows an explanatory sequential mixed method design for data collection. Explanatory mixed methods are those in which the researcher first conducts quantitative research, analyses the results and then builds on the results to explain them in more detail with qualitative research. It is considered explanatory because the initial quantitative data collected for the survey will provide a more general statistical picture of the variables, which can then be explained further with the qualitative data to provide us with a more in-depth understanding of student perceptions, thus following the evaluative (Bassey, 1999) and explanatory (Creswell & Clark, 2011) sequential framework.

The timing of the research is sequential (quantitative followed by qualitative data collection). The quantitative part will be used to provide the general statistical picture of the phenomenon under investigation as well as identifying possible participants for the qualitative part. The qualitative part, on the other hand, will be used to explore the participant views in-depth to look to explain the statistical results obtained from the survey.

Based on the above, my overarching methodological approach is an evaluative case study and the data collection follows an explanatory sequential mixed method design. The case study is within the context of the ‘Introduction to Programming’ module at an English-speaking institution of higher learning in Southern Europe, college XYZ and is based upon participatory action research practice. The population being studied are students registered in the module for four consecutive semesters. Scratch software was used to enable students to undertake visual programming. My research questions in this context were:

RQ1: How do visual programming environments affect students' performance in the course (assessment and final grades)?

RQ2: How do students perceive the Scratch visual programming environment?

> a) How do students perceive enjoyability, ease of use, usability and usefulness?
>
> b) How do students relate these qualities to their achievement of the module's learning objectives (output quality)?

RQ3: How does students' motivation for learning programming relate to their perceptions about visual programming environments?

RQ4: How do students' learning styles relate to their perceived enjoyment, ease of use, usability and usefulness of Scratch visual programming environment?

The dependent variables and the methods which will be used for their analysis, in order to address the above-mentioned research questions, are:

- Students' perceptions about Scratch visual programming environment's enjoyability, ease of use, usability and usefulness, measured both quantitatively using data collected from the survey (Technology Acceptance Model part) and qualitatively using semi-structured interviews to address research question 1.
- Students' performance in the course, measured quantitatively using assignment and examination scores (leading to final course grades). This can be compared to students' performance in previous semesters (before the introduction on the visual programming environments), to address research question 2: Students' performance in the Scratch assessment compared to their performance in a Java assessment.
- Students' motivation for learning programming is measured quantitatively using data collected from the survey (Motivated Strategies for Learning) and explored qualitatively using semi-structured interviews and class observations to address research question 3.

- Students’ learning approaches, measured quantitatively using data collected from the survey (Learning Styles Questionnaire), compared to students’ perceptions about the Scratch visual programming environment’s enjoyability to address research question 4.

Data collected from the semi-structured interviews are studied in depth in order to identify the variations in students’ perceptions about programming in general as well as about visual programming environments and to form an outcome space. On the other hand, data collected from the surveys are analysed quantitatively using statistics. Data collected from the formative examinations will be used to inform the research about the possible variations between students’ perceptions and their actual performance in an examination setting.

Given this perspective, I will summarise my own stance about the overall design of this research in Figure 6.1.

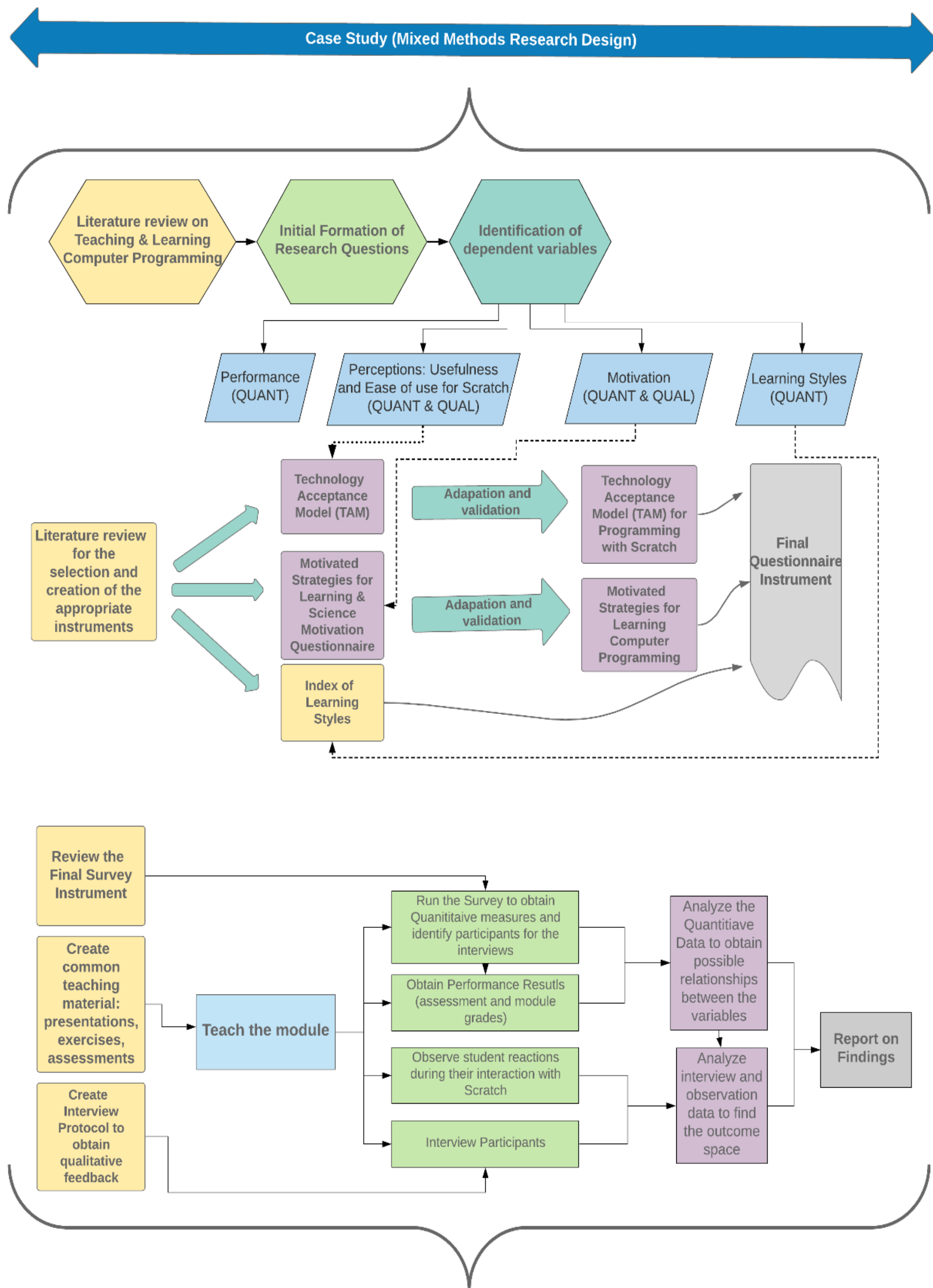


Figure 6.1: Case study research design

6.3 Pedagogic Design: Teacher's Role and Students' Activity

From Fall Semester 2016 onwards, Scratch 2.0 was used during the first two weeks of instruction of the introduction to programming module, spanning six theory and four laboratory sessions (a total of ten instructional hours).

The teaching pedagogy of this part of the module combined elements of design (prescribed tasks) and improvisation (within pre-designed learning activities). This approach promoted a creative class environment in which students proposed or recommended next steps in an activity, especially because it involved game development. The prescribed content outline is presented as follows.

Theory Session 1: Introduction to the environment, description of code-blocks and practice with the code editor, using sprites, costumes, changing backgrounds.

> Activity 1: Understand/predict the output of the Scratch program.
>
> Activity 2: Execute the program to visualise the output and fix the logical error.

Theory Session 2: Introduction to basic programming constructs, such as variables, input, output, conditions, loops and basic event handling available in Scratch toolbox.

> Activity 1: Write the pseudocode for a Body Mass Index calculator. Implement the pseudocode using Scratch.
>
> Activity 2: a step-by-step tutorial of how to create a pong game.

Laboratory Session 1 – assignment: solve the maze (level of difficulty: easy).

Laboratory Session 2 – assignment: create a birthday cake (level of difficulty: easy).

Theory Session 3: Explanation of more advanced programming concepts, such as arrays, cloning (instantiation) and message-broadcasting. All concepts were demonstrated by the professor using live coding.

Activity 1: Read the specifications and create an Internet Protocol (IP) packet switcher, using a Domain Name Server (DNS) resolver array.

Laboratory Session 3 – assignment: create a fruit ninja game (level of difficulty: medium).

Laboratory Session 4 – assignment: create a hunting game (level of difficulty: medium-hard).

Theory Sessions 3-4-5: In-class group work - Donkey-Kong inspired platform game. Students work in pairs towards the development of a Donkey-Kong inspired platform game. The instructor's role in this phase was more of a facilitator than a teacher, assisting whenever students did not know how to progress.

While students worked on their computers during laboratory sessions, instructors kept general notes on their interaction with the program, their emotional expressions, their levels of attention and perseverance, and their performance (see Appendix Three).

Keeping notes of human behaviour imposes a limitation on the study due to inherent partiality of the observer; furthermore, the process could not be exhaustive in terms of data gathering, given that there was only one observer for the over fifteen students in the classroom. Performing audio-visual recordings could have been used to overcome this limitation (Cohen *et al.*, 2013) and multimodal discourse analysis (Kress & Van Leeuwen, 2001) could have enriched the study with additional perspectives including the analysis of student interactions with the environment (recorded using screen capture software) and the recording of student facial expressions and verbal comments (recording using computer cameras).

Visual research methods would have provided a rich amount of data for analysis; however, constrained by the fact that the study took place within a formal classroom setting, video or screen recording might have proved to be obtrusive to the lesson, and would additionally require the consent of all the students. Furthermore, according to Bassey (1999), making it obvious to subjects that they are being recorded might instigate a change in behaviour.

At the end of the two weeks, students were assigned the first part of their summative coursework: to develop a hangman game in Scratch, utilising a fixed dictionary of ten words. The program had to randomly pick a word from the dictionary and the user had to guess the word. User input was to be validated and compared to the letters of the word picked, allowing one to evaluate the appropriate use of strings and conditionals by the programmer. Ten tries were allowed in each game, thus demonstrating the appropriate usage of repetition. Code had to be documented using comments. Modularity of the code was also a factor to be assessed. As an additional challenge, students were asked to propose and implement extra functionality to enhance their game.

Scratch coursework assessed students' knowledge of all concepts taught: arrays; random numbers; conditions; loops; event-handling; message broadcasting; cloning; timers; custom blocks; game mechanics (score, win/lose conditions); and code documentation. This coursework part accounted for 20% of the students' final grade.

After the 2 weeks of VPE instruction, students progressed to learning how to program using Java (refer to section 1.3). For their Java coursework assessment, students were required to implement the same hangman game. The Java assessment accounted for 40% of the students' final grade.

Students were also assessed with a midterm examination in pseudocode and Java, accounting for the remaining 40% of the students' final grade.

6.4 Development of the Questionnaire Survey Tool

To identify student perceptions of the enjoyment, ease of use, usefulness, output quality and attitude towards using Scratch, Davis's Technology Acceptance Model was adapted and validated. Details about adaptations follow in sub-section 6.3.3 concerning the questionnaire Section 2 - Overall Evaluation and Acceptance of Scratch and in sub-section 6.3.4 concerning questionnaire Section 3 - Perceived Ease of Use and Perceived Usefulness.

To identify student motivations to learn programming, a mixture of questions from Motivated Strategies for Learning Questionnaire (MSLQ) (Pintrich & de Groot, 1990b) and from the Science Motivation Questionnaire (SMQ-II) (Glynn *et al.*, 2009) were adapted. The process of the identification and selection of questions included, follows in sub-section 6.3.5 concerning questionnaire Section 4 - Motivated Strategies for Learning.

Finally, to identify student learning styles, Felder and Soloman's Index of Learning Styles instrument was adopted without modification (see sub-section 6.3.6 concerning questionnaire Section 5 - Index of Learning Styles).

All three instruments, along with general demographic information were included in the final survey instrument and administered to students enrolled in the 'Introduction to Programming' module.

The finalised survey contains six main sections and was administrated online using the Qualtrics Survey Platform of Lancaster University.

6.4.1 Section 0 – Participant Information Sheet and Consent Form

Section 0 contains the participant information sheet where the students are informed of the purpose of the study and are requested to check all questions.

Participants that provide at least one negative response to the questions above are immediately disqualified and are transferred to the "Thank you" exit page. The participant information sheet, as well as the concept form, obtained ethics clearance from ethics committees of both Lancaster University and XYZ College where the study took place.

6.4.2 **Section 1 - Participant Demographic Information**

In section 1, participants were asked five demographic questions, one question concerning the reason they took the course, one question about their current programming level (which branched to which programming languages they were already being taught and whether they were familiar with block-based programming in the past), three Likert-type scale questions based on their overall opinion about Scratch and their intentions of using it in the future, and two open-ended questions about Scratch features that they found useful and ones they disliked (possible perceived barriers).

6.4.3 **Section 2 - Overall Evaluation and Acceptance of Scratch**

Questions included in Section 2 are based on Davis's overall system evaluation in the Technology Acceptance Model (TAM) (Davis, 1985). TAM has been widely applied to identify user attitudes towards the use of technology and to predict the adoption of a system (Chang & Cheung, 2001; Wixom & Todd, 2005; Shroff *et al.*, 2011; Weng *et al.*, 2018). Wording of the questions was modified to fit the context under investigation.

The final survey tool section 2 contained five questions (Q15 – Q21) aiming to measure "attitude" towards using Scratch, utilising semantic differential (bipolar) rating scales, based on Martin Fishbein and Icek Ajzen's theory of reasoned action (Fishbein & Ajzen, 1975) and Osgood measurement techniques of belief, attitude, intention and behaviour (Osgood *et al.* 1957).

Enjoyment according to the definition provided by the Collins English dictionary is the "feeling of pleasure and satisfaction that you have when you do or experience something that you like" (Collins English Dictionary, 2019). Carroll and Thomas add that in order for students to engage in activities and consider them fun *"is all right to fail"* (Carroll & Thomas, 1988). The same view is supported by Deci (1976) stressing that there is external reward related to the "fun" activity apart from the feeling of competency. Davis *et al.* (1992) align with the views of Deci (1976), Malone (1981) and Carroll and Thomas (1988) that perceived enjoyment could be considered as an example of intrinsic motivation, whereas perceived output quality, relevance and

effectiveness could be considered as examples of extrinsic motivation. Both of them have been included in this study following the TAM2 technology acceptance model (see Figure 6.2) to measure perceived enjoyment and output quality along with ease of use and usefulness to support the learning objectives of the introduction to programming module.

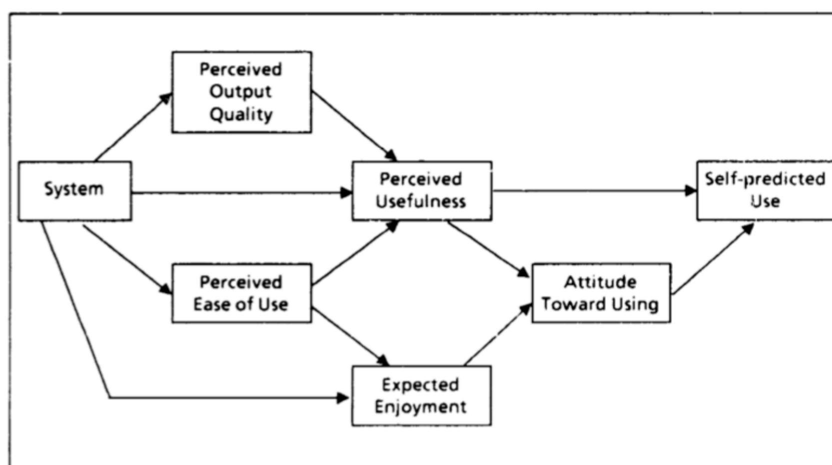


Figure 6.2: TAM2 extended to include enjoyment and output quality

Osgood has proven that the semantic differential approach with five items (five bipolar pairs of adjectives) yields reliable findings, which highly correlate with alternative Likert numerical measures of the same attitude (Osgood *et al.*, 1957). Examples of responses in the form of adjective pairs have been found to reflect the evaluation or judgement about an object, concept, or behaviour along a dimension of favour or disfavour, good or bad, like or dislike, enjoyable or unenjoyable, desirable or undesirable, good or bad, pleasant or unpleasant, relevant or irrelevant, interesting or not interesting on the Semantic Differential (SD) scale.

The reason to adopt the specific adjective pairs: boring-fun, ineffective-effective, unenjoyable-enjoyable, irrelevant-relevant, unpleasant-pleasant is three-fold. Firstly, their validity has been established in previous research (Davis, 1985; Igbaria *et al.*, 1995; Chang & Cheung, 2001; Wixom & Todd, 2005). Secondly, they reflect motives of using technology derived from a larger pool extracted from past research which is similar to this research context. Lastly, they were selected from a larger item pool of adjectives as being the most representative ones, using a card-sorting survey. Ten professors who teach various programming modules (subject field experts) at XYZ college

were asked to choose 6 adjectives from a larger pool and place them in the two categories (enjoyment/output quality). The larger pool contained also the adjectives: efficient, beneficial, important, interesting and demonstrable. The results using a standardisation matrix are shown in Table 6.2.

Standardisation Matrix		
Variable name	**Construct: Enjoyment**	**Construct: Output Quality**
Beneficial	1	3
Demonstrable	3	4
Effective		9
Efficient		5
Enjoyable	9	
Fun	10	
Functional		10
Important	2	3
Interesting	5	1
Pleasant	9	
Relevant		10

Table 6.2: Standardisation matrix - Card Sorting

As a result, the overall attitude construct included in the survey encompasses an enjoyment sub-construct (questions 15, 17 and 19) and a cognitive instrumental process sub-factor (questions 16, 18, 21). Both enjoyment and cognitive instrumental processes (output quality, result demonstrability, relevance) have been shown by prior studies to significantly influence user acceptance (Venkatesh & Davis, 2000).

6.4.4 Section 3 - Perceived Ease of Use and Perceived Usefulness

In Section 3, questions were again adapted from Davis's Technology Acceptance Model (1985).

Davis (1985), in his doctoral thesis, proposed that an information system's user acceptance can be predicted by user motivation. He also argued that user motivation is influenced by an external stimulus of the actual system's features and capabilities (see Figure 6.2).

The technology acceptable model contains twelve Likert-type questions of the same scale. Davis's technology acceptance survey (Davis, 1989) consists of two factors: perceived usefulness and perceived ease of use and six (6) statements in which the (…) ellipse can be replaced by the system under consideration for user acceptance.

Perceived ease of use (PEU)

- EASE1: Learning to operate the (. . .) is easy for me
- EASE2: I find it easy to get the (. . .) to do what I want it to do
- EASE3: Usage of the (. . .) is clear and understandable
- EASE4: I find it cumbersome to use the (. . .)
- EASE5: It is easy for me to remember how to perform tasks using (. . .)
- EASE6: Overall, I find the (. . .) easy to use

Perceived Usefulness (PU)

- USE1: Using (. . .) enables me to accomplish tasks more quickly
- USE2: Using (. . .) improves my job performance
- USE3: Using (. . .) increases my productivity
- USE4: Using (. . .) enhances my effectiveness on the job
- USE5: Using (. . .) makes it easier to do my job
- USE6: Overall, I find (. . .) useful in my job

I studied the questions originally created and tested by Davis during the development of the tool, as well as a number of other similar questions adapted by subsequent studies.

Keeping similar wording where possible, I included five questions concerning student opinion about Scratch's perceived ease of use. The selection of the final statements, as they appear below, was finalised after a focus-group review

session with the four fellow professors (subject field experts) who participated in the selection of the adjectives for the previous survey section.

- Q23: Learning to operate Scratch is often frustrating. (EASE1)
- Q24: It is easy for me to remember how to perform tasks inside the Scratch Environment. (EASE2)
- Q25: I find it easy to get Scratch to do what I want it to do. (EASE3)
- Q26: Usage of Scratch is clear and understandable. (EASE4)
- Q27: Overall, I find Scratch easy to use. (EASE5)

Again, maintaining similar wording where possible, I included five questions concerning student opinion about Scratch's perceived usefulness.

- Q28: Using Scratch helped me improve my computing skills. (USE1)
- Q29: Scratch makes it easier for me to convey an algorithm into a program, rather than using a text-based programming language. (USE2)
- Q30: Scratch improved my understanding of all critical aspects of the software development process (which are the main learning outcomes of this module). (USE3)
- Q31: Scratch makes it easier for me to understand the main programming concepts (variables, loops, decisions, etc.). (USE4)
- Q32: Overall, I find Scratch useful for this module. (USE5)

The version of the TAM used to evaluate the perceived enjoyment, output quality, ease of use and usefulness of Scratch is depicted in Figure 6.3. The arrows which demonstrate the relationships between the variables are missing, since the scope of this study was not to create and verify a model for the acceptance of Scratch rather than relate the TAM variables to the participants' learning styles.

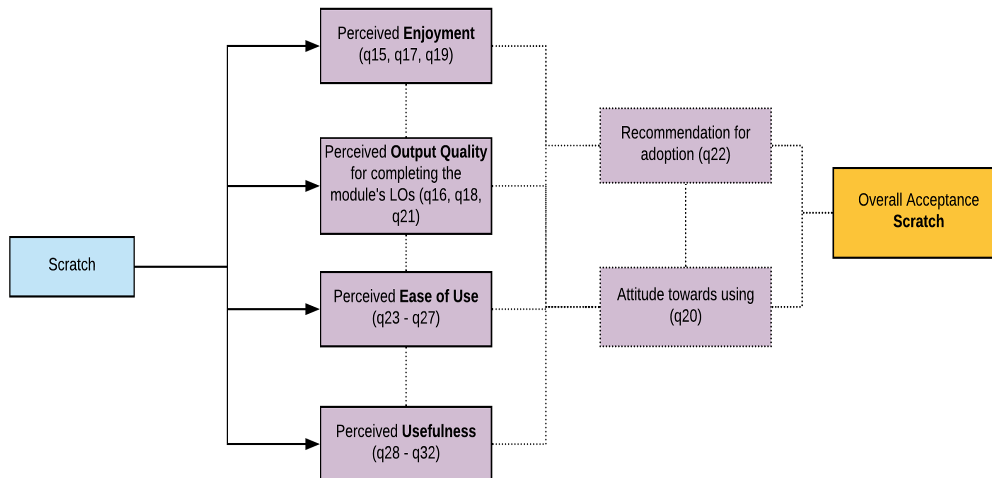


Figure 6.3: TAM of Scratch

6.4.5 Section 4 - Motivated Strategies for Learning

In order to standardise terminology between the two distinct tools, I will use the word "category" in this section to represent the meaning of components or scales or summative scales and the word "statement" to represent the items or the questions of the questionnaire.

The MSLQ is a self-reporting instrument developed to measure students' motivation, orientations and use of learning strategies. The first version of the Motivated Strategies for Learning Questionnaire (MSLQ) (Pintrich & de Groot, 1990) contained 5 categories: intrinsic value, self-efficacy, test anxiety, cognitive strategy for use and self-regulation, with a total of fifty-six statements. The final version (Pintrich *et al.*, 1991) is composed of two sections. The motivation section contains six categories and a total of thirty-one statements, while the learning strategies section contains nine categories and a total of fifty statements. More specifically:

The motivation section contains three main components which include the size scales mentioned above:

1) a value component which includes scales for:
   a) intrinsic goal orientation,
   b) extrinsic goal orientation, and

    c) task value;
2) an expectancy component which includes scales for:
    a) control of learning beliefs,
    b) self-efficacy for learning and performance,
    c) an affective component which includes a scale for, and
    d) test anxiety.

The learning strategies section includes two main components:

1) the cognitive and metacognitive strategies component which include scales for:
    a. rehearsal,
    b. elaboration,
    c. organisation,
    d. critical thinking, and
    e. metacognitive self-regulation;
2) the resource management strategies component which includes scales for:
    a. time and study environment,
    b. effort regulation,
    c. peer learning, and
    d. help seeking (Duncan & Mckeachie, 1991; Pintrich *et al.*, 1991).

The Science Motivation Questionnaire II, on the other hand, contains five categories and each category is composed of five statements, totalling twenty-five statements, related to intrinsic motivation, self-efficacy, self-determination, grade motivation and career motivation.

These two questionnaires have common categories: intrinsic motivation, task value, extrinsic motivation, self-efficacy, self-regulation and self-determination. Both tools consist of statements and use Likert-type scales to obtain user input which are reflecting extreme positions on a continuum across which people are likely to agree (very true of me) or disagree (not at all true to me). Summative scores are constructed by taking the mean of the statement scores that make

up each category. Both tools have been tested multiple times for their validity and reliability by their creators (Glynn *et al.*, 2009; Pintrich *et al.*, 1993; Glynn *et al.*, 2011).

There are three main reasons for not applying the scales from the existing questionnaires. The main reason for creating a new questionnaire is because I identified a gap in the literature in finding a tool to access student motivation in learning how to program. Secondly, the statements of the existing questionnaires did not fully address the topic of computer programming and, finally, not a single questionnaire addressed all motivational components required for this research. For example, career motivation exists only on the SMQ-II and is directly related to extrinsic motivation. Intrinsic motivation is addressed in the Science Motivation questionnaire in a way that lacks the component of task value, while the Motivated Strategies for Learning Questionnaire lacks the component of career motivation. The motivational components that influence learning provided the basis of the selection of the main categories: intrinsic motivation (including task value), self-efficacy, self-determination and extrinsic motivation (grade and career motivation). The existing statements were rephrased to include "computer programming" or "learn how to program" concepts, in order to make them more specific.

The first step taken to create section 4 of this survey was to merge all existing statements in the categories of interest and attempt to establish face and content validity. As mentioned previously, content validity can be measured by relying on the knowledge of people who are familiar with the construct being measured. Eight experts in the field of education reviewed all statements for readability, clarity and comprehensiveness. Experts reviewed all questions by grading them as "essential" (score of 1), "useful but not essential" and "not necessary" (score 0) in order to measure student motivation. A Content Validity Ratio (CVR) was calculated for all statements using the formula: (Score of items - Total number of panellists /2) / (Total number of Panellists / 2) (Lawshe, 1975).

Given the table provided by Lawshe (1975), the minimum CVR required for any item to be included in a questionnaire is 0.75 when the number of reviewers (panellists) are eight. As a result, all questions with a CVR >= 0.75

were selected to be included in the subsequent test for scale reduction and are highlighted in Appendix Two.

In a second test for content validity, 15 professionals in the areas of educational psychology and 15 educators in the areas of computing, information systems and informational technology were asked to select the five most representative items out of the ones which had a CVR >= 0.75 in each of the six motivational components. Following the recommendations of Hinkin (1998), the goal could be the retention of four to six items per construct. Schriesheim (1995) and Hinkin (2006) also support that, although including more items might increase the internal consistency of a single construct, a lengthy questionnaire can maximise the bias caused by boredom and fatigue (Schmitt & Stuits, 1985). The resulting scales were composed of the 5 top-rated questions in the sections of intrinsic motivation (see Figure 6.4), self-efficacy (see Figure 6.5), self-determination (see Figure 6.6) and extrinsic (career and grade) motivation (see Figure 6.7). In the figures following, the top 5 questions which were selected for the study are highlighted with blue color. Reversed questions are marked with (*R).

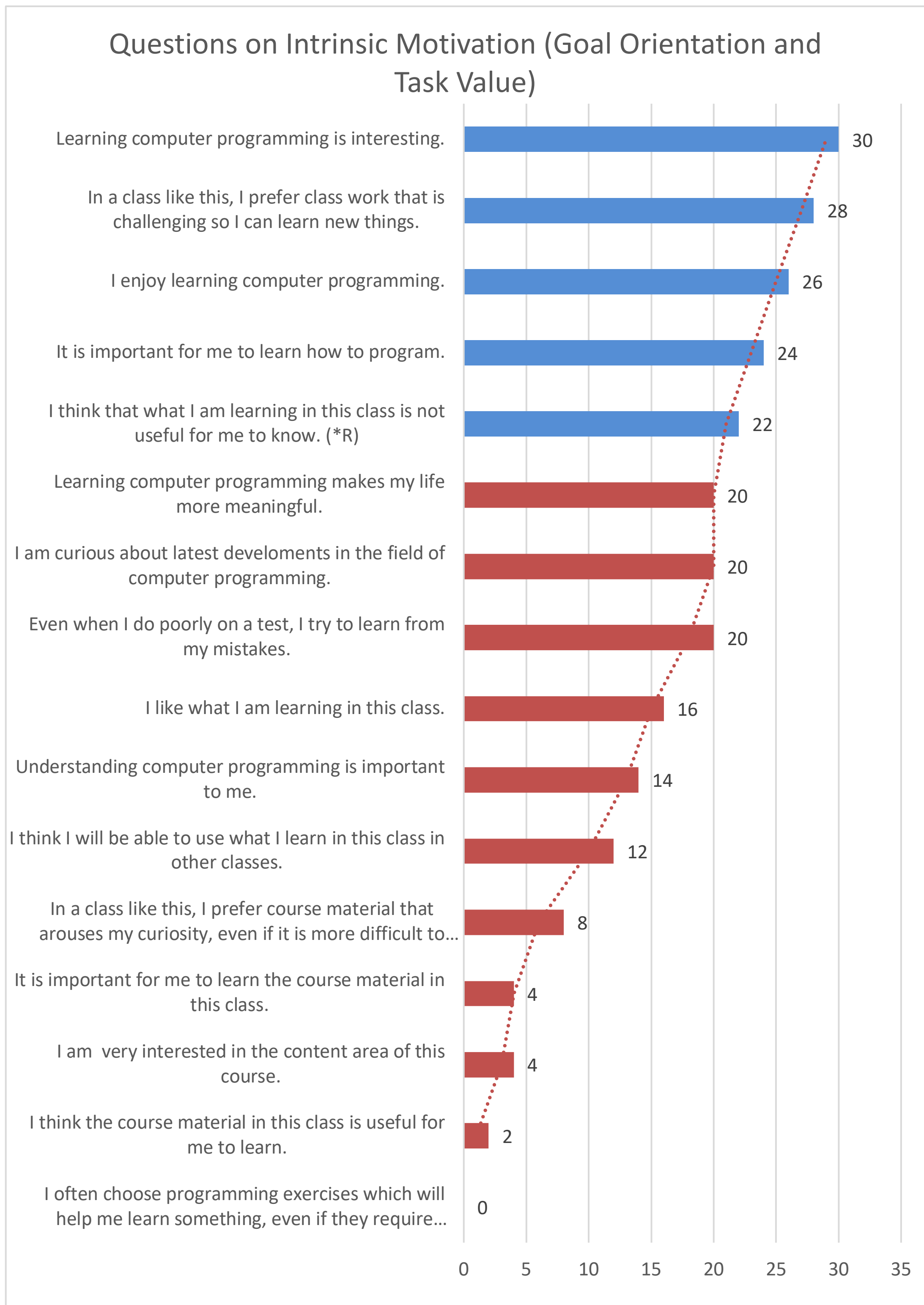


Figure 6.4: Intrinsic motivation scores

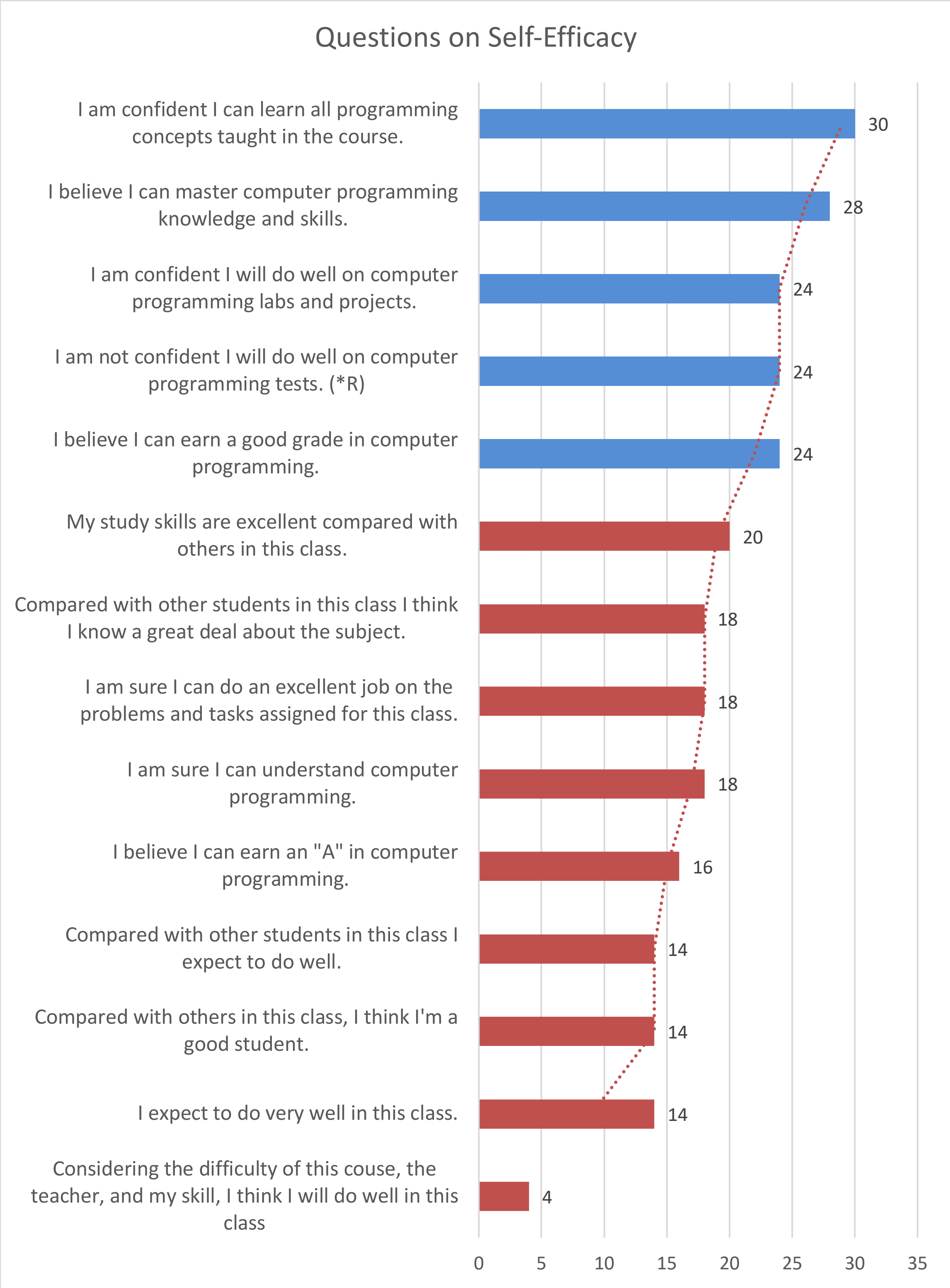


Figure 6.5: Self-efficacy scores

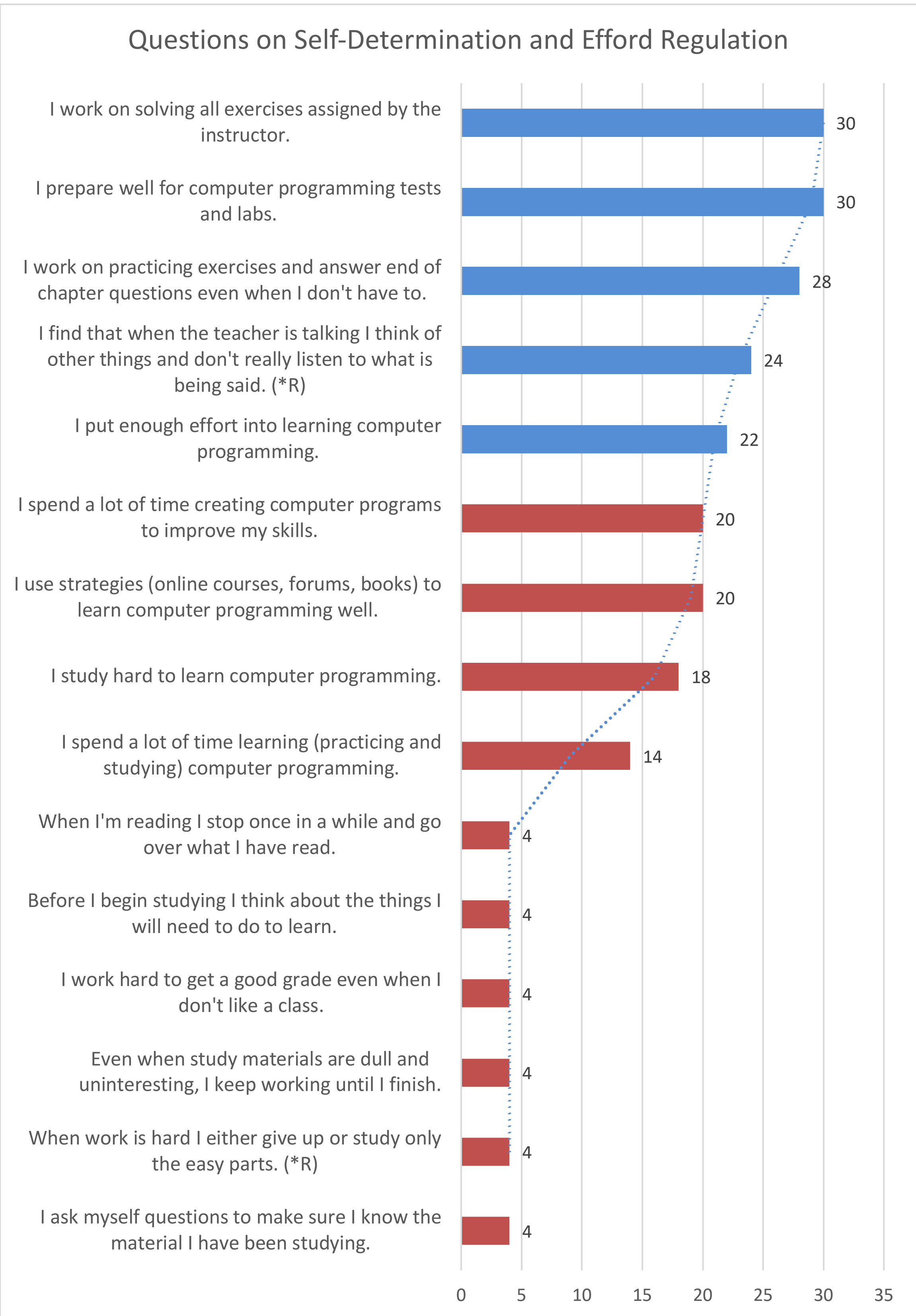


Figure 6.6: Self-determination scores

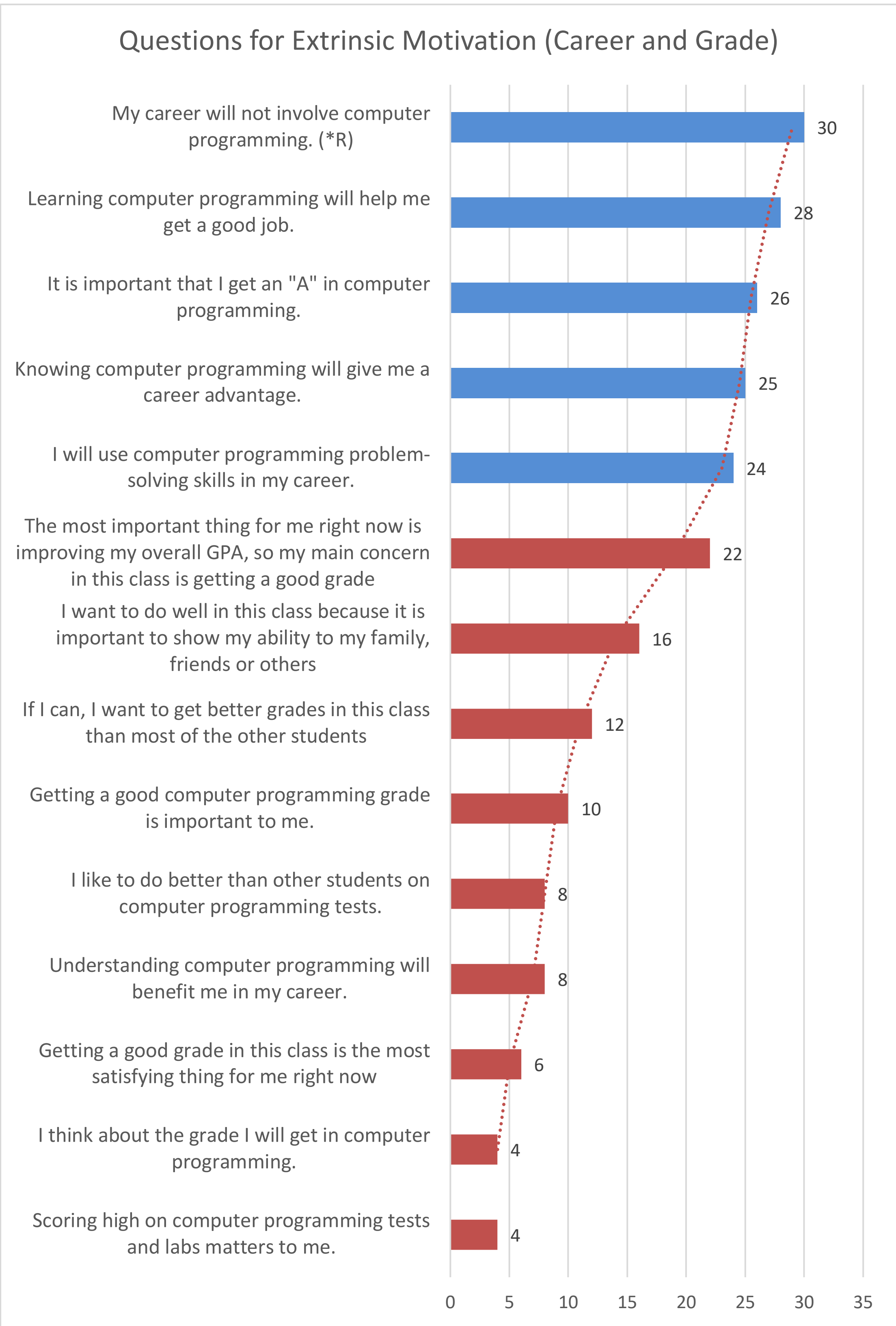


Figure 6.7: Extrinsic Motivation (Career and grade) scores

To identify students who check the Likert-scale values without reading the questions carefully, one item in each scale has been reversed and the statement has a negative meaning (shown with *R). Examples of reversed questions are q35, q39, q44 and q49 (see Appendix One, Section 4). The ratings of these reversed statements were reversed before computing the individual scores on the summative scales.

6.4.6 **Section 5 - Index of Learning Styles**

In section 5 of the questionnaire, I applied Felder and Soloman's Index of Learning Styles instrument (Felder & Soloman, 1993) to collect information about student learning styles. The aim was to compare the data obtained with the students' learning styles in order to identify possible patterns and verify if there was some correlation between students' learning styles and their preference towards visual programming environments.

The Index of Learning Styles® (ILS) is a **forty-four-item** forced-choice instrument developed in 1991 by Richard Felder and Barbara Soloman to assess preferences on the four scales of the Felder-Silverman model, discussed in section 2.2.

The classification of students in each dimension (visual/verbal, active/reflective, sequential/global, intuitive/sensing) is based on the answers they provide to these questions. Each learning style dimension score is calculated by adding up the individual scores of 11 yes/no questions that represent that dimension.

For example, a score ranging from 0 to 11 in the visual/verbal dimension will place the student somewhere in a line from strongly verbal (0) to strongly visual (11), from strongly reflective (0) to strongly active (11), from strongly global (0) to strongly sequential (11) and from strongly sensing (0) to strongly intuitive (11) (Felder, 2005). Moderate preference for learning in a particular style (score 2-3 on the left or 8-9 on the right) or mild preference (score 4-5 or the left or 6-7 on the right) is also calculated and reported. Mild preferences do not generally classify a person in any of the two poles in that dimension. On

the other hand, a learner's strong preference for a learning style might expose learning difficulties in an environment which does not support that style.

The questions can be found in Appendix One in Section 5.

6.5 Validity and Reliability

6.5.1 Validity

Polit and Beck (2004) define validity as the degree to which an instrument measures what it is designed to measure. Cronbach and Murphy (1970) state that *"the end goal of validation is explanation and understanding"* and their views are in accordance with Messick (1987) who describes a test's validity in terms of "construct validity". A construct is a hypothetical characteristic of the participants taking the test, assumed to be measured in the test's results (Cronbach & Meehl, 1955). Essentially, the main question is *"does the test measure the construct it is supposed to measure?"* Wainer and Braun (2013) also agree that all information collected about a test can contribute to the overall understanding of its construct validity, which includes all forms of validity evidence (content-related, criterion-related and construct-related). More specifically:

- content-related validity can be evaluated based on professional judgments about the content relevance and appropriateness of the test's items with regards to the construct being measured (Messick, 1987; Polit & Beck, 2004);
- criterion-related validity can be evaluated by comparing test scores with external variables (criteria) which can also measure the qualities under investigation (Messick, 1987); and
- construct-related validity can be evaluated by examining which qualities the test measures and the degree to which the test scores relate to the theory that defines these qualities (Cronbach, 1957).

There are many different methods by which researchers can address the issues of validity and reliability, although their inherent weaknesses cannot be

completely removed (Cohen *et al.*, 2013). These methods can be used to examine both internal and external construct validity and include among others: content validity ratios (CVR), test/retest, confirmatory factor analysis, group differences, correlation matrices, comparison with external criteria, analysis of variances and alpha coefficients.

It should be noted that both Gronlund (1971) and Messick (1987) claim that validity should be seen as a matter of degree and not as an absolute value.

6.5.2 **Validity Issues Addressed in this Study**

The TAM questionnaire, which was used as the basis for section 2 in the questionnaire, was tested for construct validity during its initial development (Davis, 1985) and in numerous studies afterwards (Davis, 1989; Davis, Bagozzi, & Warshaw, 1989; Venkatesh & Davis, 2000; Wixom & Todd, 2005). Additionally, to further verify the scales, a confirmatory factor analysis was used to test whether or not the data collected from the questionnaire fit the hypothesized measurement model and as such to confirm construct validity of the tool.

Table 6.3 presents the Cronbach Alpha coefficients for TAM scales.

TAM Scales	Cronbach Alpha	N of Items
Ease of use (q23, q24, q25, q26, q27)	0.840	5
Usefulness (q29, q28, q30, q31, q32)	0.946	5
Enjoyment (q15, q19, q17)	0.952	3
Output quality (q16, q18, q21)	0.943	3

Table 6.3: Cronbach alpha for TAM

Table 6.4 presents the factor loadings of the questionnaire and the four components extracted: Usefulness, Ease of Use, Enjoyment and Output Quality.

Rotated Component Matrix				
	1 – Usefulness	2 – Ease of Use	3- Enjoyment	4- Output Quality
Q29	**.856**			
Q28	**.848**			
Q30	**.840**			
Q31	**.835**		.310	
Q32	**.785**		.356	.350
Q25		**.842**		
Q24		**.827**		
Q26		**.786**		
Q27		**.734**		
Q23		**.682**		
Q15			**.892**	
Q19			**.891**	
Q17	.392		**.809**	
Q16				**.914**
Q18				**.841**
Q21	.347			**.832**
Extraction Method: Principal Component Analysis. Rotation Method: Varimax with Kaiser Normalization. a Rotation converged in 5 iterations.				

Table 6.4: CFA for TAM

The obtained values were above the recommended level of .70, thus indicating adequate internal consistency (Cronbach, 1951; Peterson, 1994; Tavakol & Dennick, 2011). This pattern of high reliability and validity is consistent with much prior research (Davis, 1989; Venkatesh & Davis, 2000; Wixom & Todd, 2005). The scales were also tested for convergent validity, considering the factor loadings, composite reliability (CR) and the average variance extracted (AVE) (see Table 6.5).

TAM Scales	**AVE**	**CR**	**SQRT(AVE)**
Usefulness	0.694178	0.918958	0.833173
Ease of Use	0.748009	0.898859	0.864875
Enjoyment	0.694178	0.918958	0.833173
Output Quality	0.744967	0.897408	0.863115

Table 6.5: AVE and CR values for TAM

The calculated AVE values exceeded the recommended value of 0.50 and CR values exceed 0.70, so the questionnaire scales can be considered as adequate (Fornell & Larcker, 1981; Hair *et al.*, 2016).

For the purpose of obtaining content validity regarding the selection of the adjective pairs used in the TAM section of the questionnaire, 10 professors from college XYZ provided their feedback by performing a card sorting exercise (see Table 6.2).

The MSLQ and Science Motivation Questionnaire, the tests from which the motivation section items were selected, have been examined for generalisability, content, face, structural, construct and predictive validities during their development studies and beyond (Pintrich *et al.*, 1993; Glynn *et al.*, 2009; 2011; Taylor R., 2012; Salta & Koulougliotis, 2015).

For the purpose of obtaining content validity for the motivation section of the questionnaire (using questions from both MSLQ and the Science Motivation Questionnaire) and with a goal to retain four to six items per construct, 30 professionals (subject experts) provided their feedback which resulted in the selection of the most representative items per construct using the content validity ratio (CVR) method as described in sub-section 6.4.5 .

Four IT professors, including myself, who have extensive teaching background in introduction to programming and other programming courses, reviewed the resulting questionnaire and made appropriate suggestions, which were taken into consideration.

Finally, a pilot survey was conducted among 4 senior IT graduates to determine whether there were any misconceptions in the wording of the statements and to test the effort required to complete the questionnaire. Feedback from the pilot survey resulted in minor revisions to the questions and the removal of 3 items.

Before analysing all collected data, the resulting motivation scales were tested to measure the internal consistency among the items of the scales using Cronbach alpha coefficients (see Table 6.6).

Motivation scales	Cronbach's Alpha	N of Items
Intrinsic value (q39, q41, q43, q47, q52)	0.825	5
Extrinsic value (q35, q37, q40, q46, q53)	0.840	5
Self-regulation (q36, q38, q45, q49, q50)	0.805	5
Self-efficacy (q34, q42, q44, q48, q51)	0.888	5

Table 6.6: Cronbach alpha's for Motivation Scales

A confirmatory factor analysis also verified that the questions fitted into the four scales. Table 6.7 presents the factor loadings for each extracted component.

Rotated Component Matrix				
	1. Self-Regulation	2. Extrinsic	3. Self-Efficacy	4. Intrinsic
Q39				0.858
Q41				0.787
Q43				0.690
Q47				0.822
Q39				0.703
Q37		0.642		
Q40		0.860		
Q46		0.699		
Q35		0.660		
Q53		0.548		
Q36	0.644			
Q38	0.775			
Q49	0.453			
	1. Self-Regulation	2. Extrinsic	3. Self-Efficacy	4. Intrinsic
Q50	0.834			
Q54	0.817			
Q34			0.781	
Q42			0.568	
Q44			0.783	
Q45			0.784	
Q48			0.526	
Q51			0.720	
Extraction Method: Principal Component Analysis. *Rotation Method: Varimax with Kaiser Normalization. a Rotation converged in 13 iterations.*				

Table 6.7: CFA for Motivation scales

The scales were also tested for convergent validity considering the factor loadings, composite reliability (CR) and the average variance extracted (AVE) (see Table 6.8).

Motivation Scale	AVE	CR	SQRT(AVE)
Intrinsic	0.6000	0.8820	0.7747
Extrinsic	0.5360	0.8490	0.7321
Self-regulation	0.5170	0.8370	0.7188
Self-efficacy	0.5810	0.8920	0.7620

Table 6.8: AVE and CR values for motivation scales

The calculated AVE values exceeded the recommended value of 0.50 and CR values exceed 0.70, so the questionnaire scales can be considered as adequate (Fornell & Larcker, 1981; Hair *et al.*, 2016).

The Index of Learning Styles questionnaire was also examined for validity during its development. Construct validity, test-retest reliability, internal consistency and inter-scale orthogonality measurements have been carried out by a number of researchers as stated by Felder (2005) and since this questionnaire was used without any modifications, no further tests for validity were performed.

6.5.3 **Reliability**

Joppe (2000) defines reliability as the extent to which test results can be consistently reproduced under similar methodologies and to which the data can be collected from a representative sample of the population under study. Reliability in quantitative research differs from reliability in qualitative research. In quantitative research reliability can be addressed in terms of replicability over time and over groups of responders (Cohen *et al.*, 2013).

6.5.4 **Reliability Issues Addressed in this Study**

In the context of this study, and to ensure reliability, two different checks were made. The first check was performed to assure stability and replicability over time. Six students took the survey twice, within a period of a month. Their results were tested for deviations. Five students were more or less consistent in all of their answers, while one student had some differences in the individual answers for the acceptance of Scratch (TAM) but the overall average in each scale was very close.

The second test was performed to ensure replicability over groups of respondents. Eight different groups of students participated in the research and consistently produced similar results. These students were registered in eight different classes of the 'Introduction to Programming' module during four semesters. Two classes were taught by me and six by other professors of college XYZ. The conditions under which the data collection took place were standardised. The survey took place in the classroom at the end of the semester and during the last 30 minutes of the instruction period. Students who did not wish to participate were allowed to leave the room. To ensure that I (as the researcher) was not affecting the test results, the mean scores in the 8 groups (in all sections of the survey) were tested and found that they did not vary significantly. Additionally, the mean grades for coursework, midterm and final grades were tested for mean differences using a t-test and no statistically significant differences were found.

6.6 Qualitative Data Collection - Interviews

6.6.1 Interview Protocol

Students to be interviewed were selected from the survey by indicating their intention to participate in an interview. They were contacted through email to make a face-to-face appointment. In this email, students were informed about the location, purpose and duration of the interview.

All interviews took place in the same environment (an office at XYZ College) with which students and professors are familiar. Before the interview, I addressed terms of confidentiality, and obtained participant permission to voice-record the interview. Before initiating the interview, participants were asked if they had any questions concerning the study or the protocol.

6.6.2 Interview Questions for Students

Although the interviews were semi-structured, I used the following questions as a general guideline:

- What is your perception about programming?
- Have you attended a course on Scratch in the past?
- Overall, did you enjoy working Scratch? If so, why? If not, why not?
    - Do you find Scratch easy to use? Why?
    - Do you find Scratch useful for the specific module? Why?
    - Do you find Scratch interesting? If yes, mention some characteristics of Scratch that made it interesting for you…
- Which are the major disadvantages you see in the use of Scratch in the 'Introduction to Programming' module?
- Where you motivated to use Scratch outside the scope of this module? To develop your own games…
- Did you try to further enhance (at home) Scratch projects we developed in class?

- Which of the two coursework assessments (Scratch/Java) did you spend more time developing? Why?
- Which of the two coursework assessments did you enjoy developing more?

6.7 Ethical Framework

My research was carried out in a real-world situation involving real students participating in a required course for their major and involved open communication among the people involved. I, as the researcher, paid close attention to ethical considerations in the conduct of my work. Having in mind a number of principles concerning research ethics, I followed the guidelines presented by Winter (Winter, 1987).

All the involved parties in my research (students, teachers, and XYZ College administration) were fully informed about the aims of my research and requested in advance to give me permission to conduct my research, using the participant information sheet.

- Ethical approval was obtained from both the University of Lancaster and the XYZ college ethics committee (see Appendix Four).
  - The letter of approval obtained from the University of Lancaster confirmed that the study could be conducted ethically.
  - The letter of approval obtained from XYZ College's ethics committee confirmed that the research could be carried out using a sample group of students attending the 'Introduction to Programming' module and professors with teaching programming experience. XYZ college also approved the survey questionnaire and interview protocol process.
- All participants were you allowed to influence my research (I did not exclude any student who volunteered to participate).
- Students who did not wish to participate were respected and their decision did not in any way affect their course grade.

- My research progress was visible and open to suggestions from others.
- I obtained written permission before making any in-class observations, interviews or using survey results.
- Participants were reported anonymously. Participants' names and addresses were omitted from the data and did not appear on any documents other than the consent forms, which were stored in a private, secure cupboard. The anonymity of information provided was taken into consideration at all stages of the study, including transcription, coding and data analysis, as well as writing up the results.
- Students were allowed to read their own interview transcripts before they were used in my research and/or published.
- I accepted full responsibility for maintaining confidentiality.
- All student feedback was immediately downloaded and deleted from the online survey tool as soon as each semester survey was closed.
- Survey results were kept in password encrypted Microsoft (MS) Excel files. No personal student information was kept in the MS Excel files.
- In the transcription of the interviews, each student name was replaced with a participant number.

6.8 Conclusion

This chapter presented the research design and justified the selection of an evaluative case study as the overarching methodological approach and an explanatory sequential mixed method design as the data collection strategy. It provided a detailed description of the development process of the questionnaire tool and presented the interview protocols. It addressed validity and reliability issues and concluded with a description of the ethical framework.

The next chapter proceeds with a description of the data gathering process and a detailed analysis of the data collected from student surveys, interviews and class observations.

Chapter 7 Data Analysis and Findings

7.1 Introduction to Data Analysis

> *"Data analysis is the process of making sense out of the data. And making sense out of data involves consolidating, reducing, and interpreting what people have said and what the researcher has seen and read – it is the process of making meaning."* (Merriam, 1998)

In order to answer the research questions, four sources of data were used as input: student grades and results from the surveys, which were analysed quantitatively, and interview transcripts and class observations, which were analysed qualitatively. More specifically:

1) Student grades were used for the overall comparison of student performance before and after the introduction of Scratch, and for the comparison of student performance in the Scratch coursework to that in the Java coursework, to address RQ1: *"How do visual programming environments affect students' performance in the course (assessment and final grades)?"*

2) Results from student surveys were used in:

   i) identifying overall acceptance of Scratch as a teaching and learning tool, to address RQ2: "*How do students perceive visual programming environments?"*

   ii) identifying student learning styles and their possible correlation to Scratch acceptance: enjoyment, ease of use, usefulness and output quality, to address RQ4: *"How do students' learning styles relate to their perceived enjoyment, ease of use, usability and usefulness of visual programming environments?"*

   iii) identifying student motivation to learn programming and its possible correlation to their perceptions about Scratch, to address RQ3: *"How do students' motivations for learning programming relate to their perceptions about visual programming environments?"*

3) In addressing RQ2 and RQ3, feedback provided by students during interview sessions and analysis from class observations were used to provide a better insight into student perceptions about Scratch VPE enjoyment and motivation and to complement the quantitative results.

7.2 Data Gathering and Demographics

Data were gathered during the Fall Semester 2016, Spring Semester 2017, Fall Semester 2017 and Spring Semester 2018. Each semester, there are two classes of the 'Introduction to Programming' module, with a maximum of 18 students in each. Each class has its own timetabled sessions and might be taught by a different professor. Students register for a specific module class, and then attend the same one for the duration of the semester. The assessments and the module outline are common to each class. Formal class contact hours per semester are composed of thirty-five lecture hours and twenty-four laboratory sessions.

The number of participants per session is shown in Table 7.1, as well as the total number of students registered. Four different professors taught these sessions and their names were replaced by letters to maintain anonymity. Ninety-two of the 113 students registered in the module through the years agreed to participate in the study and provided their feedback using an online survey software (Qualtrics) hosted at Lancaster University. Twelve students volunteered to be interviewed and enrich this study with their qualitative feedback.

Academic Year	Semester	Module Occurrence	Number of Participants	Number of students registered
2016-2017	Fall 2016	A, Prof a	15	16
		B, Prof d	14	14
	Semester Total		**29**	**30**
	Spring 2017	A, Prof a	13	16
		B, Prof c	5	7
	Semester Total		**18**	**23**
2017-2018	Fall 2017	A, Prof a	14	16
		B, Prof d	9	16
	Semester Total		**23**	**32**
	Spring 2018	A, Prof d	12	12
		B, Prof b	10	16
	Semester Total		**22**	**28**
Total number of participants			**92**	**113**

Table 7.1: Number of participants across the years of the study

The following figures depict demographic information provided by students in section 1 of the survey (refer to Appendix Two) concerning their age range (see Figure 7.1); gender (see Figure 7.2); and academic major (see Figure 7.3).

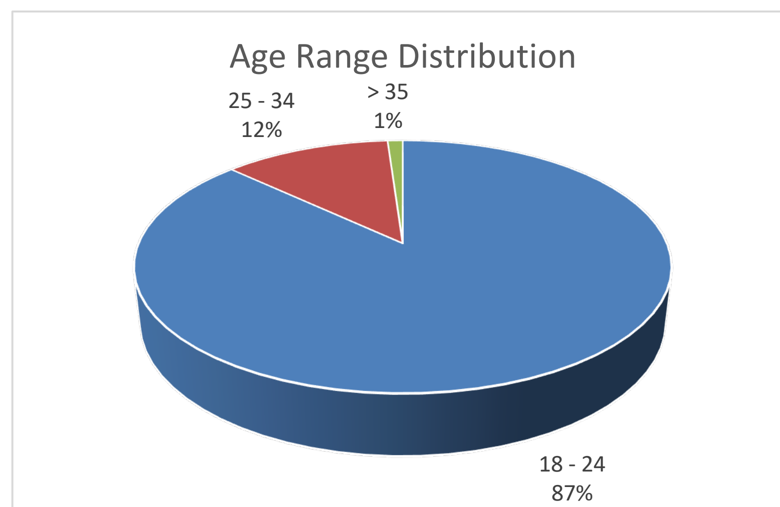


Figure 7.1: Participant age distribution in year ranges

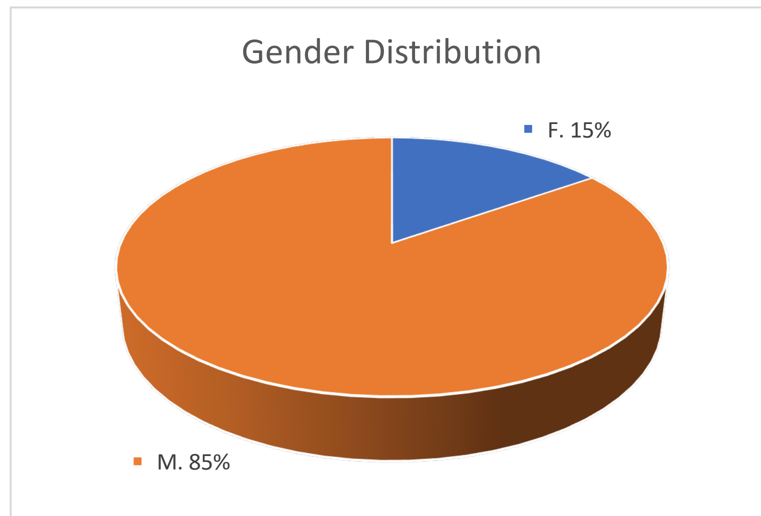


Figure 7.2: Participant gender distribution

Gender distribution for the IT course has been the same for more than 10 years with male students outnumbering female ones.

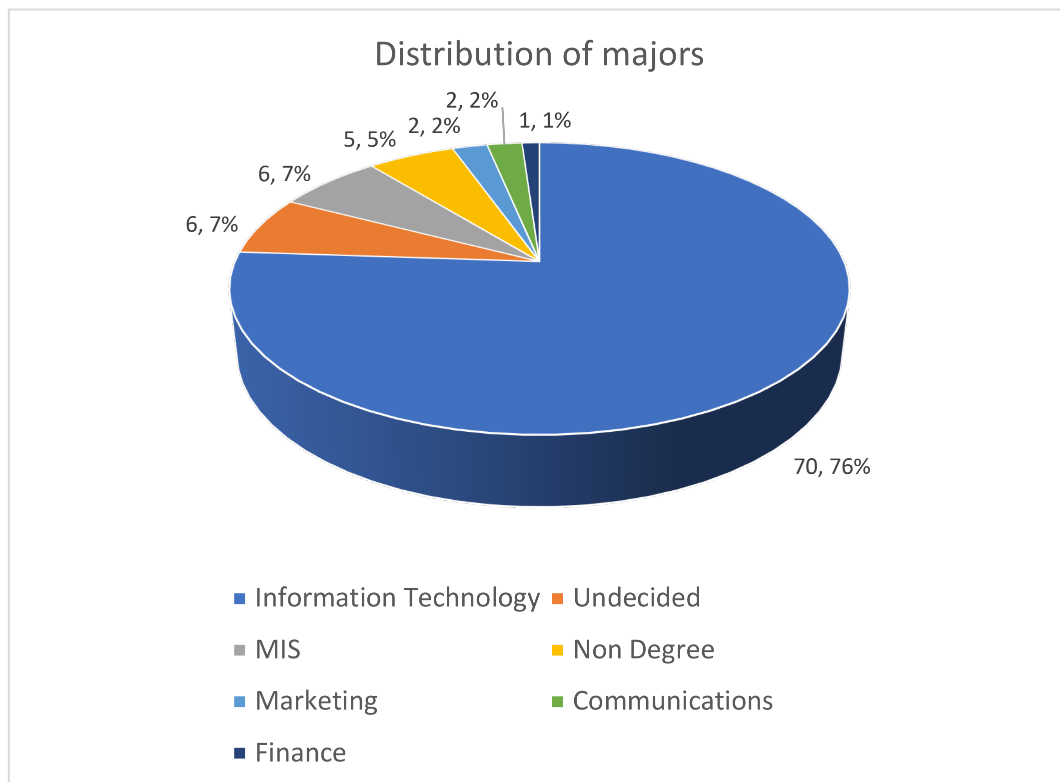


Figure 7.3: Participant distribution of majors

The distribution of majors is as expected, because the 'Introduction to Programming' module is a requirement for students majoring in IT and an elective for all other students.

Section 2 of the survey asked students to describe their perceived level of programming expertise and whether they were familiar with the Scratch programming environment.

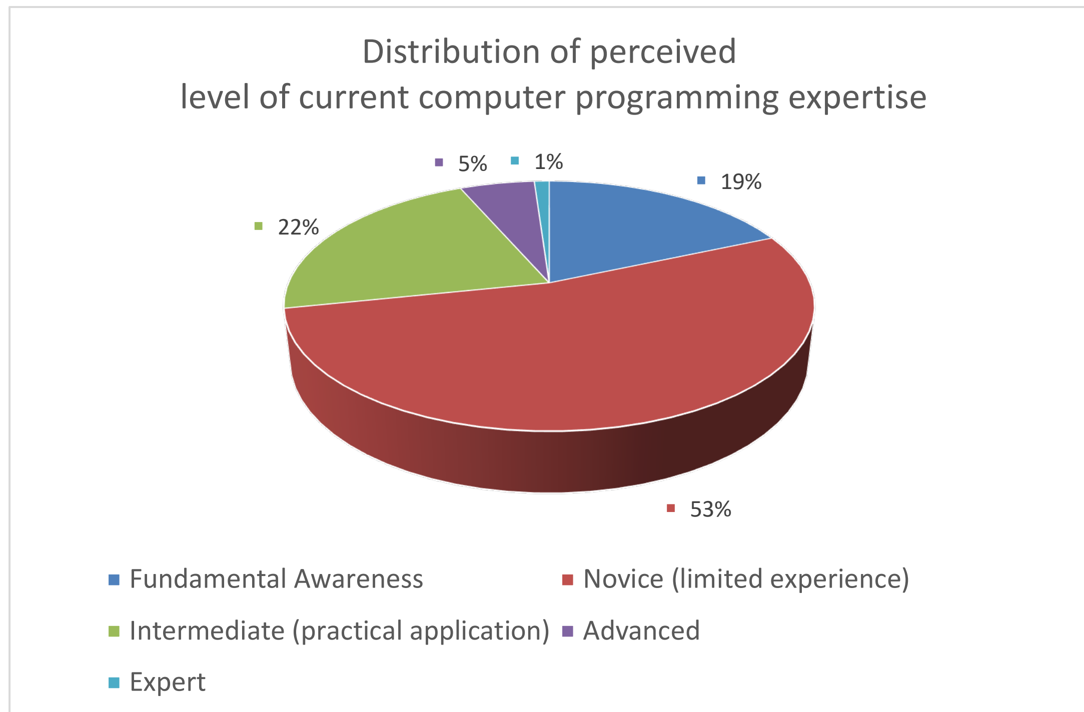


Figure 7.4: Participant perceived current computer programming level of expertise

The distribution of programming expertise in Figure 7.4 does not represent a formally assessed evaluation but how students self-evaluated their expertise. An explanation of each selection was included in the survey (Appendix One – Main Survey Instrument).

7.3 Analysis of Student Grades

The 'Introduction to Programming' module historically used the Java programming language to introduce students to programming up until Spring Semester 2016. From Fall Semester 2016 onwards Scratch was used during the first 2 weeks of instruction and students were introduced to basic programming constructs, such as variables, obtaining and validating input,

output, conditions, loops, event handling, modularity, and code documentation; subsequently, the students moved on to programming using Java.

Student final grades were calculated based on a coursework assessment, which accounted for 60%, and a midterm examination, which accounted for the remaining 40%. Until Fall Semester 2016, coursework historically consisted of two parts, the first part being the development of a problem solution using pseudocode and the second being the implementation of the same problem using Java. From Fall Semester 2016, the first part was modified to include the implementation of a program in Scratch. Mean student final grades through the years are shown in Figure 7.5.

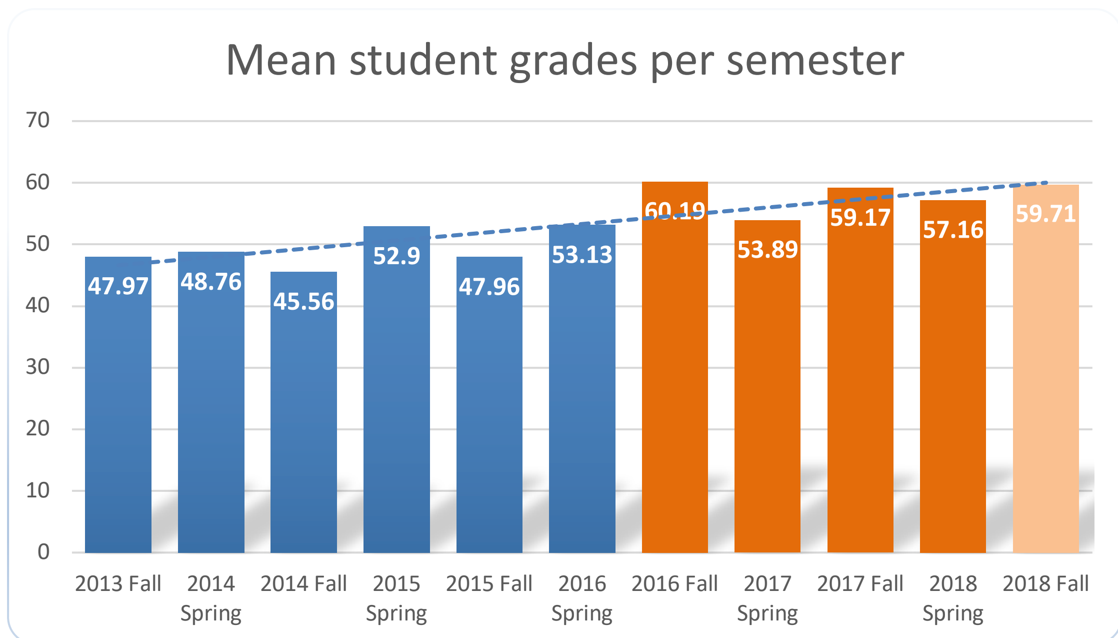


Figure 7.5: Mean student grades per semester from 2013 – 2018

Semesters appearing in blue are those that pre-dated introduction of the Scratch visual programming environment, while those appearing in orange include the usage of Scratch. Since the instruction of Scratch has not ended with the end of this study, more recent data (Fall Semester 2018) are included for this comparison. Group statistics (before and after the use of Scratch) show a mean grade difference of 9.15% (see Figure 7.6).

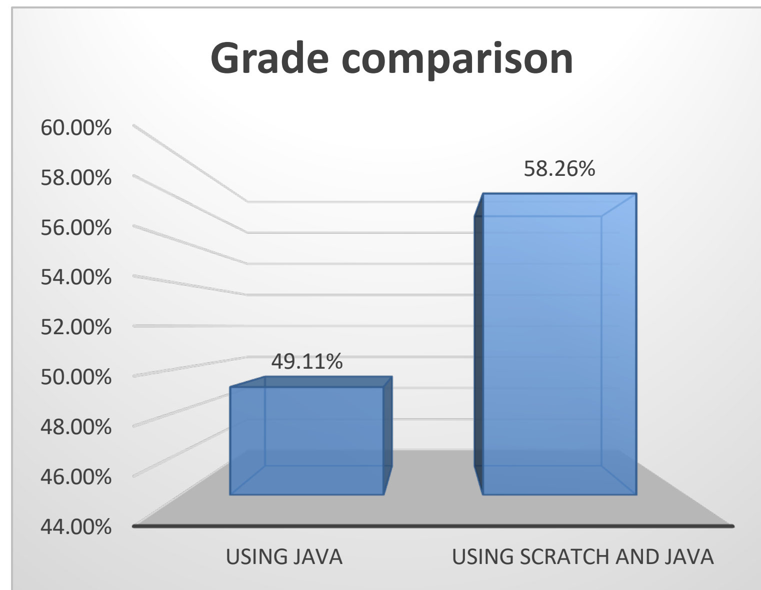


Figure 7.6: Grade comparison before and after the use of Scratch

I should stress that the student selection process has not changed since 2010 when the college XYZ was affiliated with the Open University; the module class time has been the same and so are the professors who teach the module. Other factors might have affected this shift in grades such as the students' growing familiarity with technology and motivation to learn programming. However, these factors have not been noted by any professors teaching programming modules during the past years.

A Shapiro-Wilk test of Normality ($p>0.05$), as well as a visual inspection of the histogram, box plots and QQ-plots, showed that student grades for all students in all years are approximately normally distributed. As such, an independent samples t-test was run to determine if there were differences between the grades achieved by students before and after the intervention. Homogeneity of variances was noted, as assessed by Levene's test ($p=0.476$). The grades of students who were introduced to programming using Scratch were greater ($N=110$, $M=58.26\% \pm 1.78$, $SD=18.6$) than those of students who were introduced to programming using Java ($N=141$, $M=49.11\% \pm 1.48$, $SD=17.6$), demonstrating a statistically significant difference of 9.15% (95%, CI from 4.63 to 13.48), $t(249) = 3.84$ and $p=0.000089$ ($d=0.5$). This leads us to conclude that the difference of means of students' grades is noteworthy and that they were improved substantially for the Scratch group.

Another interesting finding comes from the observation of pass/fail rates (see Figure 7.7). There was homogeneity of variances, as assessed by Levene's test (p=0.884). The overall pass/fail rate of students introduced to programming using Scratch demonstrated an improvement of 15% (69% of students passed and only 26% failed the module) over the pass/fail rate of students introduced to programming using Java (57% of students passed and 42% failed). This statistically significant improvement (95%, CI from 2.76 to 28.87), t(20)=2.528 and p=0.02, leads us to conclude that the pass/fail rates were considerably enhanced with the introduction of Scratch.

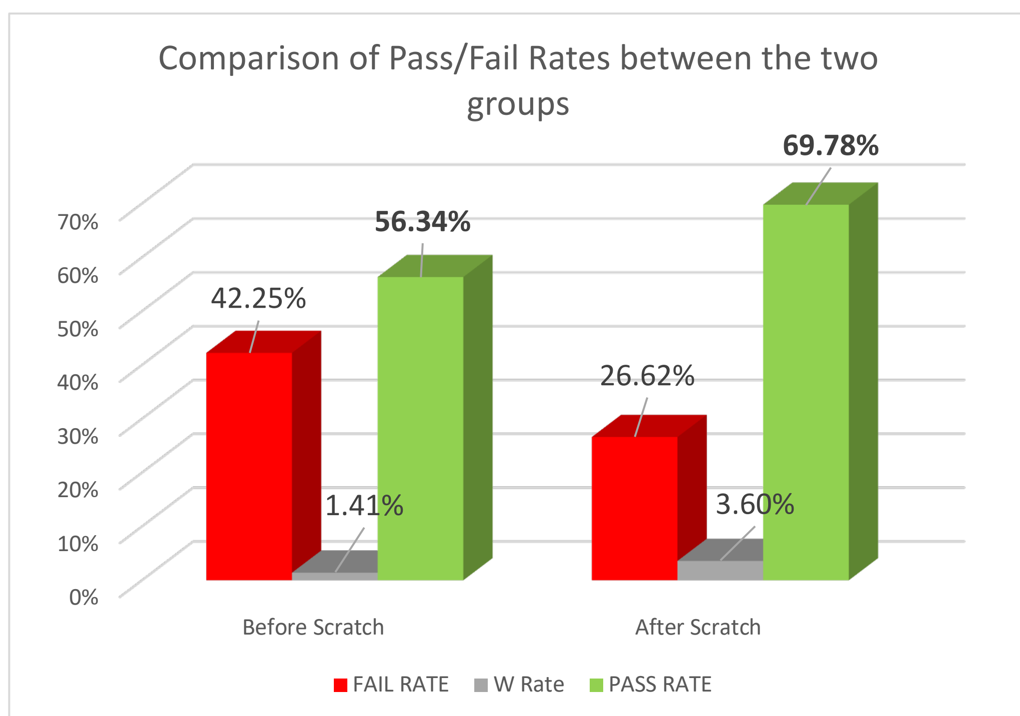


Figure 7.7: Pass/fail rates

Similar research in the area was performed by Weintrop and Wilensky (2017), who compared student performance, learning gains and enjoyment amongst high school students being introduced to block-based programming prior to text-based programming versus students being solely introduced to text-based programming. Findings from their study also showed that students in the block's pre-condition demonstrated greater learning gains and increased interest, while students with the text-based instruction perceived their experience as being more effective in improving their programming ability.

A second comparison of student performance with my data was done using the grades obtained by 92 students from the Scratch part of the coursework and the grades of the same students from the Java part of the coursework.

Both assessment rubrics evaluated the understanding and appropriate use of the same constructs, and the programming requirements focused on implementing similar functionality. Students were required to develop a hangman game both in Scratch and in Java, utilising a fixed dictionary of 10 words. The program had to randomly pick a word from the dictionary and the user had to guess the word. User input was to be validated and compared to the letters of the word picked, allowing one to evaluate the appropriate use of strings and conditionals by the programmer. Ten tries were allowed in each game, thus demonstrating the appropriate usage of repetition. Both programs had to be documented using in-line code comments. Modularity of the code was also a factor to be assessed.

A paired-samples t-test was used to determine the importance of the mean difference of 18.2% obtained. Students performed 18.2% better in the Scratch part of the coursework (M=71.86%, SD=16.7) compared to the Java part of the coursework (M=53.60%, SD=20.61), using the same marking scheme. A statistically significant mean score increase (95% CI:14.53 to 21.99, t(91)=9.72 p< 0.0005, d=1.01) is depicted in Figure 7.8.

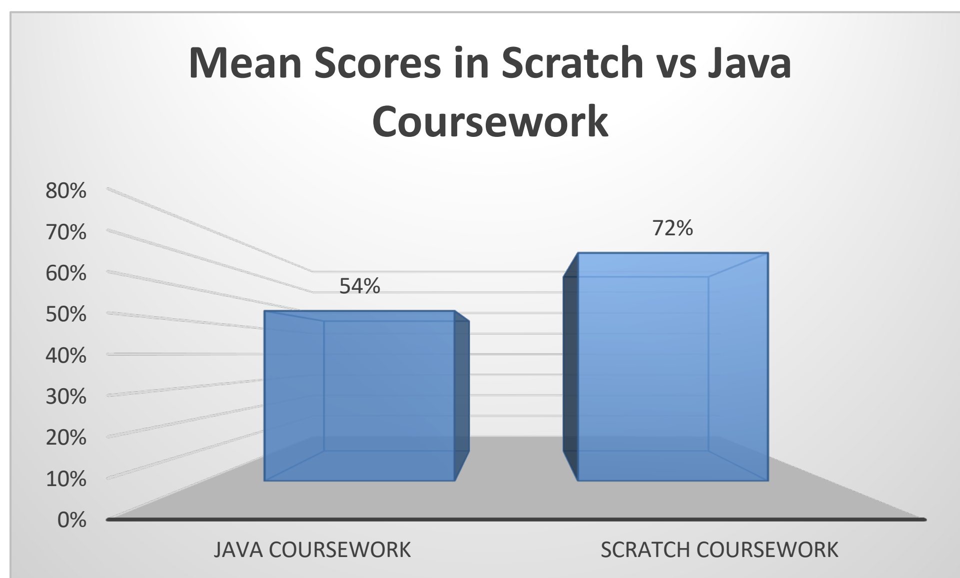


Figure 7.8: Mean coursework scores

It is worth restating, at this point, that students writing programs in Scratch do not need to be concerned about and focus on syntax errors; they need only to focus on the required functionality (see section 4.5).

Research on the comparison of student performance using different programming languages has been performed in the past. Savic *et al.* (2016) compared student performance in Modula-2 and Java but did not find any statistically significant differences while Papadakis and Orfanakis (2018) compared student performance in Alice and App Inventor and found that students performed better in App Inventor projects. Kowalczyk *et al.* (2016), on the other hand, compared the student perceptions on the readability and look and feel of both apps, but not student performance.

7.4 Results from Student Surveys

7.4.1 Student Acceptance of Scratch (TAM) Analysis

"The most challenging task is to get everything right at once: a programming language that is easy for beginners, has enough power for experts, comes with an environment which meets the user's needs, and is attractive to use..." (Green, 1990).

As described in section 6.3, the instrument for measuring overall student acceptance of Scratch is composed of 4 dimensions: perceived enjoyment; output quality; ease of use; and usefulness, as well as 2 outcome variables: intention to use (for personal reasons); and recommendation that it continues to be employed as a teaching tool in the module.

Perceived enjoyment and output quality scales consist of 3 questions each, whereas ease of use and usefulness scales consist of 5 questions each. Intention to use and recommendation to adopt are based on a single question each. Student responses are given on a Likert scale ranging from 1 (strongly negative) to 7 (strongly positive). Overall acceptance is calculated as a mean of the student answers to all questions.

The overall descriptive results of the study are presented in Table 7.2, while a breakdown per semester and per professor follows in Table 7.3. Using this

detailed information, a further investigation can be conducted, to identify whether each professor's teaching style might have influenced student perceptions about Scratch.

	Output Quality	Enjoyable	Ease of Use	Useful-ness	Use for the Module?	Intend to Use	Overall Acceptance
N	92	92	92	92	92	92	92
Missing	0	0	0	0	0	0	0
Mean	**4.36**	**4.80**	**5.27**	**4.21**	**4.46**	**3.32**	**4.40**
Median	**4.67**	**5.00**	**5.60**	**4.40**	**5.00**	**3.00**	**4.54**
Std. Deviation	1.43	1.29	1.14	1.50	1.75	1.42	0.98
Range	5.67	5.33	5.60	5.60	6.00	5.00	4.31
Minimum	1.33	1.33	1.20	1.40	1.00	1.00	1.90
Maximum	7.00	6.67	6.80	7.00	7.00	6.00	6.21

Table 7.2: Descriptive statistics of Scratch acceptance

Observing the means for each subscale, we can see that, although they find the tool very easy to use (5.27) and somewhat enjoyable (4.80), students are almost neutral in their opinion about the software output quality (which is the demonstrability of the final programs and their functionality) and its usefulness (4.21). What is interesting to explore qualitatively is why their intention to use Scratch outside the scope of the module is much lower (3.32) than their recommendation that the tool be adopted into this introductory course (4.46). In Table 7.3, descriptive statistics for each acceptance dimension per professor are presented.

Dimensions (Scales)	Professor	N	Mean	Std. Deviation	Std. Error	Min.	Max.
Output Quality	p1	24	4.29	1.52	0.31	1.33	6.33
	p2	21	4.36	1.21	0.26	1.67	6.33
	p3	28	4.37	1.28	0.24	1.67	6.33
	p4	19	4.44	1.83	0.42	1.33	7.00
	Total	**92**	**4.36**	**1.43**	**0.15**	**1.33**	**7.00**
Enjoyable	p1	24	4.54	1.39	0.28	1.33	6.33
	p2	21	5.12	1.07	0.23	2.67	6.67
	p3	28	5.02	1.09	0.21	2.67	6.67
	p4	19	4.46	1.57	0.36	1.33	6.67
	Total	**92**	**4.80**	**1.29**	**0.13**	**1.33**	**6.67**

Dimensions (Scales)	Professor	N	Mean	Std. Deviation	Std. Error	Min.	Max.
Ease of Use	p1	24	5.02	1.42	0.29	1.20	6.80
	p2	21	5.31	0.93	0.20	3.33	6.60
	p3	28	5.25	1.11	0.21	2.40	6.80
	p4	19	5.56	1.02	0.24	3.33	6.80
	Total	**92**	**5.27**	**1.14**	**0.12**	**1.20**	**6.80**
Usefulness	p1	24	4.13	1.29	0.26	1.80	6.80
	p2	21	4.71	1.28	0.28	2.80	6.80
	p3	28	4.26	1.69	0.32	1.40	7.00
	p4	19	3.69	1.62	0.37	1.40	6.40
	Total	**92**	**4.21**	**1.50**	**0.16**	**1.40**	**7.00**
For the Module?	p1	24	4.46	1.59	0.32	1.00	7.00
	p2	21	4.00	1.64	0.36	1.00	7.00
	p3	28	4.75	1.82	0.34	1.00	7.00
	p4	19	4.53	1.98	0.46	1.00	7.00
	Total	**92**	**4.46**	**1.75**	**0.18**	**1.00**	**7.00**
Intend to Use	p1	24	3.46	1.41	0.29	1.00	6.00
	p2	21	3.14	1.28	0.28	1.00	6.00
	p3	28	3.11	1.52	0.29	1.00	5.00
	p4	19	3.63	1.46	0.34	1.00	6.00
	Total	**92**	**3.32**	**1.42**	**0.15**	**1.00**	**6.00**
Overall Acceptance of Scratch	p1	24	4.32	1.07	0.22	1.90	6.21
	p2	21	4.44	0.63	0.14	3.20	5.47
	p3	28	4.46	1.01	0.19	2.64	6.10
	p4	19	4.39	1.20	0.28	2.18	6.08
	Total	**92**	**4.40**	**0.98**	**0.10**	**1.90**	**6.21**

Table 7.3: Descriptive statistics of Scratch acceptance per professor

A one-way ANOVA was conducted to determine if the overall student acceptance of Scratch, along with its sub-scales, was statistically different for groups of students which were taught by different professors. Participants were classified - for this test - into 4 groups: p1 ($n=24$); p2 ($n=21$); p3 ($n=28$); and p4 ($n=19$). There were no outliers, as assessed by visual inspection of the box-plots; data were normally distributed, as assessed by the Shapiro-Wilk test ($p>0.05$); and there was homogeneity of variances, as assessed by Levene's test ($p=0.079$) for the overall TAM, but the differences in the student perceptions about Scratch were not statistically significant, as shown in Table 7.4.

One-way ANOVA		
Between Groups (df = 3)	**F**	**Sig.**
Output Quality	0.037	0.991
Enjoyable	1.509	0.218
Ease of Use	0.820	0.486
Usefulness	1.601	0.195
For the Module?	0.742	0.530
Intend to Use	0.691	0.560
Overall Acceptance	0.105	0.957

Table 7.4: One-way ANOVA - Student acceptance of Scratch between professors

Since there were no significant differences between the groups, and each professor's unique teaching style or personal opinion about the tool did not influence the students' perceptions, it was deemed that all data could be treated as one group for the analysis to follow.

A Spearman's rank-order correlation was run to assess the relationship between the TAM subscales for Scratch and grades obtained in the Scratch assessment, as well as the final grades. The test did not show a statistically significant correlation between the variables (see Table 7.5).

Spearman's rho Correlation Coefficient / Sig (2-tailed)		
TAM Scales	**Scratch Assessment**	**Final Grade**
Overall Acceptance of Scratch	-0.027	0.026
	0.801	0.803
Output Quality	-0.201	-0.108
	0.054	0.306
Enjoyable	0.007	-0.031
	0.945	0.772
Ease of Use	0.011	-0.025
	0.919	0.816
Usefulness	-0.157	-0.126
	0.135	0.233
Appropriate for the Module?	0.006	0.059
	0.957	0.579
Intend to Use	0.019	0.079
	0.856	0.456

Table 7.5: Spearman's rho correlations between Scratch acceptance and student grades

Interpreting the results, we can conclude that, for the specific group of students, their acceptance of Scratch did not correlate to their performance in the assessments. It is interesting to note that, although students found the tool very easy to use (5.27) and somewhat enjoyable (4.80), this did not relate to their performance in the coursework.

7.4.2 Student Index of Learning Styles (ILS) Analysis

The Index of Learning Styles (Felder & Soloman, 1993) is composed of 44 questions, designed to assess the level of a student's learning preference using the 4 dimensions of the Felder-Silverman model (Felder & Silverman, 1988): active/reflective; visual/verbal; global/sequential; and sensing/intuitive. Eleven questions are associated with each dimension, with each question having only 2 possible answers (a and b). The scoring method, according to Felder counts all "a" and "b" responses and produces a dimension score of 0 to 11 for "a" and from 0 to 11 for "b". As the count of "a" answers increases, the count of "b" answers decreases and vice versa. If counting "a" scores only, a value from 10 to 11 shows a strong preference on one side of the dimension, 8 to 9 a moderate one, 4 to 7 a mild preference on either side, 2 to 3 a moderate preference on the other side, and 0 to 1 a strong preference on the other side.

For the data analysis of this study, participants were placed in one of three groups per category, distinguishing a preference, for example, between: visual, balanced or verbal; active, balanced or reflective; sequential, balanced or global; or sensing, balanced or intuitive.

In most studies considering the implications of student learning styles (Abdul-Rahman & Du Boulay, 2014), researchers examine students with moderate and strong preferences, as students with mild preferences do not demonstrate clearly-defined behaviour which could associate them with one or another side of the dimension. Therefore, in this study also, strong and moderate styles were grouped together while low scores on either side indicated a balanced preference.

Visual programming environments aim at providing learners with an environment where they can learn programming while having 'fun' and, at the

same time, create demonstrable programs (output quality). Effectiveness and usefulness towards accomplishing the learning objectives of the module is also an important factor. Considering the learning style preferences of the learners and the possible correlation between student beliefs about these software qualities can help instructors decide whether such a programming environment is generally beneficial or is more applicable to specific groups of students.

Some descriptive statistics from the 92 students who participated in the study are shown in Table 7.6.

	Strong & Moderate	Balanced	Strong & Moderate
INPUT:	**Visual**		**Verbal**
N	47	24	21
Percent	**51%**	**26%**	23%
UNDERSTANDING:	**Global**		**Sequential**
N	23	31	38
Percent	**25%**	**34%**	**41%**
PERCEPTION:	**Sensing**		**Intuitive**
N	39	34	19
Percent	**42%**	**37%**	**21%**
PROCESSING:	**Active**		**Reflective**
N	17	65	10
Percent	**19%**	**71%**	**11%**

Table 7.6: Student dominant learning styles in the 4 dimensions

Table 7.6 shows that most students in this study associate with visual (51%), sequential (42%) and sensing (41%), while the majority (71%) are balanced in the processing (active/reflective) dimension.

According to Felder's implications of learning styles preferences:

- learners with a visual learning style preference tend to prefer pictures, diagrams and flowcharts, as opposed to verbal learners, who prefer spoken or written explanations;

- learners with a sequential learning style preference tend to gain understanding in a linear, step-wise incremental manner, while global learners prefer a holistic approach;
- learners with a sensing learning style preference tend to like learning facts and procedures, are more practical compared to intuitive learners, who are conceptual and oriented towards theories;
- learners with an active learning style preference tend to learn by trying things out and working in groups, in contrast to learners with a reflective learning style preference, who prefer to think things through and work alone.

The statistics describing the learning preferences for this group of students (most majoring in Information Technology, see Figure 7.3), are similar to research findings for CS students (Zualkernan *et al.*, 2006; Chen & Lin, 2011), which show that most programming students have a strong preference in the visual dimension.

The practical orientation of the computer programming discipline, which has been discussed in the literature, also matches the dominant sensing learning preference identified in this study.

To address RQ4: “How do students’ learning styles relate to their perceived enjoyment, ease of use, usability and usefulness of visual programming environments?” a Spearman’s correlation analysis was performed between the two categories (see Table 7.7). The assumptions for a Spearman’s correlation analysis for ordinal variables, paired observations and monotonic relationship (Spearman, 1904) are met.

Spearman's correlation analysis								
	PERCEPTION		INPUT		PROCESSING		UNDERSTANDING	
	SENSING	INTUITIVE	VISUAL	VERBAL	ACTIVE	REFLECTIVE	SEQUENTIAL	GLOBAL
Output Quality	0.215*	-0.215*	0.154	-0.154	0.112	-0.112	0.049	-0.049
Enjoyable	-0.115	0.115	0.543**	-0.543**	0.186	-0.186	0.044	-0.044
Ease of Use	0.420**	-0.420**	0.279**	-0.279**	0.012	-0.012	0.030	-0.030
Useful-ness	-0.086	0.086	0.366**	-0.366**	0.011	-0.011	0.776**	-0.776**
For the Module?	0.083	-0.201	0.286**	-0.286**	0.156	-0.156	0.334**	-0.334**
Intend to Use	0.184	-0.184	0.176	-0.108	0.08	-0.08	0.019	-0.019
Overall Acceptance	0.184	-0.184	0.511**	-0.511**	0.143	-0.143	0.380**	-0.380**
N	92	92	92	92	92	92	92	92

** Correlation is significant at the 0.01 level (2-tailed) p<0.01

* Correlation is significant at the 0.05 level (2-tailed) p<0.05

Table 7.7: Correlations between student learning styles and their perceptions about Scratch

After the execution of the test, the following can be observed:

- Perceived output quality of the programs written in Scratch shows a statistically significant correlation (p=0.215) with learners with a sensing learning style preference, at the 0.05 level of significance;

- Perceived enjoyment shows a statistically significant correlation (p=0.543) with learners with a visual learning style preference, at the 0.01 level;

- Ease of use shows a statistically significant correlation with learners having a sensing (p=0.420) and visual (p=0.279) learning style preference, at the 0.01 level;

- Usefulness shows a statistically significant correlation with learners having a visual (p=0.391) and sequential (p=0.776) learning style preference, at the 0.01 level;

- Student recommendation to continue the use of Scratch for this module shows a statistically significant correlation with learners having a visual (p=0.286) and sequential (p=0.334) learning style preference, at the 0.01 level;

- It is interesting to also note that there is no significant correlation between students' learning style preferences and the intent to use Scratch outside the scope of the module.

For the overall acceptance of Scratch (which is the average of student perceptions in all TAM scales), we observe a higher correlation for learners having a visual learning style preference (p=0.511) and a lower correlation for learners having a sequential learning style preference (p=0.380), while there is no correlation in the sensing/intuitive and active/reflective dimensions.

7.4.3 Student Motivation (MSLQ)

Table 7.8 presents the means and ranges for all individual statements concerning student motivation to learn computer programming, grouped by motivational scales. A colour heat map shows the highest-rated motivational items in green, with 6.11 as the maximum mean value and lowest in red, with 4.34 the minimum mean value, in a scale from 1 - 7.

N=92		Mean	Median	Minimum	Maximum
Extrinsic (Mean = 5.88, SD = 0.93011)	Q35	5.34	5.5	3	7
	Q37	**6.10**	6	3	7
	Q40	5.96	6	3	7
	Q46	**6.10**	7	3	7
	Q53	5.91	6	1	7
Intrinsic (Mean = 5.86, SD = 0.89616)	Q39	5.96	6	3	7
	Q41	**6.11**	6	2	7
	Q43	6.00	6	3	7
	Q47	**6.02**	7	1	7
	Q39	5.21	5	2	7
Self-Efficacy (Mean = 5.538, SD = 1.19917)	Q34	5.74	6	1	7
	Q42	5.71	6	2	7
	Q44	5.27	6	1	7
	Q48	5.55	6	1	7
	Q51	5.42	6	1	7
Self-Regulation (Mean = 4.998, SD = 1.05581)	Q36	5.29	5.5	2	7
	Q38	4.34	4	1	7
	Q45	5.26	6	1	7

N=92		Mean	Median	Minimum	Maximum
	Q49	5.18	6	1	7
	Q50	4.92	5	2	7

Table 7.8: Motivational Component Mean Scores

The results show that *"better job/career prospect"* is the highest rated extrinsic motivational factor among participants while *"enjoyment of programming"* is the highest rated intrinsic one. Students overall reported a rather high level of self-efficacy but almost neutral levels of self-regulation. Given the importance of student motivation to learn and the positive linear correlation of motivation and self-efficacy with their academic performance studies (Mega, Ronconi, & De Beni, 2014; Walker, Greene, & Mansell, 2006; Zimmerman, 2008), these results could possibly explain the high student performance in the module over the previous 2 years.

A Shapiro-Wilk test of normality and a visual inspection of the histograms and Q-Q plots revealed that motivational scores in all categories (intrinsic, extrinsic, self-regulation and self-efficacy) are not normally distributed. Thus, to identify whether the professor has an effect on student motivation, a non-parametric Kruskal-Wallis test showed the distribution of all motivational scores are the same across professors (see Table 7.9).

Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of INTRINSIC is the same across categories of PROFESSOR.	Independent-Samples Kruskal-Wallis Test	.894	Retain the null hypothesis.
2	The distribution of EXTRINSIC is the same across categories of PROFESSOR.	Independent-Samples Kruskal-Wallis Test	.894	Retain the null hypothesis.
3	The distribution of SELF_REGULATION is the same across categories of PROFESSOR.	Independent-Samples Kruskal-Wallis Test	.489	Retain the null hypothesis.
4	The distribution of SELF_EFFICACY is the same across categories of PROFESSOR.	Independent-Samples Kruskal-Wallis Test	.860	Retain the null hypothesis.
5	The distribution of MOTIVATION is the same across categories of PROFESSOR.	Independent-Samples Kruskal-Wallis Test	.871	Retain the null hypothesis.

Asymptotic significances are displayed. The significance level is .05.

Table 7.9: Kruskal Wallis test for motivational scale distribution across students taught by different professors

A Spearman's 2-tailed correlation test was performed to identify possible relationships between student motivational components and their acceptance of Scratch. Results are presented in Table 7.10.

Spearman's rho Correlation Coefficients				
(N=92)	Intrinsic	Extrinsic	Self-regulation	Self-efficacy
Usefulness	-0.713**	-0.577**	-0.689**	-0.972**
Enjoyable	0.116	0.186	0.112	-0.005
Output Quality	0.025	0.09	0.127	-0.019
Ease of Use	0.065	0.051	0.041	0.025
Recommendation for the Module?	-0.17	-0.143	-0.193	-0.361**
Intend to Use	-0.238*	-0.201	-0.295**	-0.438**
*** Correlation is significant at the 0.01 level (2-tailed).*				

Table 7.10: Spearman's rho correlation between student motivation to learning programming and acceptance of Scratch

Results show:

- strong negative correlations between all components of student motivation to learn programming and the perceived usefulness of Scratch, with the strongest negative correlation between self-efficacy and perceived usefulness;
- moderate negative correlation between self-efficacy, student recommendations to use Scratch for the module and their intention to use it outside the scope of the class;
- moderate negative correlation between self-efficacy and intention to use Scratch;
- weak negative correlations between intrinsic motivation, self-regulation and their intention to use Scratch.

The characterisation of the correlations as strong, moderate, modest and low was made according to Cohen's guidelines (Cohen *et al.*, 2013).

An interesting observation is that students who believed in their abilities and had a strong intrinsic and extrinsic motivation to learn programming did not perceive Scratch as useful for the module. In a related study, Martinez *et al.*,

(2017) reported negative feedback concerning the suitability of Scratch for an introduction to programming, where 55% of university students majoring in game development thought that Scratch should be suppressed or deserve shorter instruction time.

A final Spearman's 2-tailed correlation test was performed to identify possible relationships between student motivational components and performance in both assessments as well as their final module grade. Results are presented in Table 7.11.

Spearman's rho Correlation between student motivation to learn programming and performance			
N= 92	**Scratch Assessment Grade**	**Java Assessment Grade**	**Final Grade**
Self-efficacy	.692**	.742**	.776**
Extrinsic	.542**	.628**	.641**
Self-regulation	.649**	.730**	.741**
Intrinsic	.591**	.637**	.665**
*** Correlation is significant at the 0.01 level (2-tailed).*			

Table 7.11: Spearman's rho correlation between student motivation to learning programming and performance

The test revealed significant positive correlations between student performance and motivation, with even higher correlations with the Java assessment grade. Examining the correlation coefficients, it can be observed that self-efficacy and self-regulation might have a greater impact on student performance than extrinsic motivation, but all factors significantly influence performance. This interpretation is in line with related research on motivation and academic performance (Pintrich & de Groot, 1990; Schunk, 1991; Zimmerman, Bandura, & Martinez-Pons, 1992).

7.5 Results from the Analysis of Interview Data and Class Observations – Qualitative Feedback

Qualitative feedback was collected from 12 students who volunteered to be interviewed. Seven students were interviewed by me personally and 5 students were interviewed by the professor teaching a different section of the same module who also followed the same interview protocol. In order to reduce the

possible influence (or “interview bias”) stemming from direct student-teacher relationships, my colleague and I decided to cross-interview each other’s students. The interviews were recorded, transcribed and coded using NVivo software.

In order to obtain student feedback on research question 1: “How do students perceive Scratch visual programming environment, how they perceive its enjoyability, ease of use, usability and usefulness and how they relate these qualities to their achievement of the module’s learning objectives?”, I focused the questions on the perceived advantages and disadvantages of using Scratch for the introduction to programming. Findings from the qualitative analysis were used to enhance, explain and elaborate on the results collected from the surveys.

The coding frame for the advantages theme was developed in advance, using the deductive approach, while the code frame for the disadvantages theme was created from the analysis using the inductive approach (see Appendix Three). The resulting child nodes (sub-codes) were grouped into the main codes after the analysis of the interview transcripts using the constant comparative method (Glaser & Anselm, 1967). The coding scheme, as well as the quotes in each category, were reviewed and agreed upon by both interviewers.

A summary of associated categories is presented in Figure 7.9 and Figure 7.10. The numbers which appear on each node are the frequency tallies of each concept as reported by each participant. In some cases, participants reported the same concept more than once.

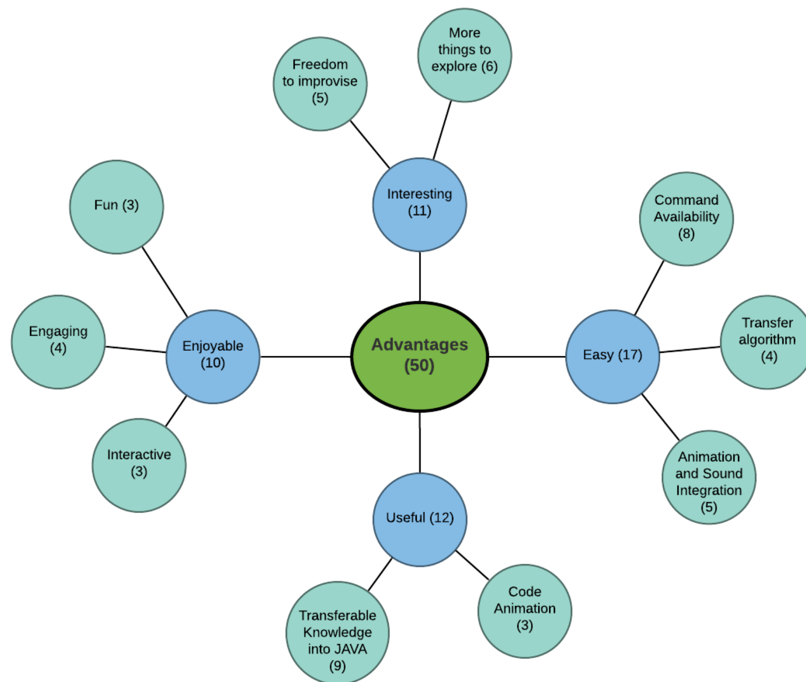


Figure 7.9: NVivo coding of Scratch advantages as perceived by 12 students

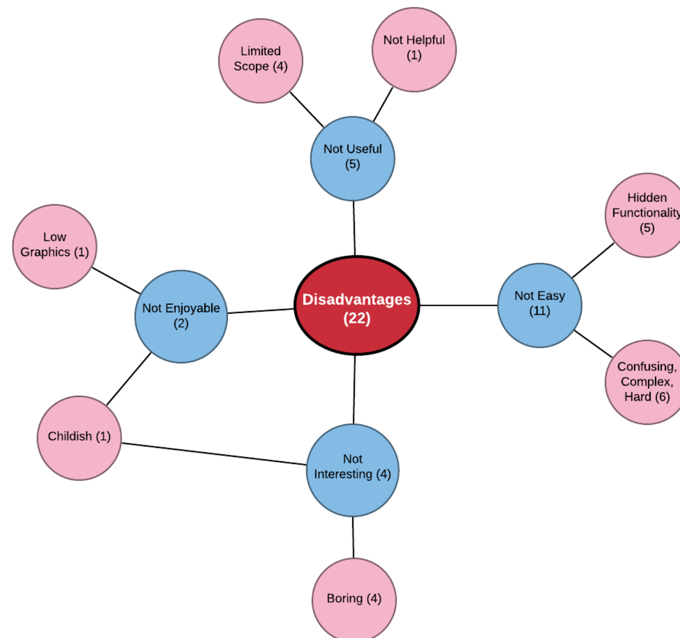


Figure 7.10: NVivo coding of Scratch disadvantages as perceived by 12 students

From analysis of the interview transcripts, it became obvious that students found more advantages than disadvantages in the use of Scratch. As such, I gained an insight as to why students found Scratch useful, enjoyable, interesting and easy.

"Freedom to improvise" and "having more things to explore" were new concepts, which have not been identified in the literature, as to why Scratch is interesting. "Interactivity", "engagement" and "fun" were identified as the reasons why students enjoyed using Scratch. "Code animation during program execution" helped students clarify repetition programming constructs and generally many respondents considered that programming knowledge gained from Scratch "transferred into Java" by clarifying concepts. "Availability of commands" and "easy integration of sound and animation" were two of the reasons why students found Scratch easy. Another perceived advantage was the easier transfer of an algorithm to a Scratch program.

Several disadvantages were mentioned regarding the use of Scratch, but most of these were reported by a single student, who clearly disliked it. The participant specifically said that: "*I found Scratch extremely confusing with low graphics and not helpful at all. It's not real programming and I am afraid that if I tell someone that in my college, we use Scratch to understand programming concepts, he/she will think that we have a very low educational level*". The student's belief that code produced in a visual programming environment is *not real code* or does not have a real-world applicability, has also been reported in the literature, along with the perception that it is limited in scope and thus less powerful (Weintrop & Wilensky, 2015). In a sense, this is not far from reality, as Scratch is an educational programming environment and should be evaluated as such.

The most commonly expressed disadvantage (reported 6 times) is that Scratch is "confusing, complex and hard", but its number of mentions was lower compared to how often Scratch was referred to as easy (17 comments) and contradicts the results obtained from the qualitative analysis concerning the ease of use (see Table 7.2).

The fact that 5 students considered that Scratch has "hidden functionality" was a surprise, since similar comments have not been found reported in the literature. Indeed, in the environment of Scratch, there are some hidden features, which could possibly confuse novice programmers. By holding down the Shift, Control or Command Key and clicking on an object or area on the screen, more options appear. For example, from using shift and clicking on the file menu in the web applications, more commands appear. The same happens on the desktop application with more commands appearing in the edit menu (see Figure 7.11).

Figure 7.11:Scratch "hidden" features

Three participants characterised Scratch as "boring", while the rest considered it "fun" and "engaging". Similar findings are reported by Ouahbi *et al.* (2015) where 15% of high school students in his study, found programming with Scratch to be boring.

It is worth noting that a student, who claimed that Scratch is interesting, also stated that he/she would be bored to use it outside of the class.

The following interview questions aimed to investigate students' motivation concerning the use of Scratch outside the scope of the module's assessments:

- Did you try to furtherly enhance (at home) Scratch projects we developed in class? Why?
- Where you motivated to use Scratch outside the scope of this module? i.e. developing your own games?

To the above questions, only one of the 12 participants answered positively, stating: "Yes, I sometimes enhanced games we did in class" and another participant said: *"I started to develop one, but I did not finish it due to time restrictions"*. Given the above feedback, it is reasonable to conclude that students' motivation to work at home on enhancing a program was not affected by their perception of Scratch as being fun, interesting and easy. Even students who viewed Scratch as useful did not demonstrate a greater motivation to develop a Scratch game outside the scope of the module. Studies on student motivation (Black & Wiliam, 2006; Stefanou *et al.*, 2018) have also pointed out the problem related to diminishing motivation when students work on a project which will not be academically assessed.

Representative student comments on why they would not use Scratch outside the scope of the lesson include:

- *"Let's say Scratch is interesting... just for the duration we got involved in class. If I had to use it for a greater amount of time, I think I would be bored"*
- *"I found Scratch fun in the class, but not fun enough to create a program, if there was no grading involved"*
- *"If I did not have anything else to do... maybe"*
- *"I do not have time for childish games, I am more interested in learning real programming"*

The final two interview questions aimed to compare student perceptions between Scratch and Java programming:

- Which of the two coursework assessments did you enjoy more developing?

- Which of the two coursework assessments (Scratch/Java) did you spend more time developing? Why?

Nine students mentioned that they enjoyed developing the Scratch game more than the Java game, and only 3 enjoyed coding in Java. This may be attributable to the level of previous programming experience these students had. It should be noted that the 3 students who preferred Java over Scratch

did in fact have prior programming experience and were the same ones that highlighted many Scratch disadvantages.

Surprisingly enough, all the participants stated that they spent more time in developing the Java game. Most participants ascribed this to difficulties in translating their ideas into programming language commands and finding "bugs". Some representative student responses were:

- *"I found it very difficult to translate my thoughts in a programming language"*
- *"Debugging was harder"*
- *"It was easier to program in Scratch… I could see clearly how the code was executing and finding logical errors was more obvious that in Java code"*
- *"I found the whole Java programming process difficult and time consuming"*
- *"I spend hours trying to figure out what I was doing wrong [in Java]"*

Referencing the qualitative feedback obtained during the interviews in an attempt to explain the higher grades obtained in Scratch coursework compared to Java coursework (see section 7.3), most interviewees found the difficulty level of performing the same tasks in Scratch to be lower than in Java, when performing identical tasks. This might be a possible explanation of why they performed significantly better.

To complement, complete and contrast student motivation findings created from the analysis of interviews and questionnaires, students' behaviour was observed while using Scratch in the classroom. This was done to mitigate the risks and limitations of addressing the concept termed "motivation". As explained by Madrid and Canado (2001), we cannot observe a person's motivation; what we can do is observe a person's behaviour. Through the observation of behaviour, we can deduce the existence of a greater or lesser degree of motivation (West & Uhlenberg, 1970). In these class observations, the tutors agreed to follow a systematic direct observation and keep notes around four behaviours of interest, which were defined a priori: emotional expressions (positive or negative); attention to the task; perseverance in completing the activity; and performance (see Appendix Three). A summary from the class observations extracted from the professors' notes, grouped

according to the four behaviours of interest, is shown in Table 7.12.Table 7.12: Summary of notes from class observations

Assignment / Difficulty	Emotional Expressions	Attention to the task	Perseverance	Performance
1 easy	mostly smiles	great attention, competition	great	excellent
2 easy	playful mood; laughs and smiles	initially bored, then focused and intrigued	*Not recorded by instructors*	excellent
3 medium	excitement,	curiosity	great	excellent, including improvements
4 medium-hard	interest	mostly focused, few bored	some	very good

Table 7.12: Summary of notes from class observations

Motivation, fun, and enthusiasm levels were reflected in the class observations. Overall, students demonstrated a positive engagement with Scratch in the classroom (see Appendix Three). This finding corroborates the questionnaire results, which show that students accepted Scratch with a mean score of 4.4/7 on the Likert scale (see section 7.4.1 Therefore, we can conclude that students were overall in favour of this pedagogical approach.

7.6 Conclusion

This chapter presented and analysed the data collected from multiple sources (survey tools, interviews and class observations) during this case study and reported on the findings.

Evidence suggests that students found Scratch to be easy, useful, enjoyable and engaging, but only within the scope and purpose of the module. On the other hand, students demonstrating strong intrinsic motivation to learn programming and high levels of self-efficacy did not perceive Scratch to be as useful as other students did. Results also indicate that a relationship exists between the acceptance of a visual programming environment and students' learning style preferences; Scratch was found more useful and enjoyable by those reporting visual and sequential learning approaches. Furthermore, overall student performance and pass-fail rates showed considerable improvement following the introduction of Scratch.

In the following chapter, the findings are discussed in relation to the research questions, the study's limitations are acknowledged and suggestions for future work are provided.

Chapter 8 Conclusions

8.1 Contribution of this Study to the Research Literature

Teaching novices computational thinking and computer programming is a challenging endeavour. This thesis, inspired by my own experience as an educator and reported challenges that computer science educators face in introductory programming courses, presents an extensive investigation into the use of visual programming environments to support the teaching and learning of introductory programming modules. The work of this thesis contributes to the enhancement of existing knowledge surrounding such usage.

More specifically, evidence from the first part of this study (pilot study), indicates that Scratch gained more acceptance in terms of student preference compared to Greenfoot, Alice and APP Inventor visual programming environments. Scratch was found to be easy, enjoyable and engaging.

In relation to the first research question*: "How do visual programming environments affect students' performance in the course (assessment and final grades)?", e*vidence demonstrates a clear effect (15% improvement) on the pass/fail rate of students. The educational effectiveness of Scratch is supported by the noticeable increase (9.15%) in mean final grades across semesters. The average final student grade from Fall 2013 until Spring 2016 (before the introduction of Scratch) was 49.11% (n=141), whereas, from Fall 2016 until Fall 2018, the average student grade increased to 58.26% (n=110). Despite having examined and eliminated some known factors which might have contributed to this improvement, such as different professors, changes to module learning objectives, different student selection processes, and variability in difficulty levels of assessments, any other factors that might have influenced this shift of grades are not apparent, and are outside the control of the study. It should also be stressed that students performed better in the Scratch part of the coursework compared to the Java part of the coursework, using the same project idea and within a consistent marking scheme. The aforementioned finding that the Scratch VPE could potentially help students perform better in introductory modules verifies findings in the literature that have been previously reported (Cooper *et al.*, 2002; Ozoran *et al.*, 2012; Topalli & Cagiltay, 2018).

In relation to the second research question: *"How do students perceive the Scratch visual programming environment? How do students perceive enjoyability, ease of use, usability and usefulness? How do students relate these qualities to their achievement of the module's learning objectives (output quality)?",* students found Scratch very easy to use and somewhat enjoyable but were almost neutral in their opinion about its usefulness and the demonstrability of the final programs. This contradicts findings from the qualitative analysis of the interviews, which indicate that students found many more advantages than disadvantages in the use of Scratch within the module, namely:

- *"freedom to improvise";*
- *"having more things to explore"*;
- *"interactivity", "engagement" and "fun";*
- *"code animation during program execution";*
- *"availability of commands";*
- *"easy integration of sound and animation";*
- *"easier transfer of an algorithm to a Scratch program*".

Interestingly, the perceptions of the specific group of students about Scratch's ease of use, usefulness and enjoyment did not correlate with those students' performance in Scratch or Java assessments. The fact that students showed no inclination to use it outside the scope of the module is arguably another important finding.

Conclusions raised from class observations showed that, as long as the assignment was relatively easy, all students demonstrated high performance and perseverance regarding the task at hand. As difficulty levels rose, those students who found Scratch to be easy, useful and enjoyable demonstrated increased engagement, while those who found Scratch to be confusing, difficult, and not particularly useful showed signs of diminishing engagement. This confirms that learners are far more likely to succeed when factors such as perceived usefulness, enjoyment, and ease of use, are in place.

In relation to research question 3: *"How do students' motivation for learning programming relate to their perceptions about visual programming*

environments?", findings indicate a negative correlation between the two. Thus, students who believed in their abilities and had a strong extrinsic and intrinsic motivation to learn programming did not perceive Scratch as being as useful for the module as less motivated students did. Educators need to address the reality that highly motivated students generally require a more academically challenging course content. The fact that Scratch was perceived by most students as being easy and fun, might not satisfy this condition. A similar conclusion was reached by Howey in his doctoral dissertation (Howey, 1999). On the other hand, this research comes to verify related findings that highly motivated students exhibit better performance.

In relation to the research question RQ4: *"How do students' learning styles relate to their perceived enjoyment, ease of use, usability and usefulness of visual programming environments?"*, evidence suggests that the Scratch visual programming environment might be more suitable for learners demonstrating a visual and sequential learning preference, since they consider Scratch more enjoyable and useful. A negative implication that might affect learning, identified in the literature, could arise from a mismatch between the teaching style and the students' learning style preference (Felder & Henriques, 1995; Schmeck, 1988; Felder & Brent, 2005; Lawrence, 2012). In this case, educators should have in mind that the use of a tool with highly visual and structured pedagogical underpinnings could possibly have a negative learning effect on students with strong verbal or global learning style preferences.

A final contribution of this thesis to the literature is the development and validation of the associated data collection instruments, which include a technology acceptance model questionnaire, used to identify user attitudes towards the use of visual programming environments, and a Motivated Strategies for Learning Questionnaire, used to measure students' motivation, orientation and use of learning strategies in learning computer programming. These instruments were based on previously established research (Davis 1985; Pintrich & de Groot, 1990; Pintrich *et al.*, 1991; Glynn *et al.*, 2009) but were adapted and modified to meet the requirements of the specific case study. These resources (included in Appendix One) will be made freely available for use by other educators and researchers alike, realising a practical research contribution beyond this thesis that is both original and substantial.

My pragmatic approach to the study examined the findings from the point of view of a practitioner aiming to support and improve the practice itself, both for my own benefit and that of other practitioners.To conclude, as an educational researcher, my aim was not to generalise the findings from this study, since the case study was conducted in a specific undergraduate module of a single college, but rather to understand student perceptions about the use of a virtual programming environment while also relating them to their learning style preferences.

The value of the work reported in this thesis is not limited to the discussed findings; it also presents a teaching methodology and a tool for obtaining student feedback. This framework might assist other educators to perform future investigations and make informed decisions with regards to incorporating a visual programming environment in their own modules.

8.2 Limitations of the Study

A participant-related limitation of the study has to do with the gender breakdown of the study. There were only 14 (15%) female participants, as opposed to 78 males (85%). This gender composition was beyond my control, since it was affected by the overall enrolment in the module and is actually quite representative of the student population across the information technology department.

Comparison of student grades (before and after the use of a VPE) was performed using data from past semesters. As a result, it cannot be ascertained to what extent the observed differences between mean scores were due to the effectiveness of Scratch, or if they simply reflect existing differences between the groups due to the differences in the annual student intake. A possible limitation of this study might be attributed to this fact. Using an experimental group (Scratch prior to Java) and a control group (only Java) would be an ideal research design, but it was not possible to obtain permission from the college to have the same module taught using different teaching methodologies.

Another potential bias might be attributed to the fact that I was an "insider researcher" – in other words, I, the researcher, was also the tutor, the class observer and the interviewer for almost half of the participant population. This has been taken into consideration and I attempted to mitigate such potential bias by involving other professors during the data collection and analysis stages. I acknowledge the significance of partiality, which might arise based on my own pre-conceived ideas about the use of visual programming environments, and this was one of the reasons behind the decision to perform mixed-methods data collection.

8.3 Recommended Areas for Future Research

A Motivated Strategies for Learning Questionnaire was used to assess overall student motivation to learn programming, but this did not provide answers as to the possible effects of Scratch in student motivation; further research would be required to explore this aspect, as motivation was only observed through student behaviour while using the tool. Observer notes were not very detailed and, in some cases, not very consistent. In order to better address the issue of motivation, more qualitative feedback is required to provide an insight into student motivation to use Scratch, which has not been established through the findings of this study.

The technology acceptance model part of the survey was not analysed as per the relationship amongst its variables - perceived enjoyment, ease of use, usefulness, output quality (attitude towards using), and intention to use - because it was not within the scope of the current research. In the future, a regression analysis on the data could provide a goodness-of-fit test and produce the path coefficients of the model. The model could subsequently be tested to verify whether the intention to use a visual programming environment can be reliably predicted from the rest of the variables.

The quantitative data collected from this study are rich and multi-faceted. They can be analysed further to answer future research questions. A recommendation for future research would be to explore the relationship (if any) between students' levels of prior programming experience and their acceptance of Scratch.

It would also be interesting to explore qualitatively the reasons why student intention to use Scratch outside the scope of the module is much lower than their recommendation that the tool be permanently incorporated into this introductory course.

8.4 A Final Reflection

The process of realising this study has helped me improve delivery of the introduction to programming module at my college, by making it more approachable and engaging for the students. It has provided me with greater insight into student perceptions about visual programming environments and perceived advantages/disadvantages from the student point of view, informed me about students' overall motivation to learn programming, and assisted me in choosing appropriate tools that satisfy student needs and motivate them to practise. These findings could serve as a reference for educators to better address student needs in their pursuit to teach programming to novices. The instruments created could be used as measurement tools for gauging students' acceptance of VPEs and their motivation to learn programming, as well as a starting point for future research.

References

Abdul-Rahman, S.-S., & Boulay, Du, B. (2014). Learning programming via worked-examples: Relation of learning styles to cognitive load. *Computers in Human Behavior*, *30(1)*, 286–298.

ACM Computing Curricula Task Force (Ed.). (2013). Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. New York, NY: ACM Press.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *10*(3), 483–495.

Al-Linjawi, A., Al-Nuaim, H., & Al-Nuaim, H. (2010). Using Alice to Teach Novice Programmers OOP Concepts. *Journal of King Abdulaziz University-Science*, *22*(1), 59–68.

Alesandrini, K., & Larson, L. (2002). Teachers bridge to constructivism. *The Clearing House: A Journal of Educational Strategies, Issues and Ideas*, *75*(3), 118–121.

Allert, J.D. (2004). Learning style and factors contributing to success in an introductory computer science course. In Looi, C-K., Sutinen, E., Sampson, D.G., Aedo, I., Uden, L. & Kähkonen, E. (Eds.) *Proceedings of the IEEE International Conference on Advanced Learning Technologies - ICALT 2004* (pp. 385-389). Los Alamitos, CA: IEEE.

Anderson, J. R., & Corbett, A. T. (1995). Knowledge decomposition and subgoal reification in the ACT programming tutor. In Greer, J. E. (Ed.) *Proceedings of AI-ED '95, 7th World Conference on Artificial Intelligence in Education* (pp. 1-7). Washington, DC: AACE.

Ausubel, D. P. (1963). The psychology of meaningful verbal learning. Oxford: Grune & Stratton.

Avison, D., Lau, F., Myers, M., & Nielsen, P. A. (1999). Action research. *Communications of the ACM*, *42*(1), 94–97.

Ball, S. (Ed.). (1977). Motivation in education. New York, NY: Academic Press.

Bandura, A. (2001). Social Cognitive Theory: An Agentic Perspective. *Annual Review of Psychology*, *52*(1), 1–26.

Bandura, A., & Wessels, S. (1994). Self-efficacy. *Encyclopedia of Human Behavior*, *4*, 71–81.

Bassey, M. (1999). Case Study Research in Educational Settings. Maidenhead: Open University Press.

Begel, A. (1996). LogoBlocks: A graphical programming language for interacting with the world. Unpublished Advanced Undergraduate Project, MIT Media Lab, Boston, MA.

Begosso, L. C., & Begosso, L. R. (2012). An approach for teaching algorithms and computer programming using Greenfoot and Python. In *Proceedings of 2012 Frontiers in Education Conference (FIE)* (pp. 1–6). Piscataway, NJ: IEEE.

Ben-Ari, M. (1998). Constructivism in computer science education. *ACM SIGCSE Bulletin*, *30*(1), 257–261.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39*(2), 32–36.

Bergin, J., Jiménez-Peris, R., Brodie, K., Patiño-Martínez, M., McNally, M., Naps, T., *et al.* (1996). An overview of visualization: its use and design. *ACM SIGCSE Bulletin*, *28*(SI), 192–200.

Bergin, S., & Reilly, R. (2005). The influence of motivation and comfort-level on learning to program, In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds.) *Proceedings of the PPIG (Vol. 17)* (pp. 293-304). Brighton, UK: University of Sussex.

Biggs, J. (1987). Student Approaches to Learning and Studying. Hawthorn,VIC: Australian Council for Educational Research.

Biggs, J., Kember, D., & Leung, D. Y. (2001). The revised two-factor Study Process Questionnaire: R-SPQ-2F. *The British Journal of Educational Psychology*, *71*(Pt 1), 133–149.

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, P., & Wiliam, D. (2006). Assessment and classroom learning. *Assessment in Education: Principles, Policy & Practice*, *5*(1), 7–74.

Bloom, M. V., & Hanych, D. A. (2002). Skeptics and true believers hash it out. *Community College Week*, *14*(15), 17.

Bonar, J., & Soloway, E. (1985). Pre-programming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Interaction*, *1*(2), 133–161.

Booch, G. (1989). What is and what isn't object-oriented design. *American Programmer*, *2*(7-8), 14–21.

Borkowski, J. G., Carr, M., Rellinger, E., & Pressley, M. (1990). Self-regulated cognition: Interdependence of metacognition, attributions, and self-esteem. In B. F. Jones & L. Idol (Eds.) *Dimensions of thinking and cognitive instruction* (pp. 53-92). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Bosse, Y., & Gerosa, M. A. (2017). Difficulties of programming learning from the point of view of students and instructors. *IEEE Latin America Transactions*, *15*(11), 2191–2199.

Bowden, J., & Marton, F. (2003). The university of learning: Beyond quality and competence. London: Taylor & Francis.

Boyle, T., Bradley, C., Chalk, P., Jones, R., & Pickard, P. (2003). Using Blended Learning to Improve Student Success Rates in Learning to Program. *Journal of Educational Media*, *28*(2-3), 165–178.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodley, I. (2004). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education, 3(1), 143-160*.

Bruner, J. S. (1964). The course of cognitive growth. *American Psychologist*, *19*(1), 1–15.

Bureau of Labor Statistics. (2019). Software Developers: Occupational Outlook Handbook: U.S. Retrieved June 22, 2019, from https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

Burnett, M. M. (1999). Visual Programming. In J.G. Webster (Ed.) *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1–13. Hoboken, NJ: Wiley.

Burnett, M. M., & Baker, M. J. (1993). A classification system for visual programming languages. *Technical Report (pp. 1-20).* Corvallis, OR: Oregon State University.

Calder, N. (2010). Using Scratch: An Integrated Problem-solving Approach to Mathematical Thinking*, Australian Primary Mathematics Classroom, 15*(4), 9-14.

Cañas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human-Computer Studies*, *40*(5), 795–811.

Carpenter, P. A., Just, M. A., & Shell, P. (1990). What one intelligence test measures: A theoretical account of the processing in the Raven Progressive Matrices Test. *Psychological Review*, *97*(3), 404–431.

Carr, W., & Kemmis, S. (1986). Becoming critical: Knowledge, education and action research. *The Journal of Educational Thought (JET), 23*(3), 209-216.

Carroll, J. M., & Thomas, J. C. (1988). Fun. *ACM SIGCHI Bulletin*, *19*(3), 21–24.

Chalk, P., Boyle, T., & Fisher, K. (2003). Improving pass rates in introductory programming. In O'Reilly, U (Ed.) *Proceedings of the 4th Annual Conference of the LTSN Centre for the Information and Computer Sciences (pp.* 6-10). Galway, Ireland: LTSN-ICS.

Chamillard, A. T., & Karolick, D. (1999). Using learning style data in an introductory computer science course. *ACM SIGCSE Bulletin*, *31*(1), 291–295.

Chang, M. K., & Cheung, W. (2001). Determinants of the intention to use Internet/WWW at work: a confirmatory study. *Information & Management*, *39*(1), 1–14.

Chen, Z., & Marx, D. (2005). Experiences with Eclipse IDE in programming courses. *Journal of Computing Sciences in Colleges*, *1*(2), 1–9.

Chen, C. L., & Lin, J. M. C. (2011). Learning styles and student performance in java programming courses. In Gersting, J., Walker, H., Grissom, S. (Eds.) *Proceedings of the 33rd International Conference on Frontiers in Education Computer Science and Computer Engineering (SIGCSE '02). New York, NY: ACM. 33-37.*

Chiang, F.-K., & Qin, L. (2018). A Pilot study to assess the impacts of game-based construction learning, using scratch, on students' multi-step equation-solving performance. *Interactive Learning Environments*, *15*(4), 1–12.

Čisar, S., Radosav, D., Pinter, R., & Čisar, P. (2011). Effectiveness of program visualization in learning java: a case study with Jeliot 3. *International Journal of Computers Communications & Control*, *6*(4), 668–14.

Cliburn, D. C. (2008). Student opinions of Alice in CS1. In *IEEE Frontiers in Education Conference* (pp. T3B-7-T3B-11), New York, NY: IEEE.

Code, J. R., MacAllister, K., Gress, C. L. Z., & Nesbit, J. C. (2006). Self-regulated learning, motivation and goal theory: implications for instructional design and e-learning. In *Proceedings of the Sixth IEEE International Conference on Advanced Learning Technologies - ICALT '06 (pp. 872-874).* Washington, DC: IEEE Computer Society.

Coffield, F., & Learning and Skills Research Centre. (2004). Learning styles and pedagogy in post-16 learning: A systematic and critical review. London: Learning and Skills Research Centre.

Cohen, L., Manion, L., & Morrision, K. (2013). Research methods in education. London: Routledge.

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. In John G. Meinke (Ed.) *The journal of computing in small colleges (CCSC '00)*, 15(5),107-116.

Cooper, S., Dann, W., & Pausch, R. (2002). Teaching objects-first in introductory computer science, In Grissom S., Knox D., Joyce D., Dann W. (Eds.) *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education - SIGCSE '03* (pp. 191-195). New York, NY: ACM.

Cooper, S., Moskal, B., & Lurie, D. (2004). Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education - SIGCSE '04 (pp. 75-79)*. New York, NY: ACM.

Creswell, J. W. (2014). Research Design: Qualitative, Quantitative, and Mixed Methods Approaches (4 ed). Thousand Oaks, CA: Sage Publications, Inc.

Creswell, J. W., & Clark, V. L. P. (2011). Designing and conducting mixed methods research. Thousand Oaks, CA: Sage Publications, Inc.

Cronbach, L. J. (1951). Coefficient alpha and the internal structure of tests. *Psychometrika*, *16*(3), 297–334.

Cronbach, L. J. (1957). The two disciplines of scientific psychology. *American Psychologist*, *12*(11), 1–14.

Cronbach, L. J., & Meehl, P. E. (1955). Construct validity in psychological tests. *Psychological Bulletin*, *52*(4).

Cronbach, L. J., & Murphy, G. (1970). Essentials of psychological testing. New York, NY: Harper & Row.

Culwin, F. (1999). Object imperatives! *ACM SIGCSE Bulletin*, 31(1) 31–36.

Curry, L. (1983). An organization of learning styles theory and constructs. Halifax, NS: Dalhousie University.

Curry, L. (1987). Integrating concepts of cognitive or learning style: a review with attention to psychometric standards. New York, NY: Learning Styles Network.

Curry, L. (1990). A critique of the research on learning styles. *Educational Leadership*, *48*(2), 50.

Da Silva C., Marcelino, M. J., & Mendes, A. J. (2007). The Impact of Learning Styles in Introductory Programming Learning. In *International Conference on Engineering Education*, Coimbra, Portugal. *European Journal of Engineering Education*. 32(1), 3-7.

Dale, N. B. (2006). Most difficult topics in CS1. *ACM SIGCSE Bulletin*, *38*(2), 49–5.

Dann, W., Cooper, S., & Pausch, R. (2001). Using visualization to teach novices recursion. *ACM SIGCSE Bulletin*, *33*(3), 109–112.

Dann, W., Cosgrove, D., & Slater, D. (2012). Mediated transfer: Alice 3 to Java. *In Proceedings of the 43rd ACM technical symposium on Computer Science Education - SIGCSE '12* (pp. 141-146). New York, NY: ACM.

Davies, S., Polack-Wahl, J. A., & Anewalt, K. (2011). A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11 (pp. 625-630)*. New York, NY: Association for Computing Machinery.

Davis, F. D. (1985). A Technology Acceptance Model for Empirically Testing New End-User Information Systems (Doctoral dissertation). Massachusetts Institute of Technology, Boston, MA.

Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, *13*(3), 319.

Davis, F. D., Bagozzi, R. P., & Warshaw, P. R. (1989). User Acceptance of Computer Technology: A Comparison of Two Theoretical Models. *Management Science*, *35*(8), 982–1003.

Davis, F. D., Bagozzi, R. P., & Warshaw, P. R. (1992). Extrinsic and intrinsic motivation to use computers in the workplace. *Journal of Applied Social Psychology*, *22*(14), 1111–1132.
Davison, R. M., Martinsons, M. G., & Kock, N. (2004). Principles of canonical action research. *Information Systems Journal*, *14*(1), 65–86.
DeCharms, R. (1968). Personal causation; the internal affective determinants of behavior. New York, NY: Academic Press.
Deci, E. L. (1975). Intrinsic motivation. New York, NY: Plenum Publishing Company.
Deci, E. L. (1976). The hidden costs of rewards. *Organizational Dynamics*, *4*(3), 61–72.
Deci, E. L. (1978). Applications of research on the effects of rewards. *The Hidden Costs of Reward: New Perspectives on the Psychology of Human Motivation*, 193–203.
Decker, A. (2003). A tale of two paradigms. *Journal of Computing Sciences in Colleges*, 19(2), 238–246.
Decker, A., & Simkins, D. (2016). Uncovering difficulties in learning for the intermediate programmer. In *Proceedings of the IEEE Frontiers in Education Conference – FIE* (pp. 1-8). Pennsylvania, PA: IEEE.
Decker, A., & Trees, F. P. (2011). Greenfoot: Introducing Java with Games and Simulations: pre-conference workshop/tutorial presentation. *Journal of Computing Sciences in Colleges archive*. 26(6), 7-9.
Dewey, J. (1938). Experience and Education. *Educational Forum*, 50(3), 241-252.
Dijkstra, E. W. (1970). Notes on Structured Programming. Eindhoven, Netherlands: Technological University of Eindhoven.
Dillon, E. C. (2012). Measuring the effects of low assistive vs. Moderately assistive environments on novice programmers (Doctoral dissertation). The University of Alabama, Tuscaloosa, AL.
Dillon, E., Anderson, M., & Brown, M. (2012). Comparing feature assistance between programming environments and their “effect” on novice programmers. *Journal of Computing Sciences in Colleges*, 27(5), 69-77.
Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73.
Du Boulay, B., & O’Shea, T. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, *14*, 237–249.
Duffy, T. M., Jonassen, D. H., & Lowyck, J. (1993). Designing environments for constructive learning. Berlin, Germany: Springer-Verlag.
Duncan, T., & Mckeachie, W. J. (1991). A manual for the use of the motivated strategies for learning questionnaire (MSLQ). Ann Arbor, MI: National Center for Research to Improve Postsecondary Teaching and Learning.
Dunn, R. S., Dunn, K. J., & Price, G. E. (1979). Identifying individual learning styles. In *Student learning styles: Diagnosing and prescribing programs.* Reston, VA: National Association of Secondary School Principles, 39-54.
Dunn, R., Honigsfeld, A., Doolan, L. S., Bostrom, L., Russo, K., Schiering, M. S., *et al.* (2009). Impact of learning-style instructional strategies on students' achievement and attitudes: perceptions of educators in diverse institutions. *The Clearing House: a Journal of Educational Strategies, Issues and Ideas*, *82*(3), 135–140.
Dweck, C. S. (1999). Self-theories: Their role in motivation, personality, and development. New York, NY: Psychology Press.

Eckerdal, A. (2006). *Novice students' learning of object-oriented programming*. (Doctoral dissertation). Uppsala University, Uppsala, Sweden.
Eckerdal, A., Thuné, M., & Berglund, A. (2005). What does it take to learn 'programming thinking'? In Proceedings of the *first international workshop on Computing education research -ICER '05 (pp.* 135-142). New York, NY: ACM.
Eisenhardt, K. M. (1989). Building theories from case study research. *Academy of Management Review*, *14*(4), 532–550.
Eisenstadt, M., (1992). In Eisenstadt, M., Keane, M. T., & Rajan, T. (Eds.) *Novice programming environments. Explorations in Human-Computer Interaction and Artificial Intelligence.* London: Routledge.
Elliot, A. J., & Harackiewicz, J. M. (1996). Approach and avoidance achievement goals and intrinsic motivation: A mediational analysis. *Journal of Personality and Social Psychology*, *70*(3), 461–475.
Entwistle, N. (2005). Contrasting Perspectives on Learning. In N. Entwistle, F. Marton, & H. Dai (Eds.) *The Experience of Learning: Implications for teaching and studying in higher education (pp.* 1–11). Edinburgh: University of Edinburgh.
Entwistle, N. J. (1991). Approaches to learning and perceptions of the learning environment. *Higher Education*, *22*(3), 201–204.
Entwistle, N., & Ramsden, P. (1983). Understanding student learning. New York, NY: Croom Helm Ltd.
Entwistle, N., & Tait, H. (1990). Approaches to learning, evaluations of teaching, and preferences for contrasting academic environments. *Higher Education, 19*(2), 169–194.
Erol, O., & Kurt, A. A. (2017). The effects of teaching programming with Scratch on pre-service information technology teachers' motivation and achievement. *Computers in Human Behavior*, *77(1)*, 11–18.
Ertmer, P. A., & Newby, T. J. (2013). Behaviorism, Cognitivism, Constructivism: Comparing critical features from an instructional design perspective. *Performance Improvement Quarterly*, *26*(2), 43–71.
Fay, A. I., & Mayer, R. E. (1988). Learning Logo: A cognitive analysis. In R. E. Mayer (Ed.), *Teaching and Learning Computer Programming Multiple Research Perspectives* (pp.55-74). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.
Felder, R. M. (2005). Applications, reliability and validity of the index of learning styles. *International Journal of Engineering Education*, *21*(1), 103–112.
Felder, R. M. (2010). A*re learning styles invalid? (hint: no!)*. *On-course Newsletter*, 57-72.
Felder, R. M., & Brent, R. (2005). Understanding student differences. *Journal of Engineering Education*, *94*(1), 57–72.
Felder, R. M., & Henriques, E. R. (1995). Learning and teaching styles in foreign and second language education, *Foreign Language Annals, 28*(1), 21-31.
Felder, R. M., & Silverman, L. K. (1988). Learning and Teaching Styles in Engineering Education. *International Journal of Engineering Education*, *78*(7), 674–681.
Felder, R. M., & Soloman, B. A. (1993). *Index of Learning Styles Questionnaire (ILS)*. Retrieved from: https://www.webtools.ncsu.edu/learningstyles/
Fields, D. A., Giang, M., & Kafai, Y. B. (2013). Understanding collaborative practices in the Scratch online community: Patterns of participation among youth designers. In N. Rummel, M. Kapur, M. Nathan and S. Puntambekar (Eds.) *Proceedings of the Computer Supported Collaborative Learning – CSCL* 2013 (pp. 200-207). Madison, WI: International Society of the Learning Sciences, Inc.

Fincher, S., Utting, I. (2010). Machines for Thinking. *ACM Transactions on Computing Education (TOCE), 10*(4), 13.
Fishbein, M., & Ajzen, I. (1975). Measurement techniques. Belief, attitude, intention, and behavior an introduction to theory and research*. Philosophy and Rhetoric, 10*(2), 1–54.
Fornell, C., & Larcker, D. F. (1981). Evaluating structural equation models with unobservable variables and measurement error. *Journal of Marketing Research*, *18*(1), 39.
Freund, S. N., & Roberts, E. S. (1996). Thetis: an ANSI C Programming Environment Designed for Introductory Use. In Karl J. Klee (Ed.) *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education -SIGCSE '96* (pp. 300-304). New York, NY: ACM.
Gagné, E. D., Yekovich, C. W., & Yekovich, F. R. (1993). The cognitive psychology of school learning. New York, NY: Harper Collins College Publishers.
Gallant, R. J., & Mahmoud, Q. H. (2008). Using Greenfoot and a moon scenario to teach Java programming in CS1 In Proceedings of the *46th Annual Southeast Regional Conference on XX - ACM-SE 46* (pp.118-121). New York, NY: ACM.
Garris, R., Ahlers, R., & Driskell, J. E. (2016). Games, motivation, and learning: a research and practice model. *Simulation & Gaming*, *33*(4), 441–467.
Gentner, D., & Stevens, A. L. (2002). Mental Models. *International Encyclopedia of the Social and Behavioral Sciences* (pp. 9683–9687). Amsterdam: Psychology Press.
Gershon, N., Eick, S. G. & Card S. (1998). Information Visualization*. Interactions, 5*(2), 5-15.
Gibbs, G., & Awards, C. F. N. A. (1992). Improving the quality of student learning. (Doctoral dissertation), University of Glamorgan, Wales.
Giraffa, L., Moraes, M. C., & Uden, L. (2014). Teaching object-oriented programming in first-year undergraduate courses supported by virtual classrooms. In Uden L., Yu-Hui T., Hsin-Chang Y., I-Hsien T. (Eds) *The 2nd International Workshop on Learning Technology for Education in Cloud* (pp. 15-26). Dordrecht, The Netherlands: Springer.
Given, L. M. (Ed.). (2008). The SAGE encyclopedia of qualitative research methods. Thousand Oaks, CA: Sage Publications, Inc.
Glaser, B. G., & Anselm L, S. (1967). The discovery of grounded theory: Strategies for qualitative research. Chicago, IL: Aldine Publishing.
Glynn, S. M. (2011). Science motivation questionnaire II (SMQ-II): Components. Retrieved from: https://coe.uga.edu/assets/downloads/mse/smqii-components.pdf.
Glynn, S. M., Brickman, P., Armstrong, N., & Taasoobshirazi, G. (2011). Science motivation questionnaire II: Validation with science majors and nonscience majors. *Journal of Research in Science Teaching*, *48*(10), 1159–1176.
Glynn, S. M., Taasoobshirazi, G., & Brickman, P. (2009). Science Motivation Questionnaire: Construct validation with nonscience majors. *Journal of Research in Science Teaching*, *46*(2), 127–146.
Gomes, A., & Mendes, A. (2008). A study on student's characteristics and programming learning. In J. Luca & E. R. Weippl (Eds.) *World Conference on Educational Media and Technology* (pp. 2895–2904). Vienna, Austria: Association for the Advancement of Computing in Education (AACE).
Goodman, N. D. (1979). Mathematics as an objective science. *The American Mathematical Monthly*, *86*(7), 540.

Gottfried, A. E. (1985). Academic intrinsic motivation in elementary and junior high school students. *Journal of Educational Psychology*, *77*(6), 631–645.
Grasha A. (2002). Teaching with style: A practical guide to enhancing learning by understanding teaching and learning styles. Pittsburg, PA: Alliance Publishers.
Green, T. R. G. (1990). The nature of programming. In Hoc J. M, Green T., Samurçay R., Gilmore D. (Eds). *Psychology of Programming* (pp. 23–44), London: Academic Press.
Gries, P., Mnih, V., Taylor, J., Wilson, G., & Zamparo, L. (2005). Memview: A pedagogically-motivated visual debugger (Vol. 2005). In *Proceedings of the 35th Annual Frontiers in Education* (pp. S1J-11). Indianapolis, IN: IEEE.
Grissom, S., McNally, M. F., & Naps, T. (2003). Algorithm visualization in CS education: comparing levels of student engagement. In *Proceedings of the 2003 ACM symposium on Software Visualization* (pp. 87–94). New York, NY: ACM.
Gronlund, N. E. (1971). Measurement and evaluation in teaching. *Journal of Teacher Education*, *23*(1), 96–97.
Guzdial, M. (2004). Programming environments for novices. In S. Fincher, & M. Petre (Eds.) *Computer science education research* (pp. 127-154). Lisse: Taylor & Francis.
Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. *ACM SIGCSE Bulletin*, *31*(3), 171–174.
Hadjerrouit, S. (2007). A blended learning model in Java programming: A design-based research approach. I*n Proceedings of the 2007 Computer Science and IT Education Conference* (pp. 283-308). Arlington: Information Sciences Institute.
Hair, J. F., Hult, G. T. M., Ringle, C. M., & Sarstedt, M. (2016). A Primer on partial least squares structural equation modelling (PLS-SEM). Los Angeles, CA: Sage Publications.
Harter, S. (1981). A new self-report scale of intrinsic versus extrinsic orientation in the classroom: Motivational and informational components. *Developmental Psychology*, *17*(3), 300.
Hartley, J. (1998). *Learning and studying: A research perspective*. Routledge, London, 1998.
Henriksen, P., & Kölling, M. (2004). Greenfoot: combining object visualisation with interaction. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '04* (pp. 73–82), New York, NY: ACM Press.
Hertz, M., & Jump, M. (2013). Trace-based teaching in early programming courses. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE 2013* (pp. 561-566), New York, NY: ACM Press.
Hijon-Neira, R., Velazquez-Iturbide, A., Pizarro-Romero, C., & Carrico, L. (2013). Improving students learning programming skills with ProGames - Programming through games system. In Gary M., Gitte L., Janet W., Marco W. (Eds.) *Proceedings of Human-Computer Interaction – INTERACT 2013* (pp. 579-586). Berlin, Germany: Springer Berlin Heidelberg.
Hinkin, T. R. (1998). A brief tutorial on the development of measures for use in survey questionnaires. *Organizational Research Methods, 2*(1), 104-121.
Hogan, J. P. (1998). Mind matters: exploring the world of artificial intelligence. New York, NY: Ballantine Publishing Group.
Holt, R. W., & Schultz, A. C. (1987). Mental representation of programs for student and professional programmers. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.),

Empirical Studies of Programmers (pp. 100–113). Norwood: Ablex Publishing Corp.
Howey, S. C. (1999). The relationship between motivation and academic success of community college freshmen orientation students. (Doctoral Dissertation). Kansas State University, Kansas City, KS.
Hu, C. (2004). Rethinking of teaching objects-first. *Education and Information Technologies*, *9*(3), 209–218.
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.
Igbaria, M., Iivary, J., & Maragahh, H. (1995). Why do individuals use computer-technology - A finnish case-study. *Information & Management*, *29*(5), 227–238.
Ismail, M. N., Ngah, N. A., & Umar, I. N. (2010). Instructional strategy in the teaching of computer programming: A need assessment analyses. *Turkish Online Journal of Educational Technology*, *9*(2), 125–131.
Jenkins, T. (2001). The motivation of students of programming. *ACM SIGCSE Bulletin*, *33*(1) 53–56.
Jonassen, D. H. (1991). Evaluating constructivistic learning. *Educational Technology*, *31*(9), 28–33.
Jones, M. W. (2010). An extended case study on the introductory teaching of programming (Doctoral Dissertation). University of Southampton, Southampton.
Joppe, M. (2000). The research process. Retrieved from: https://www.uoguelph.ca/hftm/research-process.
Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education -SIGCSE '10* (pp. 107-111). New York, NY: ACM Press.
Kay, A. (2005). Squeak Etoys authoring & media. Retrieved from: http://www.squeakland.org/content/articles/attach/etoys_n_authoring.pdf
Keefe, J. W., & Languis, M. L. (1983). Operational definitions. Paper presented to the *NASSP Learning Styles Task Force.* Reston, VA.
Keefe, James W. (1985) Assessment of learning style variables: the NASSP Task Force model. *Theory into practice, 24*(2), 138–144.
Kehoe, C., Stasko, J., & Taylor, A. (2001). Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies*, *54*(2), 265–284.
Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, *37*(2), 83–137.
Keller, J. M. (1987). Development and use of the ARCS model of instructional design. *Journal of Instructional Development*, *10*(3), 2–10.
Kemmis, S., McTaggart, R., & Nixon, R. (2013). Introducing Critical Participatory Action Research. In t*he Action Research Planner* (pp. 1–31). Singapore: Springer Singapore.
Kessler, C., & Anderson, J. (1986). Learning flow of control: recursive and iterative procedures. *Human-Computer Interaction*, *2*(2), 135–166.
Kolb, D. A. (1984). Experiential learning: Experience as the source of learning and development. Englewood Cliffs, CA: Prentice-Hall.

Kolb, D. A., & Kolb, A. Y. (2014). The Kolb learning style inventory 4.0: Guide to theory, psychometrics, research & applications. Experience based learning systems. Retrieved from: https://learningfromexperience.com/downloads/research-library/the-kolb-learning-style-inventory-4-0.pdf.

Kölling, M. (1999). The problem of teaching object-oriented programming. *Journal of Object-Oriented Programming*, *11*(8), 8–15.

Kotsovoulou, M., & Stefanou, V. (2016). Student perceptions on the effectiveness of collaborative problem-based learning using online pair programming tools. In Strouhal J. (Ed.) *Proceedings of the 12th International Conference on Educational Technologies* (pp. 32–39). Barcelona: WSEAS Press.

Kowalczyk, R., Turczy ski, Ł., & yła, K. (2016). Comparison of app inventor 2 and java in creating personal applications for android on example of a notepad. *Advances in Science and Technology Research Journal*, *10*(31), 247–254.

Kozhevnikov, M. (2007). Cognitive styles in the context of modern psychology: Toward an integrated framework of cognitive style. *Psychological Bulletin*, *133*(3), 464–481.

Krauss, S. E. (2005). Research Paradigms and Meaning Making: A Primer. *Qualitative Report*, *10(*4), 758-770.

Kress, G., & van Leeuwen, T. (2001). Multimodal Discourse: The modes and media of contemporary communication. *Journal of Communication Inquiry*, *26*(3), 338–339.

Kuri, N. P., & Truzzi, O. M. S. (2002). Learning styles of freshmen engineering students. In *proceedings of 2002 International Conference on Engineering Education.* Manchester: International Network for Engineering Education and Research.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05* (pp. 14–18). New York, NY: ACM Press.

Lamb, A., & Larry, J. (2011). Scratch: Computer Programming for 21st Century Learners. *Teacher Librarian*, 38(4), 64- 75.

Laurillard, D. (1979). The processes of student learning. *Higher Education*, *8*(4), 395–409.

Laurillard, D. (2005). Styles and approaches in problem-solving. In N. Entwistle, F. Marton, & H. Dai (Eds.), *The Experience of Learning* (pp. 126–144). Edinburgh: Scottish Academic Press

Laurillard, D., & Laurillard, D. (2006). E-learning in higher education. In Ashwin P. (Ed.) *Changing Higher Education (pp. 87-100).* London: Routledge.

Lave, J., & Wenger, E. (1991). Situated learning: legitimate peripheral participation. Cambridge: Cambridge university press.

Lawrence, G. (2012). People types and tiger stripes: a practical guide to learning styles (3rd ed.). Gainesville, FL: Center for Applications of Psychological Type, Inc.

Lawshe, C. H. (1975). A quantitative approach to content validity. *Personnel Psychology*, *28*(4), 563–575.

Leedy, P., & Ormrod, J. (2010). Practical research: planning and design. Boston, MA: Pearson.

Leitner, H. H., Malan, D. J., Maloney, J., & Wolz, U. (2009). Starting with Scratch in CS1. *ACM SIGCSE Bulletin, 41*(1), 2–3.

Lepper, M. R., & Cordova, D. I. (1992). A desire to be taught: Instructional consequences of intrinsic motivation. *Motivation and Emotion*, *16*(3), 187–208.
Letovsky, S., & Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, *3*(3), 41–49.
Li, X., & Yang, X. (2016). Effects of learning styles and interest on concentration and achievement of students in mobile learning. *Journal of Educational Computing Research*, *54*(7), 922–945.
Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist*, *20*(4), 191–206.
Liu, J., Lin, C.-H., Potter, P., Hasson, E. P., Barnett, Z. D., & Singleton, M. (2012). Going mobile with App Inventor for Android: A one-week computing workshop for K-12 teacher. *In Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13 (pp. 433-438)*. New York, NY: ACM Press.
lling, M. K., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education*, *13*(4), 249–268.
López, J. M. S., González, M. R., & Cano, E. V. (2016). Visual programming languages integrated across the curriculum in elementary school: A two-year case study using "scratch" in five schools. *Computers & Education*, *97(1)*, 1–26.
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. *ACM SIGCSE Bulletin*, *41*(3), 166.
Madan, R. L., & Tuli, G. D. (2003). Physical Chemistry. New Delhi: S. Chand Publishing.
Madrid, D. F., & Canado, M. L. P. (2001). Exploring the student's motivation in the EFL class. *Present and Future Trends in TEFL*, *12*(1), 321–364.
Malan, D. J. (2010). Reinventing CS50. In *Proceedings of the 41st ACM technical symposium on Computer science education - SIGCSE'10* (pp. 152–156). New York, NY: ACM Press.
Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education - SIGCSE '07* (pp. 223–227). New York, NY: ACM Press.
Malone, T. W. (1981). Toward a theory of intrinsically motivating instruction. *Cognitive Science*, *5*(4), 333–369.
Malone, T. W., & Lepper, M. R. (1987). Making learning fun: A taxonomy of intrinsic motivations for learning. In Snow, R. & Farr, M. J. (Ed), *Aptitude, Learning, and Instruction Volume 3: Conative and Affective Process Analyses.* Hillsdale, NJ: Lawrence Erlbaum Associates.
Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M., & Rusk, N. (2007). Programming by choice: urban youth learning programming with Scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education - SIGCSE '08 (pp.* 367-371). New York, NY: ACM Press.
Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, *10*(4), 1–15.
Martínez-Valdés, J. A., Velazquez-Iturbide, J. A., & Hijon-Neira, R. (2017). A (relatively) unsatisfactory experience of use of Scratch in CS1. In Dodero J., Sáiz M., Rube I. (Eds.) *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality - TEEM 2017* (pp. 1-7). New York, NY: ACM Press.

Marton, F., & Saljo, R. (1976). On qualitative differences in learning: Outcome and process. *British Journal of Educational Psychology*, *46*(1), 4–11.
Mayer, R. E. (1981). The psychology of how novices learn computer programming. *ACM Computing Surveys (CSUR)*, *13*(1), 121–141.
Mayes, J. T., & Fowler, C. J. (1999). Learning technology and usability: a framework for understanding courseware. *Interacting with Computers 11(5)*, 485–497.
McCall, D., & Kölling, M. (2015). Meaningful categorisation of novice programmer errors. In *Proceedings of 2014 IEEE Frontiers in Education Conference – FIE* (pp. 1–9). Madrid, Spain: IEEE.
McCane, B., Ott, C., Meek, N., & Robins, A. (2017). Mastery learning in introductory programming. In *Proceedings of the Nineteenth Australasian Computing Education Conference - ACE '17 (pp. 1-10).* New York, NY: ACM Press.
Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of programming in Scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE '11* (pp. 168-172). New York, NY: ACM Press.
Mega, C., Ronconi, L., & De Beni, R. (2014). What makes a good student? How emotions, self-regulated learning, and motivation contribute to academic achievement. *Journal of Educational Psychology*, *106*(1), 121–131.
Merriam, S. B. (1998). Qualitative research and case study applications in education. Revised and Expanded from "*Case Study Research in Education.*". San Francisco: CA, Jossey-Bass Publishers.
Merton, R. K., & Kendall, P. L. (1946). The Focused Interview. *American Journal of Sociology*, *51*(6), 541–557.
Messick, S. (1987). Validity. *ETS Research Report Series*, *1987*(2), i–208.
Michael, J. (2001). In pursuit of meaningful learning. *Advances in Physiology Education, 25*(3), 145-158.
Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming - Views of students and tutors. *Education and Information Technologies*, *7*(1), 55–66.
Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. In *Proceedings of the working conference on Advanced visual interfaces - AVI '04* (pp. 373–376). New York, NY: ACM Press.
Moreno-León, J., Robles G. (2015). Dr. Scratch: a web tool to automatically evaluate Scratch projects. In *Proceedings of the Workshop in Primary and Secondary Computing Education -WiPSCE '15* (pp132-133). New York, NY: ACM Press.
Morris, M. G., Speier, C., & Hoffer, J. A. (1999). An examination of procedural and object-oriented systems analysis methods: does prior experience help or hinder performance? *Decision Sciences*, *30*(1), 107–136.
Morrison, G. R., Ross, S. M., Kemp, J. E., & Kalman, H. (2011). Designing effective instruction (6 ed.). Hoboken, NJ: Wiley.
Moskal, Barbara, Lurie, D., & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. *ACM SIGCSE Bulletin*, *36*(1), 75–79.
Mozilla Developer Network. (2013). *Multi-Paradigm Programming Language*. Retrieved from: https://developer.mozilla.org/en-US/docs/multiparadigmlanguage.html
Mselle, L. J. (2010). Enhancing comprehension by using random access memory (RAM) diagrams in teaching programming: Class experiment. In *Proceedings of 22nd Annual Workshop - PPIG 2010.* Madrid, Spain: PPIG.

Muraina, I. O., Adegboye, A., Adegoke, M. A., & Olojido, J. B. (2019). Multimodal Instructional Approach: The Use of Videos, Games, Practical and Online Classroom to Enhance Students' Performance in Programming Languages. *American Journal of Software Engineering and Applications*, *8*(2), 44–49.
Murray, H. A. (1938). Explorations in personality. Oxford: Oxford University Press.
Myers, B. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing, 1*(1) 97–123.
Myers, I. B. (1998). MBTI manual: a guide to the development and use of the Myers-Briggs Type Indicator. Palo Alto, CA: Consulting Psychologists Press.
Naps, T. L. (1997). Algorithm Visualization on The World Wide Web - The Difference Java Makes! In Miller J., Davies G. (Eds.) *Proceedings of the 2nd conference on Integrating technology into computer science education - ITiCSE '97* (pp. 59-61). New York, NY: ACM Press.
Nicholls, J. G. (1984). Achievement motivation: conceptions of ability, subjective experience, task choice, and performance. *Psychological Review*, *91*(3), 328–346.
Nikou, S. A., & Economides, A. A. (2014). Transition in student motivation during a scratch and an app inventor course. In *Proceedings of IEEE Global Engineering Education Conference - EDUCON* (pp. 1042–1045). Istanbul: IEEE.
Norman, D. A. (1987). Some observations on mental models. In R. M. Baecker & W. A. S. Buxton (Eds.), *Human-computer interaction: a multidisciplinary approach* (pp. 241-244). San Francisco: Morgan Kaufmann Publishers Inc.
Norman, D. A. (1988). Design of everyday things. London: The MIT Press.
Osgood, C. E., Suci, G. J., & Tannenbaum, P. H. (1957). The measurement of meaning. Champaign, IL: University of Illinois Press.
Ouahbi, I., Kaddari, F., Darhmaoui, H., Elachqar, A., & Lahmine, S. (2015). Learning basic programming concepts by creating games with Scratch programming environment. *Procedia - Social and Behavioural Sciences*, *191*, 1479–1482.
Ozoran, D., Cagiltay, N., & Topalli, D. (2012). Using Scratch in introduction to programming course for engineering students. In *Proceedings of 2nd International Engineering Education Conference - IEEC2012* (pp. 125–132). Antalya, Turkey.
Panselinas, G., Fragkoulaki, E., Angelidakis, N., Papadakis, S., Tzagkarakis, E., & Manassakis, V. (2018). Monitoring students' perceptions in an App Inventor school course. *European Journal of Engineering Research and Science*, (CIE), 5–7.
Papadakis, S., & Orfanakis, V. (2018). Comparing novice programming environments for use in secondary education: App Inventor for Android vs. Alice. *International Journal of Technology Enhanced Learning*, *10*(1/2), 44–30.
Papadakis, S., Kalogiannakis, M., Orfanakis, V., & Zaranis, N. (2014). Novice Programming Environments. Scratch & App Inventor: a first comparison. In *Proceedings of the 2014 Workshop on Interaction Design in Educational Environments* - IDEE '14 (pp. 1-7). New York, NY: ACM Press.
Papert, S. (1980). Mindstorms: Computers, children, and powerful ideas. New York, NY: Basic Books.
Papert, S. (1987). Microworlds: transforming education. In R. W. Lawler & M. Yazdani (Eds.), *Artificial intelligence and education* (pp. 1–16). Norwood, NJ: Ablex Publishing.
Papert, S., & Harel, I. (1991). Situating constructionism. *Constructionism*, 36(2), 1-11.
Pask, G. (1976). Styles and strategies of learning. *British Journal of Educational Psychology*, *46*(2), 128–148.

Pask, G. (2010). Learning strategies, teaching strategies, and conceptual or learning style. In Schmeck, R (Ed.) *Learning Strategies and Learning Styles* (pp. 83–100). Boston, MA: Springer.

Pattis, R. E. (1993). The "procedures early" approach in CS1: a heresy. In *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education - SIGCSE '93* (pp. 122–126). New York, NY: ACM Press.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 1–13.

Pea, R. D., & Kurland, D. M. (1983). On the cognitive prerequisites of learning computer programming. *AEDS Journal*, *18*(3), 183-194.

Pears, A., Malmi, L., Adams, E., Bennedsen, J., Devlin, M., Seidman, S., *et al.* (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, *39*(4), 204–223.

Pellas, N., & Peroutseas, E. (2016). Leveraging Scratch4SL and Second Life to motivate high school students' participation in introductory programming courses: findings from a case study. *New Review of Hypermedia and Multimedia*, 23(1), 51–79.

Pennington, N. (1987a). Comprehension strategies in programming. In E. Soloway, G. M. Olson, & S. Sheppard (Eds.), *Empirical Studies of Programmers*: *Second Workshop* (pp. 100–113). Norwood, NJ: Ablex Publishing Corp.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*(3), 295–341.

Peterson, R. A. (1994). A meta-analysis of Cronbach's coefficient alpha. *Journal of Consumer Research*, *21*(2), 381–12.

Piaget, J. (1977). The development of thought: Equilibration of cognitive structures. (Trans A. Rosin). Oxford: Viking.

Pintrich, P. R. (2000). The role of goal orientation in self-regulated learning. In P. R. Pintrich, M. Boekaerts, & M. Zeidner (Eds.), *Handbook of Self-Regulation* (pp. 451–502). San Diego, CA: Academic Press.

Pintrich, P. R. (2004). A Conceptual Framework for Assessing Motivation and Self-Regulated Learning in College Students. *Educational Psychology Review*, *16*(4), 385–407.

Pintrich, P. R., & de Groot, E. V. (1990). Motivational and Self-Regulated Learning Components of Classroom Academic Performance. *Journal of Educational Psychology*, *82*(1), 33–40.

Pintrich, P. R., Smith, D. A. F., Garcia, T., & Mckeachie, W. J. (1991). *A Manual for the Use of the Motivated Strategies for Learning (MSLQ)* (pp. 1–75). Ann Arbor, Michigan: National Center for Research to Improve Postsecondary Teaching and Learning.

Pintrich, P. R., Smith, D. A. F., Garcia, T., & Mckeachie, W. J. (1993). Reliability and predictive validity of the motivated strategies for learning questionnaire (*MSLQ*). *Educational and Psychological Measurement*, *53*(3), 801–813.

Pittman, T. S., Emery, J., & Boggiano, A. K. (1982). Intrinsic and extrinsic motivational orientations: Reward-induced changes in preference for complexity. *Journal of Personality and Social Psychology*, *42*(5), 789–797.

Platsidou, M., & Metallidou, P. (2009). Validity and Reliability Issues of Two Learning Style Inventories in a Greek Sample: Kolb's Learning Style Inventory and Felder & Soloman's Index of Learning Styles. *International Journal of Teaching and Learning in Higher Education*, *20*(3), 324–335.

Polit, D. F., & Beck, C. T. (2004). Nursing research: Principles and methods (7th Edition). Philadelphia, PA: Lippincott Williams & Wilkins.
Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: teaching CS0 with Alice, 213–217. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education - SIGCSE '07* (pp. 213-217). New York, NY: ACM Press.
Prawat, R. S., & Folden, R. E. (1994). Philosophical perspectives on constructivist views of learning. *Educational Psychology*, *29*(1), 37–48.
Prensky, M. (2010). Teaching digital natives: Partnering for real learning. Thousand Oaks, CA: Corwin Press.
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.
Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program. *ACM SIGCSE Bulletin*, *36*(3), 171-176.
Ramsden, P. (1981). A study of the relationship between student learning and its academic context (Doctoral Dissertation). University of Lancaster, Lancaster.
Ratcliffe, M., Thomas, L., Woodbury, J., & Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. *ACM SIGCSE Bulletin, 34*, 33–37.
Rayner, S., & Riding, R. (2010). Towards a categorisation of cognitive styles and learning atyles. *Educational Psychology*, *17*(1-2), 5–27.
Reges, S. (2006). Back to basics in CS1 and CS2. *ACM SIGCSE Bulletin, 38,* 293–297.
Resnick, M., & Ocko, S. (1990). LEGO/logo: Learning through and about design. In Ferguson, D. (Ed.) *Advanced educational technologies for mathematics and science* (pp. 61–89). Berlin, Germany: Springer Berlin Heidelberg.
Resnick, M. (2007). All I really need to know (about creative thinking) I learned (by studying how children learn) in kindergarten, In *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition -C&C '07* (pp.1-6). New York, NY: ACM Press.
Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., *et al.* (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60.
Reynolds, M. (1997). Learning Styles: A Critique. *Management Learning*, *28*(2), 115–133.
Riding, R. J., & Sadler-Smith, E. (1997). Cognitive style and learning strategies: some implications for training design. *International Journal of Training and Development*, *1*(3), 199–208.
Rieber, Lloyd P, Smith, L., & Noah, D. (1998). The Value of Serious Play. *Educational Technology*, *38*(6), 29–37.
Rieber, Loyd P. (1992). Computer-based microworlds: A bridge between constructivism and direct instruction. *Educational Technology Research and Development*, *40*(1), 1–14.
Riechmann, S. W., & Grasha, A. F. (1974). A rational approach to developing and assessing the construct validity of a student learning style scales instrument. *The Journal of Psychology*, *87*(2), 213–223.
Rigby, P. C., & Thompson, S. (2005). Study of novice programmers using Eclipse and Gild. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange - eclipse '05* (pp. 105–109). New York, NY: ACM Press.
Robson, C., & McCartan K. (2016). Real world research (4nd ed). Chichester: John Wiley & Sons.

Rotter, J. B. (1966). Generalized expectancies for internal versus external control of reinforcement. *Psychological Monographs General and Applied*, *80*(1), 1–28.
Rotter, J. B. (1990). Internal versus external control of reinforcement: A case history of a variable. *American Psychologist*, *45*(4), 489–493.
Rugaber, S. (2007). Program comprehension for reverse engineering. In *AAAI Workshop on AI and Automated Program Understanding* (pp. 1-5). San Jose, CA: AAAI Press.
Runeson, P. (2012). Case study research in software engineering: guidelines and examples. *Case Study Research in Software Engineering: Guidelines and Examples*. Hoboken, NJ: John Wiley & Sons, Inc.
Ryan, R. M., & Deci, E. L. (2000). Intrinsic and extrinsic motivations: classic definitions and new directions. *Contemporary Educational Psychology*, *25*(1), 54–67.
Sajaniemi, J. (2002). Visualizing roles of variables to novice programmers. In Jing Y. (Ed.) *Proceedings of the 14th Annual Workshop on the Psychology of Programming Interest Group - PPIG-2012* (pp. 111–127). London: Brunel University.
Salta, K., & Koulougliotis, D. (2015). Assessing motivation to learn chemistry: adaptation and validation of Science Motivation Questionnaire II with Greek secondary school students. *Chemistry Education Research and Practice*, *16*(2), 237–250.
Saltan, F., & Kara, M. (2016). ICT teachers' acceptance of "Scratch" as algorithm visualization software. *Higher Education Studies*, *6*(4), 146–10.
Sanders, K., Boustedt, J., Eckerdal, A., Mccartney, R., Moström, J. E., Thomas, L., & Zander, C. (2012). Threshold concepts and threshold skills in computing. In *Proceedings of the ninth annual international conference on International computing education research - ICER '12* (pp.23-30). New York, NY: ACM Press.
Santos, Á., Gomes, A., & Mendes, A. J. (2010). Integrating new technologies and existing tools to promote programming learning. *Algorithms*, *3*(2), 183–196.
Savery, J. R., & Duffy, T. M. (2001). Problem based learning: An instructional model and its constructivist framework. *Educational technology, 35(5),* 31-38.
Savi , M., Ivanovic, M., Radovanovi , M., & Budimac, Z. (2016). Modula-2 versus Java as the first programming language: Evaluation of students' performance. In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016, Vol. 1164* (pp. 415-422). New York, NY: ACM Press.
Schmitt, N., & Stuits, D. M. (1985). Factors defined by negatively keyed items: The Result of Careless Respondents? *Applied Psychological Measurement*, *9*(4), 367–373.
Schriesheim, C. A. (1995). An exploratory and confirmatory factor-analytic investigation of item wording effects on the obtained factor structures of survey questionnaire measures. *American Educational Research Journal*, *21*(6), 1177–1193.
Schmeck, R. R. (Ed.). (1988). Learning strategies and learning styles. New York, NY: Plenum Press.
Schunk, D. H. (1991). Self-Efficacy and Academic Motivation. *Educational Psychologist*, *26*(3-4), 207–231.
Schunk, D. H. (2012). Learning theories: an educational perspective. Boston, MA: Pearson.
Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering, 25*(4), 557-572.

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality. *Biometrics*, *52*(3), 591–611.
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.
Shroff, R. H., Deneen, C. C., & Ng, E. M. W. (2011). Analysis of the technology acceptance model in examining students' behavioural intention to use an e-portfolio system. *Australasian Journal of Educational Technology*, *27*(4), 600-618.
Shu, N. C. (1989). Visual programming: perspectives and approaches. *IBM Systems Journal*, *28*(4), 525–547.
Shuell, T. J. (1986). Cognitive conceptions of learning. *Review of Educational Research*, *56*(4), 411.
Siemens, G. (2005). Connectivism: A learning theory for the digital age. *International Journal of Instructional Technology and Distance Learning, 2*(1). Retrieved from: http://itdl.org/journal/jan_05/article01.htm.
Skinner, B. F. (1953). Science and human behaviour. New York, NY: Macmillan.
Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 840–841.
Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. New Haven: Department of Computer Science, Yale University.
Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: an empirical study. *Communications of the ACM*, *26*(11), 853–860.
Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, *13*(4), 1–64.
Spearman, C. (1904). The proof and measurement of association between two things. *The American Journal of Psychology*, *15*(1), 72–101.
Stake, R. E. (1995). The Art of Case Study Research. Thousand Oaks, CA: Sage Publications, Inc.
Stefanou, V., Kotsovoulou, M., & Makri, D. (2018). Using E-Assessment Software to Support Formative Assessment: a Phenomenographic Study of Instructors' Experiences. In Chova L, Martinez A., & Torres I. (Eds.) *Proceedings of the 12th International Technology, Education and Development Conference - INTED 2018* (pp. 1066–1075), Valencia, Spain: IATED.
Steffe, L. P., & Gale, J. (1995). Constructivism in education. Hillsdale, N.J.: Lawrence Erlbaum.
Stenhouse, L. (1978). Case study and case records: towards a contemporary history of education. *British Educational Research Journal*, *4*(2), 21–39.
Stringer, E. T., Christensen, L. M. F., & Baldwin, S. C. (2010). Integrating teaching, learning, and action research: enhancing instruction in the k-12 classroom. Los Angeles, CA: Sage.
Sun, B. (2010). Java teaching based on BlueJ platform. In Zhiwei Y., Jun S. (Eds.) *Proceeding of the 2nd International Conference on Information Engineering and Computer Science -ICIECS* (pp. 1–4). Red Hook, NY: Curran Associates, Inc.
Sykes, E. R. (2007). Determining the effectiveness of the 3D Alice programming environment at the computer science I level. *Journal of Educational Computing Research*, *36*(2), 223–244.
Tavakol, M., & Dennick, R. (2011). Making sense of Cronbach's alpha. *International Journal of Medical Education*, *2*, 53–55.

Taylor, R. (2012, May). Review of the Motivated Strategies for Learning Questionnaire (MSLQ) Using Reliability Generalization Techniques to Assess Scale Reliability (Doctoral dissertation). Auburn University, Auburn, AL.

Thomas, G. (2011). A typology for the case study in social science following a review of definition, discourse, and structure. *Qualitative Inquiry*, *17*(6), 511–521.

Thomasson, B., Ratcliffe, M., & Thomas, L. (2006). Identifying novice difficulties in object-oriented design. *ACM SIGCSE Bulletin*, 38(3), 28–32.

Topalli, D., & Cagiltay, N. E. (2018). Improving programming skills in engineering education through problem-based game projects with Scratch. *Computers & Education*, *120*, 64–74.

Tsai, L. C., Tsai, L.-C., Hwang, S. L., Tang, K. H., Hwang, S.-L., & Tang, K.-H. (2011). Analysis of keyword-based tagging behaviors of experts and novices. *Online Information Review*, *35*(2), 272–290.

Tsai, W.-H., & Chen, L.-S. (2011). A study on adaptive learning of scratch programming language, In *Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government - EEE* (pp. 1–7). Athens, Greece: CSREA Press.

Tyler, R. W. (1949). Basic principles of curriculum and instruction. In Flinders D., Flinders S., Thornton D. (Eds.) *The Curriculum Studies Reader E2* (pp 60–68). London: Routledge.

Uguroglu, M. E., & Walberg, H. J. (1979). Motivation and achievement: A quantitative synthesis. *American Educational Research Journal*, *16*(4), 375–389.

University of Kent. (2014). Greenfoot: Transforming the way programming is taught. *Impact Case Studies - REF2014*. Retrieved from: https://impact.ref.ac.uk/casestudies/CaseStudy.aspx?Id=911.

Utting, I., Cooper, S., Kölling, M., Maloney, J., & Resnick, M. (2010). Alice, Greenfoot, and Scratch - A discussion. *ACM Transactions on Computing Education (TOCE)*, *10*(4), 17–11.

Van Gorp, M. J., & Grissom, S. (2001). An Empirical Evaluation of Using Constructive Classroom Activities to Teach Introductory Programming. *Computer Science Education*, *11*(3), 247–260.

Van Roy, P. (2009). Programming paradigms for dummies: what every programmer should know, In G. Assayag and A. Gerzso (Eds.) *New Computational Paradigms for Computer Music* (pp. 1-9). Paris, France: Delatour.

Veerasamy, A. K., DSouza, D., & Laakso, M.-J. (2016). Identifying novice student programming misconceptions and errors from summative assessments. *Journal of Educational Technology Systems*, *45*(1), 50–73.

Venkatesh, V., & Davis, F. D. (2000). A theoretical extension of the technology acceptance model: four longitudinal field studies. *Management Science*, *46*(2), 186–204.

Vogt, W. P. (2007). Quantitative research methods for professionals. Boston, MA: Pearson/Allyn and Bacon.

Vogts, D., Calitz, A., & Greyling, J. (2008). Comparison of the effects of professional and pedagogical program development environments on novice programmers. In *Proceedings of the 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 - SAICSIT '08* (pp. 286–295). New York, NY: ACM Press.

Vygotsky, L. S. (1978). Mind in society: the development of higher psychological processes. Cambridge, MA: Harvard University Press.
Wagner, A., Gray, J., Corley, J., & Wolber, D. (2013). Using App Inventor in a K-12 summer camp. *In Proceeding of the 44th ACM technical symposium on Computer science education -SIGCSE '13* (pp. 621–626). New York, NY: ACM.
Wainer, H., & Braun, H. I. (2013). Test Validity. Hillsdale, N.J: L. Erlbaum Associates.
Walker, C. O., Greene, B. A., & Mansell, R. A. (2006). Identification with academics, intrinsic/extrinsic motivation, and self-efficacy as predictors of cognitive engagement. *Learning and Individual Differences*, *16*(1), 1–12.
Watson, C., & Li, F. W. B. (2014). Failure rates in introductory programming revisited, In *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14* (pp. 39–44). New York, NY: ACM.
Weintrop, D. (2015). Comparing text-based, blocks-based, and hybrid blocks/text programming tools. In *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER '15* (pp. 283–284). New York, NY: ACM Press.
Weintrop, D., & Wilensky, U. (2015). To block or not to block, that is the question. In *Proceedings of the 14th International Conference on Interaction Design and Children - IDC '15* (pp. 199–208). New York, NY: ACM Press.
Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education*, *18*(1), 1–25.
Weng, F., Yang, R.-J., Ho, H.-J., & Su, H.-M. (2018). A TAM-based study of the attitude towards use intention of multimedia among school teachers. *Applied System Innovation*, *1*(3), 36–9.
Wertheimer, M. (1983). A Gestalt perspective on computer simulations of cognitive processes. *Computers in Human Behavior*, *1*(1), 19–33.
West, S., & Uhlenberg, D. (1970). Measuring motivation. *Theory into Practice, 9(1),* 47-55.
White, G., & Sivitanides, M. (2005). Cognitive differences between procedural programming and object-oriented programming. *Information Technology and management, 6*(4), 333-350.
White, R. W. (1959). Motivation reconsidered: The concept of competence. *Psychological Review*, *66*(5), 297–333.
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.
Wilson, B. C. (2010). A study of factors promoting success in computer science including gender differences. *Computer Science Education*, *12*(1-2), 141–164.
Wing, J. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.
Winslow, L. E. (1996). Programming pedagogy - a psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22.
Winter, R. (1987). Action-research and the nature of social inquiry: Professional innovation and educational work. Aldershot: Avebury.
Wixom, B. H., & Todd, P. A. (2005). A theoretical integration of user satisfaction and technology acceptance. *Information Systems Research*, *16*(1), 85–102.
Wolber, D. (2011). App Inventor and real-world motivation. In *Proceedings of the 42nd ACM technical symposium* (pp. 601–606). New York, NY: ACM.

Wolber, D., Abelson, H., & Friedman, M. (2015). Democratizing computing with App Inventor. *GetMobile: Mobile Computing and Communications*, *18*(4), 53–58.
Wolters, C. A. (1999). The relation between high school students' motivational regulation and their use of learning strategies, effort, and classroom performance. *Learning and Individual Differences*, *11*(3), 281–299.
Wulf, T. (2005). Constructivist approaches for teaching computer programming. In *Proceedings of the 6th conference on Information technology education - SIGITE '05* (pp. 245–248). New York, NY: ACM.
Xinogalos, S. (2014). Designing and deploying programming courses: Strategies, tools, difficulties and pedagogy. *Education and Information Technologies*, *21*(3), 559–588.
Xinogalos, S., Satratzemi, M., & Dagdilelis, V. (2006). An introduction to object-oriented programming with a didactic microworld: objectKarel. *Computers & Education*, *47*(2), 148–171.
Xinogalos, S., Satratzemi, M., & Malliarakis, C. (2015). Microworlds, games, animations, mobile apps, puzzle editors and more: What is important for an introductory programming environment? *Education and Information Technologies*, *22*(1), 145-176.
Yi it, M. F., Ba er, M., & Ondokuz, M. (2015). Learning difficulties and use of visual technologies in learning to program. *Participatory Educational Research*, *15*(2), 27–34.
Yin, R. K. (2003). Designing case studies. In Yin R. *Case study research: Design and methods* (pp. 19–56). Thousand Oaks, CA: Sage Publications.
Yukselturk, E., & Altiok, S. (2017). An investigation of the effects of programming with Scratch on the preservice IT teachers' self-efficacy perceptions and attitudes towards computer programming. *British Journal of Educational Technology*, *48*(3), 789-801.
Zainal, N. F. A., Shahrani, S., Yatim, N. F. M., Rahman, R. A., Rahmat, M., & Latih, R. (2012). Students perception and motivation towards programming. *Procedia - Social and Behavioral Sciences*, *59(1)*, 277–286.
Zimmerman, B. J. (1990). Self-regulated learning and academic achievement: an overview. *Educational Psychologist*, *25*(1), 3–17.
Zimmerman, B. J. (2008). Investigating self-regulation and motivation: historical background, methodological developments, and future prospects. *American Educational Research Journal*, *45*(1), 166–183.
Zimmerman, B. J., Bandura, A., & Martinez-Pons, M. (1992). Self-motivation for academic attainment: The role of self-efficacy beliefs and personal goal setting. *American Educational Research Journal*, *29*(3), 663–676.
Zualkernan, I. A., Allert, J., & Qadah, G. Z. (2006). Learning styles of computer programming students: A Middle Eastern and American comparison. *IEEE Transactions on Education*, *49*(4), 443–450.
Zusho, A., Pintrich, P. R., & Coppola, B. (2003). Skill and will: The role of motivation and cognition in the learning of college chemistry. *International Journal of Science Education*, *25*(9), 1081–1094.
Zwanenberg, N. V., Wilkinson, L. J., & Anderson, A. (2000). Felder and Silverman's Index of Learning Styles and Honey and Mumford's Learning Styles Questionnaire: How do they compare and do they predict academic performance? *Educational Psychology*, *20*(3), 365–380.

"Creating Java Programs with Greenfoot". Oracle Academy, 2015, https://academy.oracle.com/pages/greenfoot_course.pdf

"Design Process Games – Alice". Alice.org, 2016, https://www.alice.org/resources/lessons/design-process-games/

"Getting Started with Java Using Alice". Oracle Academy, 2016, https://academy.oracle.com/pages/alice_course.pdf.

"Java Fundamentals – Course Description". Oracle Academy, 2016, https://academy.oracle.com/pages/java_fundamentals_course.pdf

"Magic 8 Ball". Appinventor.mit.edu, 2016, https://appinventor.mit.edu/explore/ai2/magic-8-ball

"Mole Mash". Appinventor.mit.edu, 2016, https://appinventor.mit.edu/explore/ai2/molemash

Appendix One – Main Survey Instrument

Introduction to Information Technology and Programming

Section 1 - Demographic Information

6) Gender
Male Female Other (write in)_____ Prefer not to say

7) What is your age*
under 18 18-24 25-34 35-54 55+

8) What is your current College Level? *
Freshman Sophomore Junior Senior Graduate

9) What is your major? *
MIS IT Other Business Administration Major Other Arts & Sciences Major

10) What it your pathway?
Software Development Digital Media Network Technologies Undecided

Section 2 - Introduction to Programming - General Questions

11) Describe your current level of computer programming expertise in any programming language:

- Fundamental Awareness
- Novice (limited experience)
- Intermediate (practical application)
- Advanced
- Expert

Explanation of selections:
Fundamental Awareness *means that you just have an idea of what programming is, but you have never written a computer program.*
Novice *means that you had some limited experience in the past. For example, in ITC1070 Introduction to information systems, you had written small programs or even in high school you were taught programming concepts, but you do not feel that you know well the subject.* ***Intermediate*** *means that you had some programming experience in the past, and you are able to write small programs utilising the basic programming constructs (variables, selections and repetitions).*
Advanced *means that you are able to understand and write complete programs utilising object-oriented concepts.*
Expert *means that you are professional programmer and you have implemented complete software systems.*

12) Which programming languages have you been taught in the past? *

Java JavaScript Python C C++

Other - Write In:

13) Were you familiar with SCRATCH or any other block-based programming environment before this class?

Yes No

14) What was the main reason for registering for this module? (Check the one that BEST describes your feelings)

I enjoy programming
Jobs in programming pay well
I find programming challenging:
I think that it will improve my career prospects
I am interested in programming
I consider it an easy elective
Course time and day fitted my schedule
Introduction to programming is a requirement for my major
It was recommended by my advisor/friend/family
I am curious to find out what programmers do
Other - Write In:

Section 3a - Overall Evaluation and Acceptance for Scratch

What is your opinion about using *Scratch* as part of Introduction to Programming?

15) Boring 1 ______________[4 Neutral]_____________ 7 Fun
16) Not Effective 1 ______________[4 Neutral]_____________ 7 Effective
17) Not Enjoyable 1 ______________[4 Neutral]_____________ 7 Enjoyable
18) Irrelevant 1 ______________[4 Neutral]_____________ 7 Relevant
19) Unpleasant 1 ______________[4 Neutral]_____________ 7 Pleasant

20) I _____ to use *Scratch* to create my own programs/games.

Do not Intend 1 ______________[4 Neutral]_____________ 7 Intend

21) Using *Scratch,* I can create functional/operational games, which I can demonstrate to my friends and family.

Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

22) I find *Scratch* as a preferable way to introduce novices to programming than traditional teaching with Java

Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

Section 3b - Technology Acceptance Model

Perceived Ease of Use

23) Learning to operate *Scratch* is often frustrating. (*R)
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

24) It is easy for me to remember how to perform tasks inside the *Scratch* environment.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

25) I find it easy to get the *Scratch* to do what I want it to do.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

26) Usage of *Scratch* is clear and understandable.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

27) Overall, I find *Scratch* easy to use.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

Perceived Usefulness

28) *Scratch* helped me improve my computing skills.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

29) *Scratch* makes it easier to convey an algorithm into a program than in a text-based programming language.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

30) *Scratch* helped me clarify all stages of the software development process: requirements analysis, design, development and testing.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

31) Learning *Scratch* improved my programming skills (such as: using variables, obtaining user input, iteration, selection, code modularity etc).
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

32) Overall, I find *Scratch* useful for this module.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

(This question appears when the participant provides an answer greater than 4 in question 32)
33) For the clarification, of which basic programming concepts did you find *Scratch* useful?

Program Logic and Algorithm development
Variables
Loops (Iterations)
Conditions (Selections)
Procedures
Event-Handling
Keyboard Input
Input validation
Other - Write In: _________

Section 4 - Motivated Strategies for Learning Questionnaire

34) I believe I can master programming knowledge and skills.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

35) My career will not involve computer programming.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

36) I put enough effort into learning computer programming.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

37) Learning computer programming will help me get a good job.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

38) I work on practicing exercises and answering end of chapter questions even when I do not have to.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

39) I think that what I am learning in this class is not useful for me to know.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

40) I will use computer programming skills in my career.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

41) Learning computer programming is interesting.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

42) I believe I can earn a good grade in introduction to programming.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

43) It is important for me to learn how to program.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

44) I am not confident I will do well on computer programming tests/exams.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

45) I prepare well for programming tests and labs.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

46) Knowing how to program will give me a career advantage.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

47) I enjoy learning computer programming. I enjoy/like what I am learning in this programming class.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

48) I am confident I will do well on computer programming labs and projects.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

49) I find that when the teacher is talking, I think about other things and don't really listen to want is being said.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

50) I work on solving all exercises assigned by the instructor.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

51) I am confident I can learn all programming concepts taught in the course.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

52) I prefer class work that is challenging so I can learn new things.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

53) It's important to me to get an "A" in computer programming.
Strongly Disagree 1 __________ [4 Neutral] __________ 7 Strongly Agree

Section 5 - Index of Learning Styles (ILS) Learning Style Questionnaire

54) I understand something better after I...
   a. try it out.
   b. think it through.

55) I would rather be considered ...
   a. realistic.
   b. innovative.

56) When I think about what I did yesterday, I am most likely to get ...
   a. a picture.
   b. words.

57) I tend to ...
   a. understand details of a subject but may be fuzzy about its overall structure.
   b. understand the overall structure but may be fuzzy about details.

58) When I am learning something new, it helps me to...
   a. talk about it.
   b. think about it.

59) If I were a teacher, I would rather teach a course...
   a. that deals with facts and real-life situations.
   b. that deals with ideas and theories.

60) I prefer to get new information in...
   a. pictures, diagrams, graphs, or maps.
   b. written directions or verbal information.

61) Once I understand...
   a. all the parts, I understand the whole thing.
   b. the whole thing, I see how the parts fit.

62) In a study group working on difficult material, I am more likely to...
   a. jump in and contribute ideas.
   b. sit back and listen.

63) I find it easier...
   a. to learn facts.
   b. to learn concepts.

64) In a book with lots of pictures and charts, I am likely to...
   a. look over the pictures and charts carefully.
   b. focus on the written text.

65) When I solve math problems...
   a. I usually work my way to the solutions one step at a time.
   b. I often just see the solutions but then have to struggle to figure out the steps to get to them.

66) In classes I have taken...
   a. I have usually got to know many of the students.
   b. I have rarely got to know many of the students.

67) In reading non-fiction, I prefer...
something that teaches me new facts or tells me how to do something.
something that gives me new ideas to think about.

68) I like teachers...
   a. who put a lot of diagrams on the board.
   b. who spend a lot of time explaining.

69) When I'm analysing a story or a novel...
   a. I think of the incidents and try to put them together to figure out the themes.
   b. I just know what the themes are when I finish reading and then I have to go back and find the incidents that demonstrate them.

70) When I start a homework problem, I am more likely to...
   a. start working on the solution immediately.
   b. try to fully understand the problem first.

71) I prefer the idea of...
   a. certainty.
   b. theory.

72) I remember best...
   a. what I see.
   b. what I hear.

73) It is more important to me that an instructor...
   a. lays out the material in clear sequential steps.
   b. gives me an overall picture and relates the material to other subjects.

74) I prefer to study...
   a. in a group.

b. alone.

75) I am more likely to be considered...
   a. careful about the details of my work.
   b. creative about how to do my work.

76) When I get directions to a new place, I prefer...
   a. a map.
   b. written instructions.

77) I learn...
   a. at a fairly regular pace. If I study hard, I'll "get it."
   b. in fits and starts. I'll be totally confused and then suddenly it all "clicks."

78) I would rather first...
   a. try things out.
   b. think about how I'm going to do it.

79) When I am reading for enjoyment, I like writers to...
   a. clearly say what they mean.
   b. say things in creative, interesting ways.

80) When I see a diagram or sketch in class, I am most likely to remember...
   a. the picture.
   b. what the instructor said about it.

81) When considering a body of information, I am more likely to...
   a. focus on details and miss the big picture.
   b. try to understand the big picture before getting into the details.

82) I more easily remember...
   a. something I have done.
   b. something I have thought a lot about.

83) When I have to perform a task, I prefer to...
   a. master one way of doing it.
   b. come up with new ways of doing it.

84) When someone is showing me data, I prefer...
   a. charts or graphs.
   b. text summarising the results.

85) When writing a paper, I am more likely to...
   a. work on (think about or write) the beginning of the paper and progress forward.

b. work on (think about or write) different parts of the paper and then order them.

86) When I have to work on a group project, I first want to...

a. have a "group brainstorming" where everyone contributes ideas.
b. brainstorm individually and then come together as a group to compare ideas.

87) I consider it higher praise to call someone...

a. sensible.
b. imaginative.

88) When I meet people at a party, I am more likely to remember...

a. what they looked like.
b. what they said about themselves.

89) When I am learning a new subject, I prefer to...

a. stay focused on that subject, learning as much about it as I can.
b. try to make connections between that subject and related subjects.

90) I am more likely to be considered...

a. outgoing.
b. reserved.

91) I prefer courses that emphasise...

a. concrete material (facts, data).
b. abstract material (concepts, theories).

92) For entertainment, I would rather...

a. watch television.
b. read a book.

93) Some teachers start their lectures with an outline of what they will cover. Such outlines are...

a. somewhat helpful to me.
b. very helpful to me.

94) The idea of doing homework in groups, with one grade for the entire group...

a. appeals to me.
b. does not appeal to me.

95) When I am doing long calculations...

a. I tend to repeat all my steps and check my work carefully.
b. I find checking my work tiresome and have to force myself to do it.

96) I tend to picture places I have been...
   a. easily and fairly accurately.
   b. with difficulty and without much detail.

97) When solving problems in a group, I would be more likely to...
   a. think of the steps in the solution process.
   b. think of possible consequences or applications of the solution in a wide range of areas.

98) Can I contact you for a short interview? *

Yes No

99) Type your email to receive your Learning Style results!

____

Thank you, for your time and effort.

Maira Kotsovoulou

Appendix Two – Selection of Questions for the Motivation section of the Main Survey

In the following tables, all questions with Content Validity Ratio > 0.75, are highlighted. Minimum level of CVR for inclusion for 8 panellists is .75

Intrinsic Motivation Scale - CVR

		Score	CVR
Glynn S. (2011) - Intrinsic Goal Orientation & Task Value			
1	The computer programming, I learn is relevant to my life.	2	-0.5
3	**Learning computer programming is interesting.**	**7**	**0.75**
12	**Learning computer programming makes my life more meaningful.**	**8**	**1**
17	**I am curious about latest developments in the field of computer programming.**	**7**	**0.75**
19	**I enjoy learning computer programming.**	**8**	**1**
Pintrich P. & Groot E (1990) - Intrinsic Goal Orientation & Task Value			
1	**I prefer class work that is challenging so I can learn new things.**	**7**	**0.75**
5	**It is important for me to learn how to programme.**	**8**	**1**
6	**I like what I am learning in this class.**	**8**	**1**
9	**I think I will be able to use what I learn in this class in other classes.**	**7**	**0.75**
12	**I often choose programming exercises which will help me learn something, even if they require more work.**	**8**	**1**
17	**Even when I do poorly on a test, I try to learn from my mistakes.**	**8**	**1**
18	**I think that what I am learning in this class is useful for me to know.**	**8**	**1**
21	I think that what we are learning in this class is interesting.	6	0.5
25	**Understanding this subject is important to me.**	**7**	**0.75**
Pintrich *et al.* (1991) - Intrinsic Goal Orientation			
1	**In a class like this, I prefer course material that really challenges me so I can learn new things.**	**7**	**0.75**
16	**In a class like this, I prefer course material that arouses my curiosity, even if it is more difficult to learn**	**8**	**1**
22	The most satisfying thing for me in this course is trying to understand the content as thoroughly as possible	6	0.5
24	When I have the opportunity, I choose course assignments that I can learn from even if they don't guarantee a good grade.	6	0.5
Pintrich *et al.* (1991) - Task Value			
4	**I think I will be able to use what I learn in this course in other courses**	**8**	**1**
10	**It is important for me to learn the course material in this class**	**8**	**1**
17	**I am very interested in the content area of this course**	**7**	**0.75**
23	**I think the course material in this class is useful for me to learn.**	**7**	**0.75**
26	I like the subject matter of this course	6	0.5
27	Understanding the subject matter of this course is very important to me.	6	0.5

Self-Efficacy Scale - CVR

		Score	CVR
Glynn S. (2011)			
9	I am confident I will do well on computer programming tests.	8	1
14	I am confident I will do well on computer programming labs and projects.	8	1
15	I believe I can master computer programming knowledge and skills.	8	1
18	I believe I can earn a grade of “A” in computer programming.	8	1
21	I am sure I can understand computer programming.	7	0.75
Pintrich P. & Groot E (1990)			
2	Compared with other students in this class I expect to do well.	7	0.75
7	I'm certain I can understand the ideas taught in this course.	8	1
10	I expect to do very well in this class.	7	0.75
11	Compared with others in this class, I think I'm a good student.	7	0.75
13	I am sure I can do an excellent job on the problems and tasks assigned for this class.	7	0.75
15	I think I will receive a good grade in this class.	7	0.75
20	My study skills are excellent compared with others in this class.	7	0.75
22	Compared with other students in this class I think I know a great deal about the subject.	8	1
23	I know that I will be able to learn the material for this class.	7	0.75
Pintrich *et al.* (1991)			
12	I am confident I can understand the basic concepts taught in this course.	6	0.5
6	I am certain I can understand the most difficult material presented in this course.	6	0.5
29	I am certain I can master the skills being taught in this class.	6	0.5
31	Considering the difficulty of this course, the teacher, and my skill, I think I will do well in this class	6	0.5

Extrinsic Motivation - CVR

		Score	CVR
Glynn S. (2011) - Grade Motivation			
2	I like to do better than other students on computer programming tests.	8	1
4	Getting a good computer programming grade is important to me.	8	1
8	It is important that I get an "A" in computer programming.	7	0.75
20	I think about the grade I will get in computer programming.	7	0.75
24	Scoring high on computer programming tests and labs matters to me.	8	1

		Score	CVR
Glynn S. (2011) - Career Motivation			
7	**Learning computer programming will help me get a good job.**	**8**	**1**
10	**Knowing computer programming will give me a career advantage.**	**7**	**0.75**
13	**Understanding computer programming will benefit me in my career.**	**8**	**1**
23	**My career will involve computer programming.**	**7**	**0.75**
25	**I will use computer programming problem-solving skills in my career.**	**8**	**1**
Pintrich *et al.* (1991) - Extrinsic Goal Orientation			
7	Getting a good grade in this class is the most satisfying thing for me right now	6	0.5
11	The most important thing for me right now is improving my overall GPA, so my main concern in this class is getting a good grade	5	0.25
13	If I can, I want to get better grades in this class than most of the other students	5	0.25
30	I want to do well in this class because it is important to show my ability to my family, friends or others	6	0.5

Self-Regulation and Self-Determination - CVR

		Score	CVR
Glynn S. (2011) - Self-determination			
5	**I put enough effort into learning computer programming.**	**8**	**1**
6	**I use strategies (online courses, forums, books) to learn computer programming well.**	**8**	**1**
11	**I spend a lot of time learning computer programming.**	**8**	**1**
16	**I prepare well for computer programming tests and labs.**	**8**	**1**
22	**I study hard to learn computer programming.**	**8**	**1**
Questions proposed by the focus group			
	I spend a lot of time creating computer programs to improve my skills.	**8**	**1**
	I believe that is important to practice solving problems in order to learn to program.	**8**	**1**

		Score	CVR
Pintrich P. & Groot E (1990) - Self-Regulation & Effort Regulation			
32	**I ask myself questions to make sure I know the material I have been studying.**	**8**	**1**
34	**When work is hard I either give up or study only the easy parts. (*R)**	**8**	**1**
40	**I work on practice exercises and answer end of chapter questions even when I don't have to.**	**8**	**1**
41	**Even when study materials are dull and uninteresting, I keep working until I finish.**	**8**	**1**
43	**Before I begin studying, I think about the things I will need to do to learn.**	**8**	**1**
45	**I often find that I have been reading for class but don't know what it is all about. (*R)**	**8**	**1**
46	**I find that when the teacher is talking, I think of other things and don't really listen to what is being said. (*R)**	**8**	**1**
52	When I'm reading, I stop once in a while and go over what I have read.	5	0.25
55	**I work hard to get a good grade even when I don't like a class.**	**8**	**1**
Pintrich *et al.* (1991) - Self-Regulation			
33	During class time I often miss important points because I am thinking of other things	6	0.5
36	When study for this course, I make up questions to help me focus my studying	6	0.5
41	**When I become confused about something I am trying, I go back and try to figure it out**	**8**	**1**
44	**If course materials are difficult to understand, I change the way study the material.**	**8**	**1**
54	Before I study new course material thoroughly, I often skim it to see how it is organised.	5	0.25
55	I ask myself questions to make sure I understand the material I have been studying in this class	5	0.25
56	I try to change the way I study in order to fit the course requirements and instructor's teaching style.	5	0.25
57	I often find that I have been reading for class, but I do not know what it was all about	5	0.25
61	I try to think through a topic and decide what I am supposed to learn from it rather than just reading it over when studying	5	0.25
76	When studying for this course I try to determine which concepts I don’t understand well.	6	0.5
78	**When I study for this class, I set goals for myself in order to direct my activities in each study period**	**7**	**0.75**
79	If I get confused taking notes in class, I make sure I sort it out afterwards.	5	0.25

Appendix Three – Qualitative Analysis

Analysis of Interviews

Interviews were coded using a deductive/inductive approach. Using the deductive approach, I created the two main themes (advantages and disadvantages) before starting the analysis.

The next step was to break up each interview into paragraphs and classify related paragraphs into one of the two general themes.

Grouping all quotes which were identified to clearly belong in each theme (refer to the extract below) and using a list of closed codes relative to the interests of study (useful, easy, enjoyable and interesting), I created the main codes. Similar codes which emerged from the text were merged into the main codes. Such an example is the code "like" which was merged with the code "enjoyable". Sub-codes were then created from the text with open coding and using constant comparisons codes were grouped into similar concepts. Figure 8.1 shows the final hierarchy chart of nodes, produced by NVivo software, in the Advantages Theme.

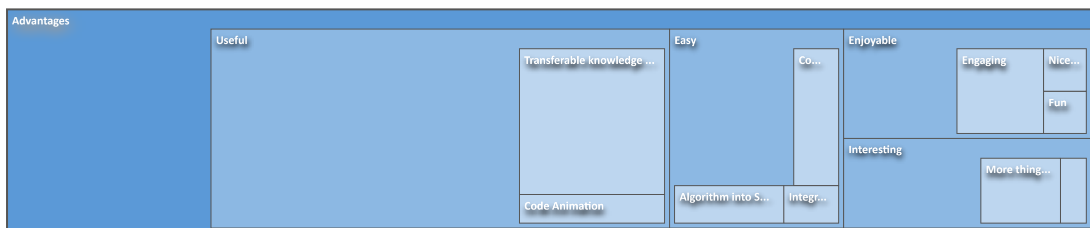


Figure 8.1: Hierarchy chart of nodes in the advantages theme

In the following extract, which was generated by grouping all student quotes which were identified as advantages in NVivo, I have highlighted in yellow the main codes and in green the sub-codes. For example, some students found Scratch useful because of code aminations and transferable knowledge into Java.

Participant 1 - 8 references coded

Yes, yes, I found it useful. A nice idea. Code animation made clear the execution of the program.

It introduced to me complex concepts in an easy way, like implementing code which could execute concurrently and across sprites, I guess. I had not encountered that before.

I also liked the interface.....all the commands required to develop a program were easily available and grouped

Participant 2 - 4 references coded

I found useful how it grouped the commands... For example, the all the commands that had to be executed if a condition was true were grouped together...

I think is just so satisfying to fix a program and see that it can actually do it...
→ VISUALIZE

Participant 3 - 4 references coded

The first project in SCRATCH motivated me to spend more time and put effort to make a more complete program, rather than a simple game. → ENGAGING

Yes, it was fun.

Participant 4 - 4 references coded

Yes, it was enjoyable...I worked a lot for the assignment... to make it better…

It got more interesting as I had to develop my game.

There were more things to explore, It helped me compartmentalise my thinking in order to achieve certain things. Make it a mechanical process... I think it helped very much with the organisation of thoughts. You knew that you wanted your project to do these things and it helped you visualise the end result...

I also did several other games... it was fun.

Participant 5 - 5 references coded

I liked SCRATCH. It makes thing more simple and easier to understand. Shows you how things work.

For the basic constructs, loops etc. I found it useful.

Participant 6 - 5 references coded

I think I enjoyed the SCRATCH process more...especially the game concept.

And aside from the programming, we could access the sprites themselves and change the colour and the backgrounds and make it more interactive...

I did enjoy it more... it was more engaging than JAVA.

Participant 7 - 9 references coded

I found it interesting...
I think it helps because it demonstrates how thinks work.
It is similar to JAVA. I found many similarities.
Yes, is was useful because it taught me the logic... how to approach a problem, find the inputs, the processing and produce results...

Yes, I understood that we used it in order to enhance our understanding of the steps required to create a program.

Participant 8 - 7 references coded

I found SCRATCH enjoyable. In my SCRATCH project I had more freedom to improvise... I had the opportunity to make it as easy or as difficult as I wanted...

It helped in understanding loops as well but more it helped me in algorithmic thinking... visualize the program structure.

Sample Interview Transcript – Participant 1

Transcript	Speaker
Hello, participant 1! Before we start, I would like to know what your pathway is?	Interviewer
Software Development	Participant 1
Let's begin from your past experience with computers. Did you have any prior programming experience?	Interviewer
I did not have any prior experience with programming... Maybe flowcharts in high school... very basic concepts... no pseudocode.	Participant 1
When did you decide that you wanted to study Information Technology?	Interviewer
Before coming to college, I was for 5 years in Medical School. I did not finish it, I got bored... and I decided to study Information Technology	Participant 1
What is your opinion about programming?	Interviewer
In general, I like computers, now that I have seen programming, I really like it, but I also like computer games!	Participant 1
Do you play games?	Interviewer

Yes, I do... but I was not involved with programming a game in the past.	Participant 1
Did you enjoy SCRATCH?	Interviewer
Yes, I did.	Participant 1
What did you like about SCRATCH?	Interviewer
I liked the interface……all the commands required to develop a program were easily available and grouped.	Participant 1
Did you find SCRATCH easy to use?	Interviewer
Yes, I found SCRATCH pretty easy to use.	Participant 1
Was there something about SCRATCH that struct to you as important?	Interviewer
Yes, now that I know the basics, I find it easy to convert an algorithm into a SCRATCH program... SCRATCH code looks like pseudocode… but I general I find programming easy...I think I have programming thinking...	Participant 1
So, you say that you find it easy to write a computer program with or without SCRATCH...	Interviewer
Yes, in general I am good in Math and Physics. I think I have the required structured thinking and logic which is required for programming as well... programming is easy for me...	Participant 1
So, did you find SCRATCH useful for this introductory module?	Interviewer
Yes, yes, I found it useful. A nice idea overall. Code animation made clear the execution of the program.	Participant 1
Did you find it interesting?	Interviewer
In the beginning I found it easy. Interesting became when the requirements and problems got more challenging. It was interesting to explore how to solve a problem, focus on the details and produce a near perfect result	Participant 1
Which where the main advantages that you saw in the use of SCRATCH?	Interviewer
It's easy to learn and I liked the interface... The programmer has all the commands available required to develop a program. He/she can insert images... can integrate sounds and animation, which would not be easy in other development environments.	Participant 1
Did you find any disadvantages?	Interviewer
I am not sure... I think it has some limitations... and it looks somewhat childish.	Participant 1
In SCRATCH you were introduced to basic programming concepts like variables, loops, conditions etc. Do you think that the knowledge you gained transferred or helped you understand better the concepts using JAVA?	Interviewer

Yes, I think it did. It also introduced to me more complex concepts in an easy way, like implementing code which could execute concurrently and across sprites, I guess. I had not encountered that before.	Participant 1
Do you think you were more motivated to create your SCRATCH game or your JAVA program?	Interviewer
Although SCRATCH was fun, I was more motivated to develop my JAVA program because it was more advanced…	Participant 1
At CS50 in Harvard university, they also use SCRATCH for the introduction to programming… Any comments on that?	Interviewer
Really? I had no idea… Maybe they know better…	Participant 1
If there was one thing you would recommend, in respect to technology used in this course, what would this be?	Interviewer
I found the course too easy. I need to be challenged more...But I also observed that some students were challenged.	Participant 1
Are other comments about the course?	Interviewer
No, no other comments...	Participant 1
Thank you very much for your time!	Interviewer

Sample Interview Transcript – Participant 4

Transcript	Speaker
Hello, P4. Before we start, I would like to know what your pathway is?	Interviewer
It's between software development and digital media...	Participant 4
When did you decide that we wanted to study Information Technology?	Interviewer
Hm... from a very young age... Maybe junior high... My mom introduced me to CodeAcademy, and I found it very interesting... I started with basic HTML... and I like that way of thinking. Algorithmic structure.	Participant 4
What is your opinion about programming?	Interviewer
My general opinion is that it is interesting and can be easy at certain aspects and difficult as well like learning a new language... So far, I like programming and the logic behind it. I also like the different ways which you can solve a problem...I guess	Participant 4
Did you enjoy SCRATCH?	Interviewer
In the beginning I thought it was a bit tedious, I guess because it was so simple, but I understand why it was necessary...for people who may not have done programming...	Participant 4
Did you find SCRATCH easy to use?	Interviewer
Yes.	Participant 4
Was there something about SCRATCH that struct to you as important?	Interviewer
Yes, implementing code which could execute concurrently and across sprites, I guess. I had not encountered that before...	Participant 4

Did you find it useful for this introductory module?	Interviewer
I found it useful. I did know SCRATCH before this course, but I did not know it that well, at that level.	Participant 4
Interesting?	Interviewer
It got more interesting as I had to develop my game. Required me to explore it more because I had it and then it became more interesting. It looks pretty easy in the beginning, but it becomes increasingly difficult... It did come to me to search for other people's project to see how they did some tasks...	Participant 4
Which where the main advantages that you saw in the use of SCRATCH?	Interviewer
It helped me compartmentalise my thinking in order to achieve certain things. Make it a mechanical process... I think it helped very much with the organisation of thoughts. You knew that you wanted your project to do these things and it helped you visualise the end result...	Participant 4
Any disadvantages?	Interviewer
Maybe... the fact that if you did not know that something was possible in SCRATCH you might not be able to do it at all...	Participant 4
In SCRATCH you were introduced to basic programming concepts like variables, loops, conditions etc. Do you think that the knowledge you gained transferred or helped you understand better the concepts using JAVA?	Interviewer
I think I had the thinking already, but it helped me solidify it more. I can see how it would help someone that does not have it already... can help them develop their thinking.	Participant 4
Do you think you were more motivated to create your SCRATCH game than your JAVA program?	Interviewer
I think I enjoyed the SCRATCH process more...the game concept. And we had aside from the programming, we could access the sprites themselves and change the colour and the backgrounds and make it more interactive... I did enjoy it more... it was more engaging...	Participant 4
Did you create any other SCRATCH games for entertainment purposes?	Interviewer
Yes, I enhanced sometimes games we did in class... I also did several other games... it was fun.	Participant 4
Did you find some transferable skills to your JAVA programming?	Interviewer
I think that the concept of methods... I was able to understand it better after we did SCRATCH.	Participant 4
Comment on HARVARD using SCRATCH in CS50	Interviewer
ha! (surprise)	Participant 4
Where there any concepts which you found more challenging?	Interviewer
In general, the entire section of arrays... it did not just click for me immediately I guess... It took me more time to adapt to it...	Participant 4

Where there any concepts which you found easy?	Interviewer
No, no…	Participant 4
Are there any aspects of the course, that motivated you to learn...	Interviewer
I guess the whole structure. The handwritten algorithm on the board and the use of SCRATCH demonstrated the proper structure of how it should go, because it helps people that may not know. Help me also understand and get a more well-rounded idea of how we should structure our code...	Participant 4
Any aspects of the course that discouraged you from learning?	Interviewer
No, I liked following along with the live programming. It was very helpful that you demonstrated the code and then we had to do an exercise from the very beginning...	Participant 4
Do you have anything to recommend as far as the teaching methodology is concerned? Or Are there any recommendations for the course in general?	Interviewer
Yes, towards the end of the course in the last few lessons, we were able to look at more advanced IDEs like Oracle's JDeveloper, maybe if we had seen a little bit more of it... it would be easier for students to move on and we were explaining object oriented programming and ready-made code... and all these things that seemed impossibly complex...	Participant 4
Are other comments about the course?	Interviewer
I liked the video tutorials you posted. Especially in programming. when you do not understand something the first time, you can repeat it until you get it. Also, the video with the demonstration of the completed coursework was very useful...	Participant 4
Thank you very much for your time!	Interviewer

Summary from Class Observation Notes

- **Session 1** – assignment: solve the maze (level of difficulty: easy)

Emotional expressions: mostly smiles.

Attention to the task: student demonstrated great attention, they wanted to solve it… competition (I did it!!!).

Perseverance: the ones who could not complete the code they used a search engine to search on the internet for solutions, they implemented the solutions and tested them out.

Performance: all students solved the problem.

- **Session 2** – assignment: create a birthday cake (level of difficulty: easy)

Emotional expressions: Playful mood; laughs and smiles.

Attention to the task: initially bored with birthday cake childishness, but later intrigued by the fact that the candles could be blown out using input from the computer microphone.

Performance: all students solved the problem.

- **Session 3** – assignment: create a fruit ninja game (level of difficulty: medium)

Emotional expressions: excitement to develop a “familiar” game, overall positive expressions.

Attention to the task: curiosity: some of them were wondering how it could be played on the computer

Performance: all students solved the basic problem. 50% of the students improved the code by adding extra elements like scores, timers, improved graphics.

- **Session 4** – assignment: create a hunting game (level of difficulty: medium-hard)

Emotional expressions: most students demonstrated positive emotions such as enjoyment, interest and curiosity.

Attention to the task: Most students were focused, but few were bored and appeared distracted (went online and started browsing other websites).

Persistence in accomplishing the task: most students completed the task on time, while some others stayed even after the break to finish their game. The ones who were initially bored did not complete the task.

Performance: Almost all students finished the game except the ones who appeared bored (but they mentioned that they found the task difficult to complete and did not know how to approach the solution. Not all implementations were excellent.

Appendix Four – Ethics Approval Forms

Lancaster University

From: Ethics (RSO) Enquiries <ethics@lancaster.ac.uk>
Date: June 25, 2013 7:21 PM
To: Kotsovoulou, Maria <m.kotsovoulou@lancaster.ac.uk>
CC: Passey, Don <d.passey@lancaster.ac.uk>, Jesmont, Alice <a.jesmont@lancaster.ac.uk>

Stage 1 self-assessment approval

Dear Maira,

Thank you for submitting your completed stage 1 self-assessment form and the additional information for **Exploring student perceptions about the use of visual programming environments, their relation to student learning styles and their impact on student motivation in undergraduate introductory programming modules**. I can confirm that approval has been granted for this project. As principal investigator your responsibilities include:

- ensuring that (where applicable) all the necessary legal and regulatory requirements in order to conduct the research are met, and the necessary licenses and approvals have been obtained;
- reporting any ethics-related issues that occur during the course of the research or arising from the research (e.g. unforeseen ethical issues, complaints about the conduct of the research, adverse reactions such as extreme distress) to the Research Ethics Officer;
- submitting details of proposed substantive amendments to the protocol to the Research Ethics Officer for approval.

Please contact the Research Ethics Officer, Debbie Knight (ethics@lancaster.ac.uk 01542 592605) if you have any queries or require further information.

Kind regards,

Debbie Knight
Research Ethics Officer Research Support Office B58, B Floor,
Bowland Main Lancaster University Lancaster, LA1 4YT
Email: ethics@lancaster.ac.uk Tel: 01524 592605
Web: Ethical Research at Lancaster: http://www.lancs.ac.uk/depts/research/lancaster/ethics.html

College XYZ

Institutional Review Board

November 14th, 2017

Maira Kotsovoulou
Assistant Professor, Information Technology

Dear Ms. Kotsovoulou,

Thank you for submitting your study entitled, “Exploring student perceptions about the use of visual programming environments in undergraduate introductory programming modules, and their impact on student motivation and performance”. The IRB has determined that your study is **Exempt** from committee review.

Please keep in mind that the IRB Committee must be contacted if there are any changes to your research protocol. The number assigned to your protocol is 201709066. Feel free to contact the IRB [irb@acg.edu] if you have any questions.

Best Wishes for your research work.

Sincerely,

Chair, IRB

Cc: Office of Provost

Appendix Five – Pilot Study's Programming Activities

Greenfoot – Guided Activity ("Creating Java Programs with Greenfoot")

Consider the following scenario: you are the pilot of a plane that has been sent out to pick up barrels that have fallen off a cargo ship. By flying over a barrel, you will automatically collect it. The problem is that sea is rough, and the barrels keep going under the water, so you have to collect them when they are at the surface. To make matters worse the area is a rocket testing site for rockets from NASA and they are unable to stop these being fired into the same area that you are in! You are going to program the above game.

The plane will always be moving but we can control its left and right turning. Flying over a barrel will mean you collect it and search for the next one. Collecting a barrel will give us a score point. The barrel will randomly appear on the screen, but only stay for a set amount of time and then go under and reappear randomly elsewhere on the screen. The rockets will appear at the top of the screen then randomly move down the screen until they disappear off the map. You must not hit the rocket. It is estimated that you will be able to survive 3 direct rocket hits and then the game will be over.

A sample of a completed game is shown in Figure 8.2: .

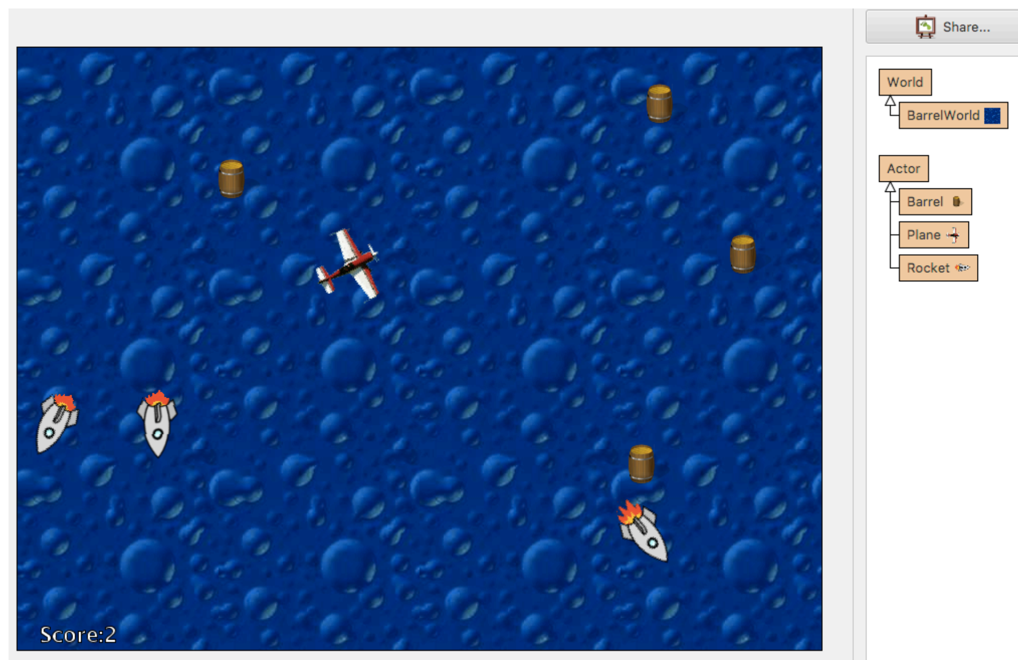


Figure 8.2: Eat the barel

Concepts demonstrated:

- variables, arrays, methods, parameters;
- conditions;
- loops;
- classes, subclasses and inheritance;
- abstraction;
- user-defined methods;
- keyboard movement and event listeners;
- sound, animation;
- game mechanics (score, win/lose conditions);
- code documentation.

Alice – Guided Activity

Lets' create a racing game. The first step is to create the map and include the start and finish lines and include a number of barriers for collision, see Figure 8.3. We will define game mechanics, such as a timer to complete the race and car health, and write custom procedures to implement it. The player will use the keyboard to drive the car.

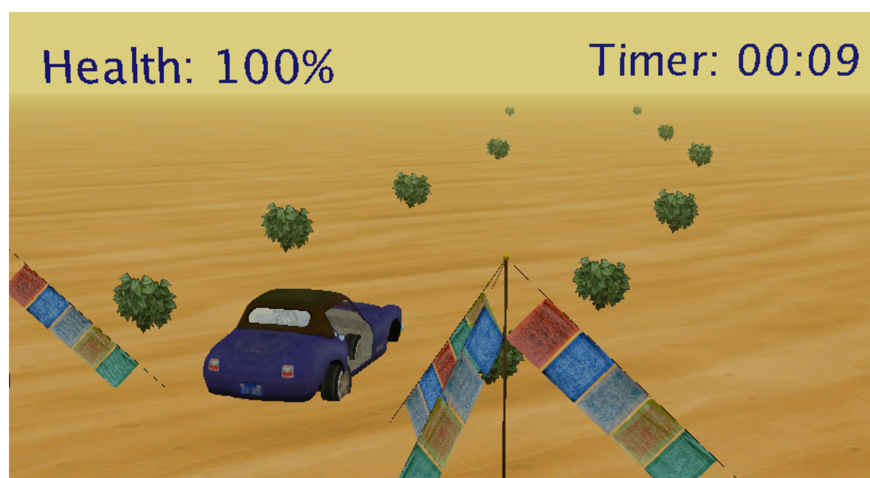


Figure 8.3: Race Game

Concepts demonstrated:

- variables, methods, parameters;
- conditions;
- loops;
- user-defined methods (procedures/functions)
- math expressions and random numbers;
- classes, subclasses and inheritance;
- repositioning objects at runtime;
- sound;
- keyboard movement and event listeners;
- camera views and markers;
- game mechanics (score, win/lose conditions);
- code documentation.

APP Inventor – Workshop Activities

Activity 1 - Magic 8 Ball ("Magic 8 Ball")

This introductory application demonstrates basic programming concepts and will help you learn how to navigate APP Inventor environment: Designer, Blocks editor and Emulator.

Concepts demonstrated:

- image sprites
- variables and lists (arrays)
- conditions
- events and basic event handling
- Accelerometer

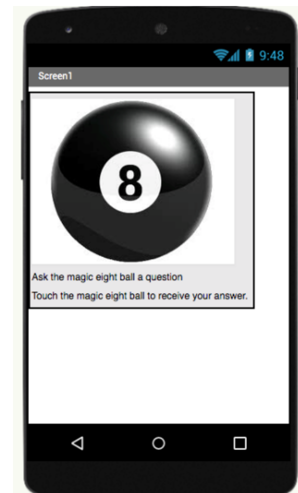


Figure 8.4: Magic-8 Ball

Activity 2 - Mole Mash ("Mole Mash")

In the game MoleMash, a mole pops up at random positions on a playing field, and the player scores points by hitting the mole before it jumps away. You'll design the game so that the mole moves once every half-second. If it is touched, the score increases by one, and the phone vibrates. When you miss the mole ten times the games should end. Pressing restart resets the score to zero.

Concepts demonstrated:

- variables;
- buttons and text blocks;
- math expressions and random numbers;
- conditions
- loops
- procedures;
- events and event handling;
- repositioning objects at runtime;
- timers and the clock component;
- game mechanics (score, win/lose conditions)

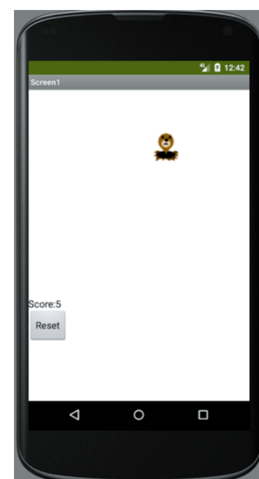


Figure 8.5: MoleMash

Scratch – Workshop Activities

Activity 1 – Virtual Network and DNS

In this activity, you are going to create a virtual network using Scratch. The idea is that you tell the packet (the yellow ball) to go to a certain computer (one of five) using an IP address. You type in a message (the data) and the packet should travel to the router and then to the destination computer with the correct IP address. The packet has to touch the computer in order to deliver the message. Extend your project by adding domain name server functionality. Instead of asking the user to go to a computer using its IP number, associate each IP with a domain name and ask the user to type a name. Your final project should look like figure.

Figure 8.6: IP packet switch

Concepts demonstrated:

- user input;
- variables;
- arrays;
- mouse interactions;
- animation.

Activity 2 – Match the Flag (Quiz)

In this activity, you are going to a create a Quiz Game. Game starts with a splash screen, see **Error! Reference source not found.** and the purpose of the g ame is to display a random set of 6 flags from a collection of 24 flags. When game starts a sprite character calls out the name of a country. The player has to select the correct flag within 5 seconds. The quiz should end after 6 rounds and display as a score as correct answer percentage.

Concepts demonstrated:

- variables;
- arrays;
- random numbers;
- conditions;
- loops;
- code modularity and custom blocks;
- message broadcasting;
- cloning;
- timers;
- game mechanics (score, win/lose conditions)

Figure 8.7: "Guess the Flag" game splash screen