

Learning to Program: The Development of Knowledge in Novice Programmers

Nadia Kasto

A thesis submitted to
Auckland University of Technology
in fulfilment of the requirement for the degree of
Doctor of Philosophy (PhD)

2016

School of Engineering, Computer and Mathematical Sciences
Auckland, New Zealand

Primary Supervisor: Assoc. Professor Jacqueline Whalley

Secondary Supervisor: Anne Philpott

Third Supervisor: Professor Stephen MacDonell

Abstract

This thesis presents a longitudinal study of novice programmers during their first year learning to program at university. The purpose of this research was to gain a deeper understanding of the ways in which novice programmers learn to program with an emphasis on their cognitive development processes. The intended outcome was a better understanding of the learning processes of novice programmers, which should enhance the ability of educators to teach, design courses, and assess programming. A key aspect of this research focused on cognitive development theories of Piaget, Vygotsky, Sfard and Cognitive Load and to what degree these theories could explain observations of novice programmers learning to write code.

In order to observe and investigate how novice programmers integrate new programming structure, concepts or elements into their current understanding of code it is necessary to be able to measure how difficult writing tasks are. Thus, the first aim of this research was to develop a task difficulty framework, which consisted of a new empirically verified software metric (code structure and readability) and a SOLO classification (task complexity) for code writing tasks. This framework was then used to design nineteen code writing tasks which were of increasing difficulty and complexity so as to trigger situations that required some form of knowledge adaptation or acquisition. Over one academic year, students were observed attempting to solve these programming tasks using a think aloud protocol and were interviewed retrospectively using a stimulated recall method. These observations were then linked to the cognitive theories in a way that provides an explanation of how programming was learned by these students.

The results of this research indicate that both cognitive and sociocultural approaches are important in the development of knowledge of novice programmers. Of the theories examined two were found to be the most useful. The first is Vygotsky's notions of the Zone of Proximal Development, the role of *more knowledgeable others*, and recent ideas about scaffolding. The second is Sfard's theory of concept development that contributes to a deeper understanding of the way novice programmers' develop patterns and reuse them in solving another programming task. The evidence about learning obtained during this study provides strong support for a change in the size and organization of the classes in which novice programmers are typically taught and in the teaching methods used.

Table of Contents

Abstract	i
Table of Contents	ii
List of Figures	vi
List of Tables.....	x
List of Abbreviations.....	xi
Attestation of Authorship.....	xii
Dedication	xiii
Acknowledgements	xiv
List of Publications	xv
Chapter 1. Introduction	1
1.1. Background of the Research	1
1.2. Key Definitions.....	3
1.3. Research Questions.....	4
1.4. Research Design	4
1.5. Structure of the Thesis	5
Chapter 2. Background	7
2.1. Introduction.....	7
2.2. General Theories of Cognitive Development and Learning.....	7
2.2.1. Piaget and Neo-Piagetian Theories	8
2.2.2. Vygotsky’s Theory and the Notion of Scaffolding.....	14
2.2.3. Sfard’s Theory.....	18
2.2.4. Cognitive Load Theory	21
2.3. Theories of Learning: Knowledge and Strategy	26
2.4. Transfer in Cognition.....	28
2.5. Summary.....	32
Chapter 3. Research Methodology	33
3.1. Introduction.....	33
3.2. A Pragmatic Research Approach.....	33
3.3. Research Instrument Design	35
3.4. Ethics Consents.....	37
3.5. Research Participants	38
3.5.1. Recruitment.....	38
3.5.2. Sampling Methods - Participant Selection.....	38

3.5.2.1.	Sampling for the Difficulty Framework	38
3.5.2.2.	Selecting Participants for Think Aloud Observations	39
3.6.	Think Aloud Method	40
3.6.1.	Think Aloud Data Collection Protocol	41
3.6.1.1.	Training.....	41
3.6.1.2.	Instruction	41
3.6.1.3.	Setting	42
3.6.1.4.	Recording Think Aloud	42
3.6.2.	Pilot Study & Data Collection Method Refinement	42
3.6.3.	Retrospective Interviews.....	44
3.6.4.	Stages of Verbal Protocol Analysis	44
3.6.4.1.	Transcription and Segmentation.....	45
3.6.4.2.	Transcript Encoding Techniques	47
3.7.	Intervention.....	49
3.8.	Summary.....	51
Chapter 4.	Framework Design.....	52
4.1.	Introduction.....	52
4.2.	Task Complexity vs. Task Difficulty.....	52
4.3.	Educational Taxonomies.....	53
4.4.	Software Metrics.....	56
4.5.	Software Metrics and Learning to Program.....	59
4.5.1.	Complexity Metrics.....	60
4.5.2.	Readability Metrics	63
4.6.	Selecting the Metrics: A GQM Approach	65
4.6.1.	Evaluating the Metrics	69
4.6.2.	Data Set	69
4.6.2.1.	Data Analysis and Results	70
4.6.2.2.	Factor Analysis-Principal Axis Factor.....	72
4.7.	Summary.....	77
Chapter 5.	Research Instrument Design.....	79
5.1.	Introduction.....	79
5.2.	Programming Courses at AUT	79
5.2.1.	The P1 Teaching Approach.....	80
5.2.2.	The P2 Teaching Approach.....	83
5.3.	The Design Process.....	84

5.4.	SOLO Classification	85
5.5.	Transfer Learning: Classification of the Tasks	91
5.6.	An Overview of the Tasks	93
5.7.	Sequence 1 – Counting Corridors.....	99
5.8.	Sequence2 – Counting Beepers	102
5.9.	Sequence3 – One-Dimensional Array	105
5.10.	Sequence4 – Two-Dimensional Array.....	106
5.11.	Sequence5 – ArrayList.....	107
5.12.	Summary	109
Chapter 6. Think Aloud: Encoding and Interpretation		110
6.1.	Introduction.....	110
6.2.	Andre’s Think Aloud Sessions	111
6.3.	Luke’s Think Aloud Sessions	134
6.4.	Kasper’s Think Aloud Sessions.....	158
6.5.	Matthew’s Think Aloud Sessions.....	179
6.6.	Summary	194
Chapter 7. Theory of Learning and Learning to program		195
7.1.	Introduction.....	195
7.2.	Piaget and Neo-Piagetian Theories.....	195
7.3.	Vygotsky’s Theory and the Notion of Scaffolding.....	201
7.3.1.	Identifying the Zone of Proximal Development (ZPD) of Participants ..	201
7.3.2.	Scaffolding Influence	204
7.3.2.1.	Soft Scaffolding – Assistance by the Researcher	205
7.3.2.2.	Hard Scaffolding – Software Scaffolding – Robot World.....	208
7.3.2.3.	Hard Scaffolding – Software Scaffolding – Unit Test.....	209
7.3.2.4.	Metacognitive Scaffolding.....	210
7.4.	Sfard’s Theory	210
7.5.	Cognitive Load Theory	218
7.6.	Summary.....	227
Chapter 8. Conclusion.....		230
8.1.	Overview of Research.....	230
8.2.	Research Questions.....	230
8.3.	Reflections on the Think Aloud Method	235
8.4.	Validity, Reliability and Generalisability of the Difficulty Framework....	236
8.5.	Trustworthiness of the Think Aloud Data	238

8.6.	Implications for Teaching.....	240
8.7.	Future Research	242
	References	243
	Glossary	259
	Appendix A. Think Aloud Data.....	260
	Appendix B. AUTEK Ethics Approval	295
	Appendix C. Prior Knowledge Questionnaire	302
	Appendix D. The Learning Outcomes of the P1 Course	303
	Appendix E. The Learning Outcomes of the P2 Course	304
	Appendix F. Questions for Developing of Writing Difficulty Metric	305
	Appendix G. Participants Categorisation According to Their Ability to Solve the Programming Tasks	308
	Appendix H. Summary of Think Aloud Recording Sessions.....	309

List of Figures

Figure 3.1 Philosophical perspective of this thesis	37
Figure 4.1 The cognitive process dimension; (left) Bloom's and (right) revised Bloom's taxonomy (adapted from Pohl, 2000, p.8)	53
Figure 4.2 SOLO taxonomy (taken from Hook, 2016, p.1).....	55
Figure 4.3 Regular expression metric calculation and control flow graphs.....	61
Figure 4.4 The GQM paradigm (taken from Basili, Caldiera, & Rombach, 1994, p. 3)	66
Figure 4.5 Example readability metric calculation	69
Figure 4.6 Scree plot for the analysis.....	76
Figure 5.1 The conceptual relationships between the questions	95
Figure 5.2 The order in which the questions are presented to the participants.....	98
Figure 5.3 The scenarios provided for Seq1 – Q1	99
Figure 5.4 Three different scenarios for Seq1 – Q2.....	100
Figure 5.5 Three different scenarios for Seq1 – Q3.....	100
Figure 5.6 Two different scenarios for Seq1 – Q4.....	101
Figure 5.7 The scenarios provided for Seq2 – Q1	102
Figure 5.8 Three different scenarios for Seq2 – Q3.....	104
Figure 5.9 Three scenarios for Seq2 – Q4.....	104
Figure 6.1 Andre's first screen image for the longest corridor	115
Figure 6.2 Andre's second screen image for the longest corridor	116
Figure 6.3 Andre's doodle for the longest corridor.....	117
Figure 6.4 Andre's third and final screen images for the longest corridor	117
Figure 6.5 Andre's first and second screen images for the shortest corridor.....	121
Figure 6.6 Andre's third and fourth screen images for the shortest corridor.....	122
Figure 6.7 Andre's final screen image for the shortest corridor	123
Figure 6.8 Andre's doodle for checking integers in a 1D Array are sorted in descending order	124
Figure 6.9 Andre's screen image for checking integers in a 1D Array are sorted in descending order	125
Figure 6.10 Andre's first and second screen images for the smallest element in a 1D array	126

Figure 6.11 Andre’s third screen image for the smallest element in a 1D array.....	127
Figure 6.12 Andre’s doodle for the small stack of beepers algorithm.....	128
Figure 6.13 Andre doodle to trace the smallest stack of beepers algorithm	128
Figure 6.14 Andre’s fourth screen image for the smallest element in a 1D array	129
Figure 6.15 Andre’s code for find the largest index	130
Figure 6.16 Andre’s first and second screen images for the largest element in a 2D array	131
Figure 6.17 Andre’s last screen image for the largest element in a 2D array	132
Figure 6.18 Andre’s screen image for highest student mark in a collection of student objects	134
Figure 6.19 Luke’s first and second screen images for counting all beepers	135
Figure 6.20 Luke’s third and fourth screen images for counting all beepers.....	136
Figure 6.21: Luke’s doodle for counting all beepers	137
Figure 6.22 Luke’s first screen image for the longest corridor.....	139
Figure 6.23 Luke’s second and third screen images for the longest corridor	140
Figure 6.24 Luke’s fourth and fifth screen images for the longest corridor	141
Figure 6.25 Luke’s sixth and seventh screen images for the longest corridor.....	141
Figure 6.26 Luke’s final screen image for the longest corridor.....	142
Figure 6.27 Luke’s first and second screen images for the smallest stack of beepers ..	144
Figure 6.28: Luke’s third screen image for the smallest stack of beepers	145
Figure 6.29 Luke’s first and second screen images for the shortest corridor	148
Figure 6.30 Trace-table for Luke’s code for the shortest corridor.....	149
Figure 6.31 Luke’s screen image for the smallest element in a 1D array.....	150
Figure 6.32 Luke’s first and second screen images for find the largest index.....	152
Figure 6.33 Luke’s first and second screen images for checking if beepers stacks are sorted.....	153
Figure 6.34 Luke’s third screen image for checking if beepers stacks are sorted	154
Figure 6.35 Luke’s first and second screen images for the largest element in a 2D array	155
Figure 6.36 Luke’s screen image for column in a 2D array which contains a smallest number.....	157

Figure 6.37 Luke’s screen image for highest student mark in a collection of student objects	158
Figure 6.38 Kasper’s first and second screen images for counting the length of one corridor.....	160
Figure 6.39 Kasper’s final screen image for counting the length of one corridor	161
Figure 6.40 Kasper’s first and second screen images for comparing the length of two corridors	163
Figure 6.41 Kasper’s first and second screen images for the longest corridor	166
Figure 6.42 Kasper’s first and second screen images for the smallest stack of beepers	167
Figure 6.43 Kasper’s third and fourth screen images for the smallest stack of beepers	169
Figure 6.44 Trace-table for Kasper’s fourth screen image for the smallest stack of beepers	170
Figure 6.45 Kasper’s final screen image for the smallest stack of beepers	171
Figure 6.46 Kasper’s first and second screen images for the shortest corridor	173
Figure 6.47 Kasper’s third and fourth screen images for the shortest corridor.....	174
Figure 6.48 Kasper’s doodle for the longest corridor algorithm.....	175
Figure 6.49 Kasper’s first and second screen images for the largest element in a 2D array	177
Figure 6.50 Trace-tables for Kasper’s code for the largest element in a 2D array	177
Figure 6.51 Kasper’s first and second screen images for column in a 2D array which contains a smallest number	179
Figure 6.52 Matthew’s first screen image for counting for counting all beepers	180
Figure 6.53 Matthew’s second and third screen images for counting all beepers	181
Figure 6.54 Matthew’s fourth screen image for counting all beepers.....	182
Figure 6.55 Matthew’s doodle for counting all beepers	182
Figure 6.56 Matthew’s first and second screen images for comparing the length of two corridors	184
Figure 6.57 Matthew’s first and second screen images for the longest corridor	185
Figure 6.58 Matthew’s third screen image for the longest corridor.....	187
Figure 6.59 Matthew’s first and second screen images for the smallest stack of beepers	189
Figure 6.60 Matthew’s first doodle to trace the small stack of beepers algorithm	189

Figure 6.61 Matthew's second doodle to trace the small stack of beepers algorithm ..	189
Figure 6.62 Matthew's first and second screen images for the shortest corridor	191
Figure 6.63 Matthew's code to print all the elements of a 1D array.....	193
Figure 6.64 Matthew's first and second screen images for the smallest element in a 1D array.....	193
Figure 6.65 Trace-table for Matthew's code for the smallest element in a 1D array ...	194
Figure 7.1 The relationship between ZPD and scaffolding.....	205

List of Tables

Table 3.1 The transcription template	47
Table 4.1 Software metrics and their applicability across programming paradigms (taken from Kasto & Whalley, 2013, p.60).....	58
Table 4.2 GQM template.....	67
Table 4.3 Metrics for instructor’s model and a percentage difficulty for each question	70
Table 4.4 The correlations between metrics and difficulty	70
Table 4.5 R-matrix	75
Table 4.6 Total variance explained	76
Table 4.7 Factor matrix for the analysis	77
Table 5.1 Main P1 topics	80
Table 5.2 The provided Robot class methods	82
Table 5.3 The workload expectation for the P1 course (taken from the course descriptor)	83
Table 5.4 Main P2 topics	84
Table 5.5 SOLO categories for code reading.....	86
Table 5.6 SOLO categories for code writing	87
Table 5.7 Novel SOLO classification categories for code writing tasks using schemas	88
Table 5.8 Transfer types (taken from Schunk 2012, p.319 Table 7.4)	91
Table 5.9 Overview of the tasks.....	94
Table 5.10 Schemas required for solving the questions.....	96
Table 7.1 The number of soft scaffolding given during the think aloud sessions for the 133 participant solutions	206

List of Abbreviations

Abbreviation	Full Description
AUTEC	Auckland University of Technology Ethics Committee
CLT	Cognitive Load Theory
IDE	Integrated Development Environment
ITiCSE	International Conference on Innovation and Technology in Computer Science Education
KMO	Kaiser-Meyer-Olkin
P1	Programming 1
P2	Programming 2
PAF	Principles Axis Factor
SRES	Software Readability Ease Score
SOLO	Structure of Observed Learning Outcomes
ZPD	Zone of Proximal Development

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Signed: 

Nadia Kasto

Dedication

In the memory of my dear *father*

To my beloved *mother*

To my incredibly wonderful husband, *Barry*, who has been supportive of my work. I love you always and forever.

To the stars of my live, *Zain & Zinah*, I love each one of you more than I can say, and I am lucky and proud to be your mother.

Nadia

Acknowledgements

Thanks God for helping me with the strength and determination to present this work.

I would like to express my sincere gratitude and appreciation to my main supervisor Assoc. Professor Jacqueline Whalley for her support through my Ph.D. journey, I would like sincerely to thank my secondary supervisor Ms. Anne Philpott for her advice. I would also like to thank sincerely to my third supervisor Professor Stephen MacDonell for his advice and valuable suggestions during the work of this thesis. I would like to thank sincerely to Dr. Robert Wellington (AUT) for his support and advice on the research method and the use of video equipment from his research lab. Furthermore, I would like to thank sincerely to my previous supervisor Dr. Jiamou Liu for his support during the first months of my Ph.D.

My special thanks go to David Whalley; although “thanks” seems inadequate for your consistent guidance and support given throughout the writing of my Ph.D.

I would like to express my thanks to the staff and my colleagues in the School of Computer and Mathematical Sciences who supported me throughout my study.

I would also like to thank the students who volunteered to take part in this research. I wish them the best of luck in future.

Thank you to my wonderful friends who gave their support and encouragement.

To my husband, Barry; my children, Zain and Zinah; my mother, Jacqueline Kasto; and my mother-in-law, Janet Awbi, my sincere appreciation for your love and support.

List of Publications

- Kasto, N. (2012). The development of knowledge in novice programmers. In *Proceedings of the 9th International workshop Computing Education Research (ICER'12)* (p. 159). Auckland, New Zealand: ACM.
- Kasto, N., & Whalley, J. (2013). Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the 15th Australasian Computer Education Conference (ACE'13)* (Vol. 136, pp. 59–65). Adelaide, South Australia: Australian Computer Society Inc.
- Kasto, N., Whalley, J., Philpott, A., & Whalley, D. (2014). Solution Spaces. In *Proceedings of the 16th Australasian Computer Education Conference (ACE'14)* (pp. 133–137). Auckland, New Zealand: Australian Computer Society, Inc.
- Whalley, J., & Kasto, N. (2014). How difficult are novice code writing tasks ? A software metrics approach. In *Proceedings of the 16th Australasian Computer Education Conference (ACE'14)* (pp. 30–40). Auckland, New Zealand: Australian Computer Society, Inc.
- Whalley, J., & Kasto, N. (2014). A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation and Technology in Computer Science Education (ITiCSE'14)* (pp. 279–284). Uppsala, Sweden: ACM.
- Awbi, N. K., Whalley, J. L., & Philpott, A. (2015). Scaffolding, the Zone of Proximal Development, and novice programmers. *Journal of Applied Computing and Information Technology (Poster)*, 19(1).

Chapter 1. Introduction

1.1. Background of the Research

Programming is a complex cognitive skill that requires mastering. The cognitive development processes that trigger learning have been a subject of discussion in the computer science education community for a number of years. A wealth of literature points to the fact that learning to program is difficult but we have little understanding as to how students learn to program (Grover, Pea, & Cooper, 2015; Lister et al., 2006; McCracken et al., 2001; Soloway & Spohrer, 1989; Perkins & Martin, 1985; Soloway, Ehrlich, Bonar, & Greenspan, 1983).

Several empirical studies have focused on the difficulties that students have with learning different language concepts, such as; input and output, assignment of values to variables, the role of variables, and iteration-statements (Izu, Weerasinghe, & Pop, 2016; Corney, Teague, Ahadi, & Lister, 2012; Kuittinen & Sajaniemi, 2004; Samurçay, 1989; Spohrer, Soloway, & Pope, 1985; Du Boulay, 1986). Others have focused on the difficulties that novices have when trying to understand object-oriented concepts (Reges, 2006; Lister et al., 2006; Fleury, 2000).

One empirical study found that while novice programmers may know the syntax and semantics of individual statements, they may not recognize how to combine these features into a valid program code (Spohrer & Soloway, 1986), and these have concluded that *“educators may be able to improve their students’ performance by teaching them strategies for putting the pieces of program code together, and by helping them learn the syntactic and semantic constructs of the language”* (p. 632). In other words, the ability to solve a code writing problem requires skills beyond practicing the syntax and semantics of a programming language, and most of the errors in students’ programs are usually related to a lack of organising knowledge and problem solving strategies, i.e. a deficiency in their ability to see internal similarities between problems and to transfer ideas from one problem to a similar problem in a different context (Muller, 2005).

The problems associated with learning to program are well documented by global, multi-institutional studies. The McCracken (2001) working group’s empirical study of novice code writing found that CS1 (first year programming) students were less proficient at programming than anyone, including their teachers, had imagined. This work set the scene for a number of other medium-to-large scale multi-national, multi-institutional studies of novice programmers. A popular explanation for the failure of novice

programmers to reliably write correct and/or high quality code is that students lack the ability to abstract a problem description, decompose it into sub-problems, and reassemble the pieces into a complete solution. One study, which extended the McCracken groups work, focused on code comprehension and found that students also fail to comprehend code, suggesting that such students have a fragile grasp of the skills that are a prerequisite for problem solving (Lister et al., 2004). Whalley et al. (2006), also studied students' ability to comprehend code and extended this line of research. The authors found that students who cannot read a piece of code and describe it in relational terms are not well equipped to write code. Recent works have focused on the relationship between tracing, explaining and writing code (Kumar, 2013; Murphy, Fitzgerald, Lister, & McCauley, 2012; Lister, Fidge, & Teague, 2009; Venables, Tan, & Lister, 2009; Lopez, Whalley, Robbins, & Lister, 2008; Philpott, Robbins, & Whalley, 2007). While most studies point to code writing being more difficult than code reading, studies by Denny, Luxton-Reilly, and Simon (2008), and Yamamoto et al. (2012) found exactly the opposite. Other studies found that there is very little correspondence between ability to write a program and the ability to read one (Winslow, 1996). Lister et al. (2009) also questioned the idea of a skill hierarchy suggesting that the observed hierarchy might actually be a consequence of the difficulty level of the problems the students were given rather than evidence of a hierarchy of skills. Some recent studies have focused on assessing the difficulty levels of code reading and code writing and concluded that one reason for many students lack of success could be the difficulty inherent in the instructional design of the course and/or the difficulty of the programming tasks (Ginat & Menashe, 2015; Whalley et al., 2011; Whalley et al., 2006).

It is generally accepted that learning to program is more difficult than learning other computing subjects at tertiary level (Oliver, Dobele, Greber, & Roberts, 2004). It has been postulated that this is because of the dependency between program concepts; a student must fully understanding one concept before they can even begin to learn a new concept and each programming problem solution can be reused in solving another programming task (Robins, 2010).

Many studies of novice computer programmers, such as the ones noted in this section, have relied on single snapshots of student work from naturally occurring data. While this has become an accepted and valid method of research it has its limitations, as assumptions are generally made about how students learn and about their learning processes. Although these studies have led to some interesting findings, we cannot truly elicit a student's

development of code comprehension and code writing skills from these snapshots. In contrast the study proposed in this thesis, is a longitudinal study that follows tertiary students through their first year of learning to program. The aim is to observe and investigate the nature of student cognitive schemas and the way in which the students adjust those schemas when undertaking code writing tasks for a period of nine months (One academic year).

In this thesis a schema is defined as an existing mental structure stored in long term memory. A schema represents an organisation and linking of knowledge. Programming schema may be composed of salient elements which are defined as small syntactic units

Much of the research in the teaching and learning of programming to date has focused on code comprehension rather than on code writing. There are several reasons why this is the case. It is generally accepted that in order to be able to write code you have to read code (Griffin, 2016; Lopez et al., 2008) therefore many studies have focused on code comprehension as a precursor to code writing. Moreover, measuring a students' code writing ability is harder than measuring code comprehension tasks because of the free-form nature of code writing which makes its interpretation ambiguous (Kumar, 2013). This makes investigating the learning of code writing inherently more complex. However, this does not mean that code writing should not be investigated. Indeed, the lack of research in the area of code writing and the well-documented difficulties of novice programmers suggest that there is a great need for such studies.

1.2. Key Definitions

This section defines the key definitions related to cognitive schemas commonly found in the literature. For the purpose of this research the following working definitions were developed. The term salient element was first used in the literature by Whalley et al.(2011). Salient elements are syntactic elements in novice code. These include FOR-loops, IF-statement or variable declarations. Salient elements are the simple elements which when combined form a schema. Schema are existing mental structures in long term memory. They represent an organisation and linking of knowledge. A plan is a set of steps used to solve a programming task. Typically a plan will consist of more than one schema. A pattern is a recurring schema or plan which is used so often it becomes a generalised or abstract notion which can be applied to different problems. A pattern is more generalised or abstract than a schema.

1.3. Research Questions

The aim of this research is to gain a deeper understanding of the ways in which novice programmers learn to program, with an emphasis on their cognitive development processes.

Research question 1 (Q1): Can we develop a framework that describes the difficulty of novice code writing tasks? The question is refined to more specific questions:

- 1.1. Can we objectively classify the difficulty of the novice programming (code writing) tasks?
- 1.2. Which existing taxonomy best illustrates the observed difficulty of programming tasks?

Research question 2 (Q2): How do novice programmers integrate new programming structure or elements into their current understanding of code? The question is refined to three more specific questions:

- 2.1. Can we identify common patterns (strategies) that students apply when attempting to write a piece of code?
- 2.2. What kind of tasks scaffold and reinforce code writing?
- 2.3. Can we identify the Zone of Proximal Development (ZPD) of a student?

Research question 3 (Q3): Does a student's approach to integrating new knowledge change over time? If it does, what triggers this change?

Research question 4 (Q4): What specific properties does a programming question or task need to trigger a learning event?

Research question 5 (Q5): Can we develop a cognitive framework that describes the ways in which novice programmers integrate new programming structure or elements? The question is refined to two more specific questions:

- 5.1. What existing frameworks, if any, can be integrated or adapted to describe the knowledge acquisition process of novice programmers?
- 5.2. Does any existing learning or cognitive theory (or combination of theories) explain our observations of novice programmers?

1.4. Research Design

The study designed to answer these questions is a longitudinal study that follows a small number of students through their first year of learning to program at Auckland University of Technology (AUT). In this study, a mixed research method is adopted providing both

quantitative and qualitative data including the analysis of student responses to exam questions and interviews and observation of students writing code. A novel framework is designed that combines the ideas of software metrics and the SOLO taxonomy and this will be used to measure the difficulty of programming tasks. This framework will subsequently be used to design a set of tasks to trigger situations that require some form of knowledge adaptation or acquisition. Once suitable programming tasks have been identified participants will be observed individually while they are attempting to solve these tasks. Data will be collected using think aloud protocols as well as direct observations. At the conclusion of each session each participant will take part in a retrospective interview about the way in which they attempted to construct a program that performed the task set and about problems they encountered and what they did to try to overcome those problems.

1.5. Structure of the Thesis

This thesis is organised into eight chapters, and structured as follows:

Chapter 1 introduces the thesis topic and outlines the research aims and design.

Chapter 2 contains the literature review, which surveys the theories of cognitive development proposed by Piaget and Neo-Piagetian theorists, Vygotsky, Sfard's theory of concept development, and Cognitive Load Theory (CLT). It also explores the application of these theories in the context of learning in general and learning to program in particular. For this research, a clear understanding of these theories is key to developing a broader understanding of the way in which learning occurs. This chapter also investigates the strategies that novices use to comprehend and generate/write a program. The chapter concludes with a review of the literature relating to the transfer of learning and analogy in cognition.

Chapter 3 presents a detailed overview of methodological principles and a discussion of the philosophical perspective for this thesis, including the research instrument design and procedures adopted for data collection and sampling. This chapter also provides a description of the think aloud method, the stages of verbal protocol analysis, and the intervention model.

Chapter 4 presents detail about the design of a framework for describing programming tasks and their difficulties. It discusses the design of a framework that combines the ideas of the SOLO taxonomy and software metrics and reports on an empirical evaluation of

students' responses to past code writing tasks and an analysis the metrics usefulness as predictors of task difficulty.

Chapter 5 focuses on the design of a set of programming tasks (the research instrument) within the difficulty framework described in Chapter 4. This chapter also addresses pedagogical approaches, development tools and content of the programming courses (Programming 1 (P1) and Programming 2 (P2)) in which this study is situated.

Chapter 6 details the think aloud transcriptions, encoding and provides a preliminary analysis for the four key participants in this study (top participants and bottom participants), which are further explored used in Chapter 7. The remaining data set from the four participants has been included in Appendix A.

Chapter 7 discusses the common patterns of learning which were extracted from think aloud data (Chapter 6) with reference to the cognitive theories of learning (Chapter 2).

Chapter 8 concludes the thesis. It identifies the limitations of the study, draws conclusions from the findings and gives suggestions for further research into the learning and teaching of computer programming to novice programmers.

Chapter 2. Background

2.1. Introduction

The literature reviewed in this chapter covers the following major themes:

- General theories of cognitive development, application of these theories in the context of learning in general, and in particularly learning to program.
- Knowledge organisation and strategies for programming.
- Transfer in cognition.

The theories discussed in the first section have all made significant contributions to educational psychology, and learning theory. They are general theories of cognitive development. For this research, a clear understanding of these theories is a key to developing a broader understanding of the way in which learning occurs. This is followed by a section dealing with knowledge organization and problem solving strategies. Studies of strategies used by programmers typically focus on what is happening in the mind of the novice programmers when attempting to solve a programming task. Finally, this chapter concludes with a literature relating to the transfer of learning and analogy in cognition.

2.2. General Theories of Cognitive Development and Learning

Cognitive development theories depend on the premise that learning is based on previous experiences and existing perspectives which influence the way new information is interpreted. Individuals engage in a learning activity by integrating that new information into their existing schema, building knowledge and skills based on prior knowledge and experience rather than just passively absorbing what is presented to them.

Recently, attention has turned to looking at these theories as a way of trying to understand how adult learners begin to learn programming. This research requires methods and theories that help to observe and explain the process of learning and development, and help to identify why students are having difficulty in learning to program. Piaget developed a theory of cognitive development which is widely regarded as providing the foundation on which later constructivist theories have been developed. Despite the criticisms that have been made about his theory his views remain well regarded and include aspects that inform both the design and analysis of this research. Moreover, those later theories of learning which have their foundations in Piagetian theory should also assist this research- for example, Vygotsky's notion of a Zone of Proximal Development (ZPD), Sfard's ideas on the process of abstraction from concrete examples to abstract

concepts, and CLT studies on how to reduce the load on working memory to optimise learning. Each of the above theories brings with it basic assumptions about the nature of learning, the phenomena of interest, and the types of explanations that can be generated. However, while multiple theories have been applied separately to computer science education, there are still no very compelling answers to the question, “Why do so many students struggle to learn to program?” This research differs from previous studies, in computer science education, in that it starts with data about students engaged in learning to program and then uses a number of theories of learning to explain our observations of novice programmers and identify why students are having problems.

This section consists of a review of each theory and how each theory has been applied in the context of learning to program. A key element to look at when examining the above theories is what they have to say about how learning occurs and about the effects of aspects of the learning environment on learning. Factors such as students’ prior knowledge, how students organize knowledge (deep and fragile knowledge), social interaction and scaffolding, practice, time zone for learning, and self-regulation of learning could provide useful insights into how students learn to program.

2.2.1. Piaget and Neo-Piagetian Theories

One of the most influential learning theories to date was developed by Jean Piaget (Hsued, 2005). Piaget’s theory was formed from a *constructivist perspective*, which sees people construct their own knowledge and understanding of the world by discovery (Schunk, 2012).

Piaget believed that individuals learn to interpret the world through building mental knowledge (schemas) and that it is only when these schemas change qualitatively that the process of cognitive development occurs. Piaget’s work focused on the individuals rather than on any sociocultural influences on cognitive development.

Piaget theorized that there are two main processes that bring about learning; *organisation* and *adaptation*. Organisation defines how existing knowledge or experiences are related. Organisation is the result of practice over a long period of time. Piaget believed that this organisation of information made the human thinking process more efficient. Adaptation, on the other hand, is defined as the process by which humans match existing knowledge with a new experience which may not fit within their existing knowledge readily. Piaget argued that schemas could change through the processes of adaptation: *assimilation* and *accommodation*. Assimilation is a process of incorporating new information into one's existing mental structure (schema) for future retrieval and use (Flavell, 1977).

Accommodation is a process through which one changes their existing mental structures (schemas) in order to accommodate new information (Flavell, 1977). Adaptation becomes necessary when disequilibrium exists, that is when the individual's beliefs (existing schema) do not match their observed reality. The processes of adaptation are used to restore equilibrium.

In his theory, Piaget identified four stages of intellectual development (Piaget & Inhelder, 1969). These four stages are: *sensorimotor*, *preoperational*, *concrete operational* and *formal operational*. Piaget argued that intelligence developed progressively as the individual moved through these stages.

Piaget, in his theory of childhood development, used the term "d calage" to represent the idea that analogous cognitive structures or processes appear at different moments of development rather than being synchronous. Vertical d calage describes the process of carrying out the same task with increasingly sophisticated conceptual approaches whereas horizontal d calage describes the time lag in achieving different tasks that require the same cognitive structures (Flavell & Piaget, 1963). He described intellectual development as an upward expanding spiral in which children must constantly reconstruct the ideas formed at earlier levels with new, higher order concepts acquired at the next level (Piaget & Inhelder, 1969; Hsued, 2005).

At about age six (primary school age), the child enters the concrete operational stage and is able to apply operations to real objects and events. The cognitive abilities to solve problems involving physical objects that require: seriation, classification, reversibility, conservation, decentring, and/or transitivity begin to develop. This represents a fundamental change in the child's development because it is the beginning of operational (i.e. rule based) or logical thought.

By the time adolescents reach the formal operational level of reasoning, they can organise information, reach logical conclusions and form hypotheses (Huber, 1988). They develop the ability to think in the abstract and can manipulate multiple variables systematically. Another ability at the formal operational stage is that of metacognition and self-regulation that entails "*reflecting on, monitoring and management of one's thoughts*" (Kuhn, 2008).

Piaget's theory has been criticized largely because he based his theory on the observations of a small set of children (and therefore his findings may not be generalizable), including his own three children, from similar sociocultural backgrounds, and so does not acknowledge the influence of cultural factors on learning and development.

The neo-Piagetian theorists have claimed that Piaget underestimated the thinking capabilities of preschool children. They have tended to place more emphasis on the influence of cultural experiences on a child's cognitive development. They have often adopted the view that people, regardless of their age, progress through increasingly abstract forms of reasoning as they gain expertise in a specific problem domain, and have attempted to modify Piaget's stages by postulating additional levels of adult reasoning (Commons, Richards, & Armon, 1984). Since there is little agreement about the post formal operational levels that have been proposed and since these levels do not required the use of a form of logical reasoning that is fundamentally different from that acquired in the formal operational stage of development, levels beyond the formal stage have not been discussed here. Similarly, because the subjects of this study were all in late adolescence or early adulthood the validity of the criticism regarding the age of onset of concrete operational thought is not relevant and so has not been critiqued in this thesis.

Although Piaget's theory largely meets three of the criteria usually applied to the concept of a developmental stage, i.e. qualitative change, ordinality and organization, the theory does not meet two of the criteria. A change between stages is expected to meet the criterion of abruptness whereas the change from one Piagetian stage to another is normally very gradual. A stage is also expected to display concurrence, i.e. there should be more or less simultaneous and similar changes across domains; but Piagetian theory fails on this criterion. The emergence of the concrete operational process of conservation, for example, often takes several years to fully develop, as the child's conservation of liquid, number, length, weight, etc. becomes established over a number of years.

Piaget and the neo-Piagetians have managed information about inconsistency in the stage of reasoning used by individuals across different domains of knowledge differently. Piaget recognised the lack of concurrence and added the notion of *décalage* to his theory, but this is essentially a term that describes rather than explains the lack of concurrence. Some neo-Piagetian theorists have suggested that it would be better to refer to levels or modes of logical reasoning rather than stages and that an individual could then use different modes in different domains of knowledge and this would not represent a failure to meet the criterion of concurrence across all areas of reasoning which is applied to a developmental stage theory (Ojose, 2008). Some neo-Piagetians have suggested that processing and working memory capabilities may explain transitions in thinking level and that differences in memory demands may explain the fact that different levels of reasoning

are observed in the same person solving problems in different domains of knowledge (Strauss, 1993).

Neither Piaget nor the neo-Piagetians have defined a set of knowledge domains within which concurrence could be expected. For example, is mathematics a domain or does it consist of several domains: algebra, geometry, trigonometry etc.? And is geometry a domain or should it be separated into smaller domains, e.g. Topography and Euclidean geometry? If moral reasoning is accepted as being a domain then we have to face the difficulty that young adults who reason about some moral problems by using formal operational logic revert to less sophisticated stages of reasoning when responding to other similar moral problems. Thus the lack of observed concurrence, even within what appears to be a domain of knowledge, remains an unresolved difficulty for both Piagetian and neo-Piagetian theorists and researchers.

A number of researchers have attempted to test the effectiveness of level of Piagetian stage of development as a predictor for success in learning computer programming. Since programming requires the ability to think in the abstract and to apply logic, the level of Piagetian stage used to solve problems appears to be a strong candidate.

Kurtz (1980) reported a strong correlation of 0.63 between a test of Piagetian reasoning problems and course grades but Barker and Unger (1983), using most of the same set of Piagetian tasks and with a far greater number of students could not confirm the relationship reported by Kurtz. Werth (1986) duplicated Kurtz's study with a small number of students and was also unable to find a relationship. Bennedsen and Caspersen (2008), and Cafolla (1988) have also reported that measures of formal operational reasoning based on student responses to Piagetian problems did not correlate strongly with student grades for programming. Two studies that did report some predictive ability separated the subjects into dichotomous categories of success based on course grades, rather than using the raw grades. Fischer (1986) used a criterion of B+ or above, and Hudak and Anderson (1990) used a criterion of 72%+, to place students into the successful category. Fischer reported that 91% of the students in the successful category were at the formal operational stage but none of those from the unsuccessful category were at that stage. Hudak and Anderson reported that they had been able to correctly predict the programming course successful/unsuccessful status of 72% of the students from the results of a test of formal operational reasoning. In a similar vein, White and Sivitanides explained the bimodal distribution of grades commonly reported for introductory programming classes using Piaget's development levels "*The low mode may*

indicate Piaget's concrete operation stage. The high mode may indicate Piaget's formal operation stage” (2002, p.10).

The lack of agreement from the findings of the various pieces of research may be a consequence of inadequacies in the measure of success in programming or due to the concurrence issue described above. The measures of success in learning to program have all been taken from formal assessments that were an integral part of the tertiary courses in which the subjects were enrolled. However, a number of other studies have found that the grades awarded students in programming courses frequently have deficiencies of validity and/or reliability and are therefore probably not adequate as measures of the ability to write computer programs. The other assumption made by the researchers is that a measure of the stage of reasoning used to solve Piagetian problems that are located in one domain of knowledge, e.g. physics or mathematics, would be an adequate predictor of the level of reasoning used in another domain, i.e. computer programming. The lack of concurrence in Piagetian stages makes this most improbable. Moreover, the age of most students in the early years of their studies falls within the 15 to 20 year range, the time given by Piaget for the development of formal operations. It should therefore be expected that many of them would not have a firm grasp of formal operations and many may not even have begun to engage in formal operational reasoning.

It must be concluded that it has yet to be demonstrated that tests of Piagetian reasoning are useful for predicting success in learning to program and therefore useful as a student selection tool. However, with greater attention to the issue of concurrence and to the transition in mode of thinking that many tertiary students are likely to be undergoing, and attention to the accuracy of the assessment of student's programming competencies, it may at least be possible to develop a measure of student reasoning that can identify those students unlikely to succeed in the traditional, large class, novice programming course. Alternatively, of course, it could be argued that the real problem is not one of predicting success but of providing novice programmers with courses that are more appropriately structured and provide better learning opportunities. Unfortunately many university level introductory programming courses are lecture based and have yet to provide for the sort of learning advised by Piaget 45 years ago: *“You cannot teach concepts verbally; you must use a method founded on activity”* (Hall, 1970, p.30).

To date there have been very few empirical studies of the cognitive development of novice programmers. However, in the last decade some Australasian researchers have

attempted to reinterpret aspects of Piagetian theory and apply these reinterpretations to an empirical study of the development of the skills of novice computer programmers.

Lister (2011) suggested that Piaget's developmental stages could be used to establish the cognitive development levels of novice programmers. His hypothesis arose as a result of earlier empirical studies of novice programmers that found novices need to be able to trace with >50% accuracy before they can begin to understand the code (Philpott, Robbins, & Whalley, 2007; Lister, Fidge, & Teague, 2009; and Venables, Tan, & Lister, 2009).

One of the limitations of Lister's work is that it focused on the skill required to solve a code comprehension problem to establish the neo-Piagetian level of the student's cognition. He did not explicitly explore subjectively or objectively the difficulty of each problem. This means that it could be argued that some observations are an artefact of a specific problem which may have contained unintended complexity or non-domain specific content. A follow up empirical studies by Teague & Lister (2014b; 2014a; 2014c) attempted to classify students as sensorimotor, preoperational, or concrete operational thinkers. Other researchers in computer education have not included the sensorimotor stage¹ because this stage consists of behaviours below what would normally be expected by adult learners when learning in new cognitive domains, especially when learning programming which requires higher cognitive abilities (Falkner, Vivian, & Falkner, 2013; and White & Sivitanides, 2002).

Another limitation of Lister and Teague's work is that the validity of the conclusions reached are dependent on the accuracy with which he was able to match the logical thought required to complete programming tasks to Piaget's developmental stages. For example, Lister proposed that *"in a programming context, a novice at the concrete operational stage should be able to easily make minor changes to code while conserving what the code achieves"* (Teague & Lister, 2014a, p.31). The implication of this supposition is that if the student does not maintain the intended output of the code but makes changes that would produce a different output then the student would be functioning at a level below concrete operations i.e. at the preoperational level. The

¹ According to Piaget, during the sensorimotor stage, infants learn to interact with the world around them, which means that an infant could easily use a mouse and randomly press the keyboard buttons. Li & Atkins (2004) found that preschool children of 3-5 years old are able to use a computer by pressing a mouse or button to trigger a visual response but they are certainly not at the level to begin to learn to program a computer. It is highly unlikely that tertiary students (late adolescent (17-19) and adult learners) engage in reasoning below the concrete operational level even when faced with new programming tasks.

difficulty here lies with use of “conservation” to describe the maintenance of “what the code achieves”. When Piaget developed his theory, conservation was defined as the ability to see that some properties are conserved (don’t vary) after an *object* undergoes a *physical transformation*. Lister is not talking about an object but about an abstraction (what the code achieves) and minor changes in a piece of code are very different from the sort of physical transformation Piaget referred to. Therefore, the inability of some university level novice programmers to see that a minor change in code has altered what the code achieves is not evidence that they have been using Piaget’s preoperational thinking.

Learners, according to Piaget, are active constructors of their world and discoverers of knowledge. In contrast Vygotsky’s social constructivism, while incorporating Piaget’s ideas of active learners, emphasises social interaction in learning and development.

2.2.2. Vygotsky’s Theory and the Notion of Scaffolding

In formulating his theory of learning, Vygotsky focused on the *sociocultural contexts* that influence learning (Vygotsky, 1978). Within his sociocultural theory, it is argued that development occurs twice: firstly in the process of social interaction and secondly within the mind of the individual (previous experiences and existing perspectives).

Vygotsky (1978) believed that social interaction with cultural tools represents the most important part of a learner’s psychological development. Cultural tools include “*all the things we use, from simple things such as a pen, spoon, or table, to the more complex things such as language, traditions, beliefs, arts, or science*” (Cole, 1997). In computer science education, the language and/or environment itself is the cultural tool to understanding the programming concepts. Hence the language and/or environment is the vehicle by which programming concepts are usually presented to a novice programmer.

Vygotsky categorised learning into three different levels:

1. What a learner can do *independently* (i.e. on their own). This stage was referred to as *the level of actual development*. It involves skills that a learner has already learned and with which they can work independently.
2. What a learner can do with the assistance of someone. This stage was referred to as *the level of potential development*.
3. What is beyond the learner’s reach even if assisted by someone else.

According to Vygotsky, the potential for cognitive development depends on the Zone of Proximal Development (ZPD). Vygotsky defines ZPD as the “*the distance between the*

actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers” (Vygotsky, 1978, p. 86). In other words, the ZPD gives an indication about what a learner can expect to achieve in the near future. Vygotsky believed that when a learner is at the ZPD for a particular task, providing the appropriate assistance by *more knowledgeable others* will give the learner enough of a boost to achieve the task and make progress. Otherwise, the learners become frustrated and cannot make progress because they have been left for too long at a point where they could not easily make progress. As a result, they can lose motivation and interest (Black, 2006). Vygotsky predicted that teaching input would be most effective if it occurred at the edge of the ZPD and that instruction located at or below a learner’s current level of understanding would not be challenging enough to prompt further development; at the same time, instructions that are beyond what a learner can perform is ineffective for stimulating learning. He postulated that instruction should therefore be targeted somewhere in between in order to enable learners to build on current knowledge. With constant practice supported by *more knowledgeable others*, a learners understanding can continue to improve. *“In order to understand that after repetition it is easier to remember, one must be experienced in memory tasks”* (Vygotsky, 1981, p.181).

For Vygotsky, metacognition and self-regulation are not achieved until adolescence, and require the exposure to scientific concepts provided by school instruction. Exposure to school tasks and the repeated practice they provide promotes the development of metacognitive knowledge about one’s own thinking. *“School instruction induces the generalizing kind of perception and thus plays a decisive role in making the child conscious of his own mental processes. Scientific concepts, with their hierarchical system of interrelation, seem to be the medium within which awareness and mastery first develop, to be transferred later to other concepts and other areas of thought”* (Vygotsky, 1986, p.171).

To date there has been no empirical research reported in the literature which investigates the relevance of Vygotsky’s theory of the ZPD to the learning of computer programming. The closest is the research of Robbins (2010), which discusses the idea of a “learning edge momentum” where the notion of the *learning edge* appears to have its foundation in Vygotsky’s idea of a ZPD. In Vygotsky’s theory as a student learns their ZPD expands, and the most meaningful learning occurs *“only just or very nearly within the range of the child’s independent ability. Rogoff (1984) called this the child’s cutting edge of*

understanding” (Mcnaughton & Leyland, 1990, p.154). This idea is similar to that of Robbin’s *learning edge*.

The bodies of work in computer science education which link to Vygotsky’s theory are in the areas of collaborative learning, software tools, cognitive apprenticeship, and e-learning. The research which has its roots in aspects of Vygotsky’s theory and is most relevant to the research reported here is the work related to *scaffolding*.

The ZPD metaphor has over time become synonymous in literature with the term *scaffolding*. However, Vygotsky did not use this term in his writing. Scaffolding was first used by Wood, Bruner and Ross, who defined it as a “*process that enables a child or novice to solve a problem, carry out a task or achieve a goal which would be beyond his unassisted efforts*” (Wood, Bruner, & Ross, 1976, p.89). In any case, the term “*Vygotsky scaffolding*” has been used by researchers to describe a teaching approach that provides resources such as tools, strategies and guides that support the learners as they learn new concepts in order to provide learners with a higher level of understanding than they could have attained through independent study. An increasing number of educators and researchers have used the concept of scaffolding as a metaphor to describe and explain the role of Vygotsky’s *more knowledgeable others* in guiding learning and development.

The dual aspects of ZPD and scaffolding help the learner to finish the task and improve the learner’s performance. However, sometimes the learners are assisted in the task but are not able to take advantage of the experience; therefore such kinds of assistance will be localised to that instance of scaffolding, and they will not provide support for learning in the future. Thus, scaffolding must cover a delicate cooperation between giving support and continuing to engage the learner actively in the learning process (Reiser, 2002).

Hannafine et al. (1999) identified four types of scaffolding:-

1. Conceptual (supportive) scaffolding: - Guiding the learner in what to consider when the problem is defined.
2. Metacognitive (reflective) scaffolding: - Guiding the learner in such a way that they are encouraged to reflect on the way in which they are learning and to look inward in order to examine what learning strategies are effective for them.
3. Procedural scaffolding: - Redirecting learners to use resources and tools.
4. Strategic scaffolding: - Guidance about alternative approaches or methods to problem solving that might help overcome the given problem.

Saye and Brush (2002) continued this line of research suggesting that there are actually two types of scaffolds namely *soft scaffolds* and *hard scaffolds*. Soft scaffolds “*are dynamic, situation-specific aid provided by a teacher or peer*”. This type of assistance is generally provided “*on-the-fly*” when the teacher observes students and provides immediate support (i.e. formative feedback) based on student responses. In contrast to soft scaffolds, hard scaffolds “*are static supports that can be anticipated and planned in advance based on typical student difficulties with a task*”. Software scaffolds fall into this category. As part of this research students will be given programming problems to solve and it might be useful to use the notions of ZPD, and a framework of different types of scaffolds students receive in order to understand their learning processes.

In the 1980’s, Soloway and associated researchers started to investigate the nature of the development of expertise in computer programming. They discovered that experts have strategies/plans (schemas) for solving computer programming problems. As a result they advocated teaching strategies and plans explicitly as scaffolds to assist novices in constructing code (Letovsky & Soloway, 1986; Soloway, 1986; Spohrer, Soloway, & Pope, 1985).

While there has been a vast amount of discussion in the literature on the teaching and learning of programming, which advocates the use of scaffolding to enhance student learning there have been very few empirical studies which provide explicit evidence that scaffolding supports the learning of computer programming. Arguably one of the most influential pieces of empirical research on the influence of *more knowledgeable others* on the learning of computer programming was that of Perkins and Martin (1985). In this study they investigated the influence of supportive scaffolding provided by instructors. They reported that they were able to extend a student’s knowledge through the use of such scaffolding, “*In particular, prompts led to a correct resolution of difficulties 32% of the time and hints an additional 17%, leaving 52% of difficulties requiring an answer provided by the experimenter*” (Perkins & Martin, 1985, p.32).

In computer science education, the term scaffolding has over time become synonymous with forms of feedback (e.g. feedback from unit tests) and/or tools designed and used for supporting learning (i.e. programming languages (e.g. Scratch) and development environments (e.g. Alice)). As a consequence “*the concept of scaffolding has been more commonly employed to describe what features of computer tools and the processes employing them are doing for learning*” (Pea, 2013, p. 429). This means that most of the reported research uses the notion of scaffolding to explain the way in which a tool is

designed or should be “*pedagogically*” used but does not further that work to an in depth investigation of the role of scaffolding in the development of skills and knowledge of novice programmers. It is possible that *scaffolding* could be considered as a subcategory of the broader class of feedback. Feedback must be presented in a certain way in order for it to scaffold learning. Some tools have been designed to scaffold learning by providing timely and useful feedback and may thereby be scaffolding learning for some students but there is no evidence in the literature that scaffolding is occurring. It should also be noted that scaffolding with tools and software does not originate from Vygotsky’s theory. Vygotskian “*scaffolding*” differs in that it is a type of cognitive apprenticeship where the learner progresses with the assistance of *more knowledgeable others*.

It is likely to be important for this research that the idea of scaffolding and the nature of scaffolding given to students is considered when designing both the research method and as a dimension of the analysis of the results.

2.2.3. Sfard’s Theory

Anna Sfard (Sfard, 1991) explored the ability to abstract from concrete examples to a generalized mathematical concept. Her work explored the dual nature of object and process in the context of mathematical concept development and described two metaphors *acquisition* (a constructivist cognition metaphor) and *participation* (a sociocultural metaphor). *Participation*, in part, involves interacting with *more knowledgeable others* to construct understanding, one of the key ideas in Vygotsky’s theory.

Acquisition involves the accumulation of a set of facts or elements of knowledge, either by reception or cognition through construction, that are abstract (Sfard, 1998). Sfard argued that the process of *acquiring* a mathematical concept involves transitioning from operational conceptions (processes, dynamic sequential and detailed) to abstract objects (static structures). She identified a framework of three phases for concept formation from process to object understanding namely, *interiorization*, *condensation* and *reification*. These three phases are reflective of Piaget’s adaptation theory in which cognitive structures are changed and reified. Understanding mathematical concepts through a set of phases leading to the abstraction is similar to Piaget’s notation of “reflective abstraction” in which actions on existing schema become interiorized, as the individual processes towards a state of equilibrium, and are then *encapsulated* (reified) as mental objects of thought.

*“At the stage of interiorization a learner gets acquainted with the processes which will eventually give rise to a new concept... These processes are operations performed on lower-level mathematical objects. Gradually, the learner becomes skilled at performing these processes. The term “interiorization” is used here in much the same sense which was given to it by Piaget (1970,p.14): we would say that a process has been interiorized if it “can be carried out through [mental] representations”, and in order to be considered, analyzed and compared it needs no longer to be actually performed.”(Sfard, 1991, p.18). At the condensation stage a learner “becomes more and more capable of thinking about a given process as a whole” (Sfard, 1991, p.19). The phase of condensation is chunking sequences of operations into smaller more manageable units. “This is the point at which a new concept is officially born” (Sfard, 1991, p.19). Cognitive development at the condensation stage manifests in an ability to readily alternate between different representations of a concept. “The condensation phase lasts as long as a new entity remains tightly connected to a certain process. Only when a person becomes capable of conceiving the notion as a fully-fledged object, we shall say that the concept has been reified. ... Reification, therefore, is defined as an ontological shift a sudden ability to see something familiar in a totally new light” (Sfard, 1991, p.19). With repeated practice, a shift from the operational to structural approach can be made; “the ... [mathematical]... computational processes were caught into a static construct just like water is frozen into a piece of ice” (Sfard 1991, p.25). Once a concept reaches reification, it can then be used as a primitive object in higher-level concept acquisition. These steps describe “the transition from computational operations to abstract objects” and this is in essence the process of *abstraction*. Within these processes there is an integral assumption that knowledge is an organized hierarchy of concepts that build on each other. Higher abstract concepts require prior knowledge of lower level concepts (deep knowledge). A reasonable extension is the conclusion that this prior knowledge must be within the learners ZPD in order for higher level concepts to be learned.*

Because Sfard focused on the development of relatively advanced mathematical thinking (Pegg & Tall, 2002) her emphasis is on Piaget’s formal development level rather on earlier forms of thinking such as preoperational or concrete operational. The learners’ characteristics are very likely to have an effect on how successfully they transition from the first phase of operational conception to abstract objects. The relevance of Sfard’s work for research on metacognition and self-regulation has been noted by Caswell and Nisbet (2005).

Sfard's framework while developed as a theory for explaining concept development in mathematics is also relevant to learning computer programming. Like mathematics, programming involves "*tightly integrated concepts*". Robins (2010) argued that novice programmers must fully understand one concept before they can even begin to learn a new concept. "*It is very difficult to describe or understand one concept/language element (such as a for loop) independent of describing or understanding many others (flow of control, statements, conditions, Boolean expressions, values, operators), which themselves involve many other concepts, and so on*" (Robins, 2010, p.26).

Lister et al. (2009) agree that skills in computer programming are analogous or comparable to mathematical procedures because "*they represent following step-by-step instructions using the operations of the respective subject area*" (p.160). However, they also argue that there is a fundamental difference between the two subject areas because computer science has both practical (skill based) and conceptual learning goals. They suggest that in computer science skills are goals themselves and not merely intermediary to reaching a more sophisticated understanding. Lister et al. (2009) supposed that there is a major distinction between abstraction in mathematics education and computing education research. In mathematics abstraction is related to *information neglect* in which learners strategically ignore or discard key concepts in order to focus on the concept at hand (Colburn & Shute, 2007). Conversely, in computer science, abstraction is related to *information hiding*. Concepts are encapsulated (i.e. generalised, avoiding contextual specificity) in order to deal effectively with invariants and create a foundation for the next level of thinking.

Despite this difference Lister et al. (2009) claim that it is possible and fruitful to relate the mathematics research findings to research related to the development of skills and concepts in novice programmers. Lister et al.(2009) also suggested that there is a direct and clear relationship between Sfard's theory, variations on Sfard's theory (Dubinsky, 1991; Gray & Tall, 2007), and the SOLO educational taxonomy (Biggs & Collis, 1982). Dubinsky described the cycle of abstraction in terms of Action, Process, and Object Schema (APOS theory). In APOS actions are said to be interiorised as processes and then conceived as objects within a wider schema. This cycle of mental abstraction has also been described in terms of procedure, process and procept where procept is a symbol which can operate dually as a process or a concept (Gray & Tall, 2007). While there are some differences in the detail of these theories, and Sfard's theory they are fundamentally the same.

In an earlier study, Eckerdal and Berglund (2005) undertook a phenomenographic analysis of interviews of first year students about their understanding of what learning to program means. The researchers found that many students talked about learning to program in terms of having to learn a new way of thinking which is different from other subjects they have studied. Eckerdal and Berglund compared their results with research on the “process-object duality”, which is central to Sfard’s concept development theory, developed in mathematics education. They hypothesized that it is of great importance that students reach an understanding that learning to program is a “method” of thinking which corresponds to “procedure conception”. They also suggest that “*such a conception scaffolds for the more abstract level of understanding, the object conception*” (p. 141).

The research reported to date which has used Sfard’s theory to explain aspects of learning to program has relied on theoretical conjecture. No study has been undertaken which empirically attempts to observe or capture the three processes of interiorization, condensation and reification in learning to write computer programs or code.

2.2.4. Cognitive Load Theory

An alternative perspective to the constructivist cognitive theories of learning is provided by Cognitive Load Theory (CLT). CLT is founded on an understanding of human cognitive architecture (Moreno & Park, 2010) and on the need for instructional design principles that are based on knowledge of the brain and how memory works.

Sweller (1994) described CLT as an information processing model of cognition with key learning activities including schema acquisition and automation of their usage. Some ideas in CLT, despite the different origin, hark back to Piaget; for example the notions of schema acquisition and automation have similarities with Piaget’s notions of *organisation* and *adaptation*. Many neo-Piagetians added ideas from CLT to their theories in order to explain observations they had made which could not otherwise be explained by their theories. CLT has proven to be useful in explaining why some learners cannot progress or have difficulty with certain aspects of learning. It provides a potential source of explanation for why certain programming tasks might be more difficult than others for novice programmers. Additionally because CLT’s primary focus is in reducing the cognitive load by improving instructional design, CLT should be useful and relevant to the design of the research instrument used in this research. What follows is a description of the most relevant aspects of CLT and research in the computing education domain which incorporates aspects of CLT.

CLT stemmed from the idea that the working memory is limited during problem solving (Miller, 1956). If the mental processing capacity of these limited resources is “overloaded” at any point during the learning process, then the learning process will be jeopardised. CLT considers the load on working memory in three dimensions – *intrinsic*, *extraneous* and *germane* cognitive load.

Certain working memory load is imposed by the basic structure of the information that a student needs to gain in order to learn. This is known as *intrinsic cognitive load*. This type of load is related to the difficulty of knowledge elements (concepts or schemas) and the degree of interactivity between those elements, and is dependent on existing cognitive schemas (Sweller, 1994). In CLT, the term *element interactivity* is used to refer to the degree of connectedness between knowledge elements. The higher the connectedness, the higher the load is on the working memory. Information that is not connected (i.e. unrelated facts) and therefore can be assimilated serially imposes a relatively low intrinsic cognitive load (Sweller & Chandler, 1994). On the other hand high element interactivity requires the learner to simultaneously process several elements at a time (Kester, Paas, & Van Merriënboer, 2010). Programming involves high element interactivity because it involves many tightly integrated concepts. As a consequence it is reasonable to assume that learning to write computer programs, especially when it involves the use of new concepts or requires the adaptation or formation of new schemas, has a high intrinsic cognitive load.

Extraneous cognitive load is related to the way in which instructional content is presented to the learner. In inefficient instructional designs it adds unnecessary load and therefore interferes with learning by overloading the working memory (Chandler & Sweller, 1991). The programming language and the development environment are the tools with which programming concepts are usually presented to a novice programmer. The concepts are, in a typical university course, presented via formal lectures and practical programming laboratories. The choice of computer programming language and development environment may have a direct impact on students’ learning (Yousoof & Sapiyan, 2015; Mason & Cooper, 2013; Ambrose, Bridges, Dipietro, Lovett, & Norman, 2010). Many researchers have advocated using simpler development environments (for example Alice (Carnegie Mellon University, 2006) and BlueJ (Kölling & Rosenberg, 2001)) or simplified programming languages (for example Scratch (Lifelong Kindergarten Group, 2007)) in order to make learning to program easier by reducing the cognitive load associated with the instructional tools. The language and development environment of

instruction is therefore *extraneous* to understanding these core programming concepts. In this research some sources of extraneous cognitive load will be outside the control of the researcher. For example, because students are studying a tertiary first year introductory programming course the mode of delivery and instructional tools are set by the courses lead instructor. Other aspects of extraneous cognitive load will be in the control of the researcher and include the tools used to gather data (e.g. smart-pen) and the programming problems (research instrument), which are presented to the participants for them to solve. Additionally, the methodology used to observe and interview the participants may impose extraneous cognitive load and it is important that steps are taken to minimise this. One possible risk is that the cognitive load imposed by the research data gathering method imposes so much extraneous cognitive load that it interferes with the participants' ability to write answers to the programming problems present to them.

It is generally accepted that learning computer programming is intrinsically and extraneously difficult (Sweller & Chandler, 1994) because students have much to learn in the first programming course including: new editor software, debugging tools, testing techniques, and interdependent concepts such as language syntax, logical sequence steps, variables, selection, iteration, etc. (Black, 2006).

Germane cognitive load is the degree of mental effort that is applied to schema acquisition, i.e. to schema construction and automation (Paas & Van Merriënboer, 1994). These activities generally consist of comparing and contrasting existing mental schemas and newly presented information in conjunction with some form of practice in order to initiate schema development (Cooper, Tindall-Ford, Chandler, & Sweller, 2001). Germane cognitive load is sometimes linked to the learner's degree of motivation and level of interest in the material being learnt. German cognitive load may be of interest when examining the cognitive processes of the participants in this research. It might be useful to attempt to measure, possibly indirectly, the degree of effort made by the learner. Measures such as time on task and recording aspects of their motivation while solving tasks, especially when they encounter a barrier to their learning, might provide some useful insights into the processes involved in learning to program.

In learning mode, new information from the environment is processed in working memory to form knowledge structures enabling this knowledge to be stored in long term memory. This process is known as schema acquisition and includes processes involved in schema construction. In schema construction, new information entering the working memory must first be integrated with pre-existing schemas in long term memory. For this

process to take place, relevant schemas in long term memory must be decoded into working memory, where integration takes place. The result is an encoding of extended schemas stored in long term memory.

There is evidence that supports the notion that self-regulated and metacognition development are strongly related to cognitive load theory and that high cognitive loads can result in a reduction in the learners' capabilities to be self-regulated learners. The level of prior knowledge, which in turn is influenced by the intrinsic cognitive load, affects a learner's metacognitive development and degree of self-regulation. A learner with a lower level of prior domain knowledge will be more likely to experience a higher level of mental effort (Van Merriënboer & Paas, 1990; Cooper & Sweller, 1987). Metacognition and self-regulation can also relate to extraneous cognitive load because the monitoring, control, and reflection activities involved require additional mental effort, and therefore may result in a decreased performance for unskilled learners (Van Merriënboer & Paas, 1990; Cooper & Sweller, 1987).

According to CLT, when learners are novices in a domain, the cognitive load associated with unguided learning is high because novices lack any sort of guide to aid their knowledge acquisition processes (Mayer, 2004). Supporting this theory is empirical research examining the nature of expertise. Experts have been shown to have a greater depth and breadth in their cognitive schemas stored in long term memory (Chase & Simon, 1973; Chi, Glaser, & Rees, 1982), which suggests that experts require a much lower mental effort to process the information.

The CLT processes of schema automation and schema acquisition are closely linked to both Piaget's ideas of organisation and adaptation, and Sfard's theory of concept construction. The process of reification is essentially chunking of existing schema to form a new schema which becomes, in Sfard's terms, an object which is then itself interiorised, condensed and reified. It has been suggested that this "*chunking*" of information enhances learning by freeing resources for germane activities. Schema automation refers to a person's ability to acquire highly structured knowledge or schema with minimal error and with very low levels of conscious attention (Van Merriënboer & Paas, 1990). The construction and automation of schema does not serve to generate information and is believed to be the result of practice over a long period of time. If a learner reaches the point where schema can be processed automatically the cognitive resources are freely available to focus on other aspects of learning, including cognitive transfer and problem solving (Van Merriënboer & Paas, 1990; Cooper & Sweller, 1987).

Most researchers have focused on reducing extraneous cognitive load in learning to program. Their focus has been on facilitating the transfer of knowledge and skills (Cooper & Sweller, 1987). Researchers, in both mathematics and computer programming education, have found evidence that the simultaneous presentation of worked examples and problems facilitates schema acquisition and automation (Van Merriënboer & Paas, 1990; Cooper & Sweller, 1987). Tafton and Raiser (1993) reported that a worked example alone can have a positive impact on learning. However, other studies have found that learners tend to look only briefly at worked examples, consulting them only when they get into difficulties in solving their tasks (Van Merriënboer & Krammer, 1987). As a result of this finding the use of *example-completion* tasks was suggested to ensure that the learner focuses on the work example prior to attempting to solve the new problem. Van Merriënboer (1990) undertook a controlled experiment with an introductory computer programming course where one group was taught using fully worked examples and another group, in the same class, were taught using example-completion problems. The example-completion group were found to perform better on a related program writing problem than those who were provided with fully worked examples. This result was replicated in a study by Van Merriënboer and De Croock (1992).

These results were later used to argue that focusing on a fully worked example when trying to solve another, albeit similar, problem results in a high *intrinsic* and extraneous cognitive load because both the worked example and the task are concurrently processed in the working memory (Van Merriënboer, Kirschner, & Kester, 2003). Gray et al. (2007) suggest the use of *fading-worked examples* rather than fully worked examples. The main difference between example-completion tasks and fading-worked examples is that in fading-worked examples, the learner can complete the partially worked examples in stages by following certain patterns. The authors proposed that fading-worked examples should focus on graduated and repeated exposure to the programming concepts. Graduated exposure is suggested to help promote near transfer skills, while repeated exposure across a variety of problems helps promote far transfer skills. Near and far transfer are considered necessary skills for building effective schema (Perkins & Salomon, 1994). Muller (2005) devised a *pattern-oriented-instruction* approach for teaching computer programming and problem solving with a view to reducing the extraneous cognitive load required for learning. Pattern-oriented-instruction is based on using algorithmic patterns to reinforce schema acquisition. A similar idea is that of teaching and learning to program using *roles of variables*. Role of variables has been

shown to help novice programmers to build abstractions of common program constructs involving variables (Sajaniemi, 2002).

Other approach to reduce extraneous cognitive load and thereby increasing the working memory available for the germane purpose (schema acquisition and construction) is *simple-to-complex-ordering* of learning tasks. Van Merriënboer, Kirschner, and Kester (2003) suggested using a simple-to-complex version of the whole task. In this strategy, the novice learners start to practise problem solving on simple programming tasks and progress towards more complex tasks. The load associated with simple versions of the task is lower than the load associated with the more complex ones. These suggestions to reducing extraneous cognitive load have not yet been supported by empirical evidence.

In conclusion, there is evidence scattered throughout the literature that cognitive load plays a significant factor when learning. CLT may provide a useful means of reflecting on the effectiveness of the research instrument and research method. CLT may also provide a useful framework for explaining the data obtained from the participants when they are working on programming problems.

2.3. Theories of Learning: Knowledge and Strategy

The way learners organise their knowledge tends to vary, and that variation is evident in how knowledge organisation guides their retrieval and use of information during comprehending and writing computer code. Therefore in studying the development of knowledge, it is also useful to investigate our current understanding of the strategies that novices use to comprehend and generate programming code.

Soloway and Ehrlich (1984) carried out one early and influential study into knowledge organisation. They proposed a *top-down strategy* of program understanding in which programmers search for evidence and use this evidence to help decompose programming plans into lower level programming plans in order to build a mental representation. An alternative view is a *bottom-up program understanding strategy* in which the programmers start with individual code statements and chunk these statements into higher-level of abstractions. This chunking process is repeated successively at higher levels until a complete mental representation of the program is formed (Schulte, Clear, Taherkhani, Busjahn, & Paterson, 2010).

Some researchers do not acknowledge the primacy of either the top-down or the bottom-up program understanding strategy, and have concluded that programmers have the capability to switch between these two models (Letovsky, 1987; Mayrhauser & Vans,

1998). Letvosky, Mayrhauser, and Vans' conclusion was based on a study of professional programmers so it is possible that novice programmers are not opportunistic in the approach to program. Pennington (1987a) (1987b) described a comprehension process in which two mental models represent programming knowledge: the program model (i.e. the program text) and the situation model (i.e. data flow within a program, and function or goal of a program). She concluded that programmers who exhibit a higher level of comprehension were observed to cross reference frequently between these two models.

In addition to the above three models, there is another model that deals with program comprehension from a different perspective. Fix et al. (1993) focused on five abstract characteristics of the mental representation of computer programs that novices lack, namely hierarchical and multi-layered structure, explicit mapping of code to goals, foundation on recognition of recurring patterns, connection of knowledge, and grounding in the program text.

Other researchers have focused on investigating the strategies that novices use to generate/write a program code. Rist (1989) claimed that programming activity itself is hierarchically structured; it is built from simple knowledge structures that are combined to form more complex structures. At the lower level of detail they write lines of code and then at a higher level the individual lines of code are combined to create a programming plan. Finally, individual program plans need to be combined (via merging, nesting, abutment, and tailoring) to form the final program structure. Rist (1989) was interested in the processes that underlie the construction of programming plans. He concluded that if the novice has the appropriate knowledge, a schema that provides the program solution can be retrieved and the program design strategy used will be a *top-down and forward* design process. If the knowledge does not exist, then a schema must be created. It is generally accepted that in schema creation, novices tend to use a *bottom-up and backward* design process (Rist, 1989). Davies (1991) was interested in the process of plan construction, and he concluded that novices use a *top-down approach*. However, Davies noted that during the latter stages of construction, because of cognitive failures, the program design process takes on a more *opportunistic approach* in which the novice programmer cross references between distinct hierarchical levels. An alternative view proposed that novices use a *top-down, depth first search* in order to decompose the problem into sub problems. The novice programmer then explores each part as far as possible at a progressive level of detail, depending on the knowledge retrieved, until that part of the solution can be implemented in program code. An abstract view of the total

solution will never appear in the novice schema (Pirolli, 1986; Pirolli & Anderson, 1985; Anderson, Farrell, & Sauers, 1984; Jeffries, Turner, Polson, & Atwood, 1981). All the above researchers argued that novice programmers try to generate a program code through *brute force* (i.e. trial and error) by writing down the necessary steps when a programming plan is not available. All of the above studies, except Rist (1989), have relied on single snapshots for studying novice programmers writing a program code strategy.

The way in which individuals organize knowledge affects their ability to retrieve and use information effectively. Studies of strategies used by programmers typically focus on trying to understand what may be happening in the mind of novice programmers when they attempt to solve a programming task. Part of this research will focus on identifying common patterns (strategies) that participants apply when attempting to write a piece of code.

The active role of existing knowledge in assisting new learning is highlighted by literature relating to the transfer of learning and analogy in cognition. The next section provides a review of the literature relating to transfer in cognition.

2.4. Transfer in Cognition

Transfer in general refers to any use of past learning (and/or knowledge) when learning something new. The adaptation and use of that knowledge will reflect its origin, its original context and its current application for current goals. The learner will influence the subject in such a way as to reflect, knowingly or otherwise, what they have previously learnt (Robins, 2010).

Transfer of learning includes *near transfer* and *far transfer*. Near transfer (i.e. isomorphic transfer) refers to transfer “within domain” where the source and target are drawn from the same domain (Vosniadou & Ortony, 1989), for example, transfer among programming languages. Wu and Anderson (1990) classified near transfer into two subclasses in the context of learning to program: *learning transfer* and *problem solving transfer*.

An example of *learning transfer* can be found in a study undertaken by Scholtz and Wiedenbeck (1990). All subjects who joined the study had previous knowledge of C (procedural program paradigm) and PASCAL (procedural program paradigm), and they were asked to write a program using ICON (procedural program paradigm). The authors concluded that the transfer of learning focused on different types of programming

knowledge; at the lowest level were language syntax and language semantics, while algorithm development and problem solving were at the highest level.

An example of research, which investigated problem solving transfer can be found in Wu and Anderson (1990). They investigated problem solving transfer between three programming languages: LISP (functional program paradigm), PROLOG (logical program paradigm), and PASCAL (procedural program paradigm). They conducted three experiments. The first two experiments were between LISP and PROLOG while the third experiment was between LISP and PASCAL. The study participants had prior knowledge of all three programming languages. The authors reported that they had found three levels of transfer between programming languages: the syntactic level (for example, using same variable names and functions), the algorithm level (for example, choosing a similar algorithm such as recursive algorithm or iterative algorithm for different languages), and the abstract solution solving level (for example, checking that the elements of one- and two-dimensional arrays were sorted, the subjects would focus on aspects of tasks that are invariant amid transformations).

Both studies concluded that subjects were transferring their mental representation of one solution to the other solution and that the kind of transfer depends on common elements between programming languages such as the function of commands and the rationale of the concepts (i.e. logical reasoning). This leads to faster problem solving and fewer errors.

Far transfer, in contrast, involves transfer “*between domains*” where what is transferred is drawn from a different domain (Vosniadou & Ortony, 1989). For example, transfer occurring from computer programming (PASCAL programming) to mathematical problem solving (Algebra word problems) is considered to require far transfer of knowledge (i.e. logical problem solving transfer) (Olson, Catrambone, & Soloway, 1987).

Salomon and Perkins (1989) identified two distinct but related mechanisms (i.e. psychological paths for transfer), the *low road* vs. the *high road*. Low road transfer happens when stimulus conditions in the transfer context and the prior context of learning are nearly identical. In other words, in the context of novice programmers low road transfer is possible when the underlying solution rationale can be extracted and represented in the form of an abstracted solution schema. This abstracted schema enables learners to correctly transfer learned solutions to problems with new surface characteristics (i.e. minor changes to the code would change the behaviour of the code). High road transfer, in contrast, “*depends on mindful abstraction from the context of learning or application and a deliberate search for connections: What is the general*

pattern? What is needed? What principles might apply? What is known that might help?" (Perkins & Salomon, 1994, p. 6458). Such transfer is not in general reflexive. It takes time for exploration and the investment of mental exertion (schema acquisition). Low road transfer and high road transfer are examples of transfer by abstraction (Perkins & Salomon, 1994). Salomon and Perkins (1989) further defined two forms of high road transfer: forward-reaching and backward-reaching. In forward-reaching transfer the learner initially learns something and abstracts it in preparation for application in a new situation. In backward-reaching transfer the learner is face with a problem and abstracts key characteristics from the problem and reaches back into their existing knowledge for matches. The idea of transfer of knowledge may prove valuable in interpreting the results of the think aloud data. Because this research aims to reach some kind of understanding of the cognitive processes in learning to program it is likely that a theory which tries to explain the way in which schema are transferred would provide a useful understanding.

There is a large body of literature which centres on psychological paths for transfer. For example, Chi and Bassok (1989) focused on students' learning from the worked examples of problems dealing with the application of Newton's law. During the study, subjects were asked to solve two categories of problems: isomorphic problems to the worked examples and non-isomorphic problems taken directly from the target chapter (i.e. use of the principle involved in the examples in different and more complex problems). The researchers concluded that practicing multiple examples fosters transfer performance (i.e. development of Neo-Piagetian concrete stage).

Fuchs et al.(2003) focused on explicitly teaching for transfer of mathematical skills by providing tasks that help learners to connect between the new problem and previously solved mathematical problems (i.e. increasing metacognition) by developing categories for sorting problems that have got an identical schema (i.e. promoting schema abstraction). During the study, subjects were asked to solve three categories of problems. The first category contained problems that required solution methods similar to those that had been taught in class except that they covered different stories and quantities. The second category contained problems that, by comparison with those taught in class, had superficial feature changes that affected neither the problem solutions nor the structures. Examples of superficial feature changes are: different question format such as using multiple choice questions, different key-word vocabulary, and combining multiple questions in a larger problem solving context. The third category contained problems that were presented as a standardized achievement (i.e. what the student needs to know

irrelevant to the study questions) including additional problem structures taught as part of the curriculum.

Teague and Lister (2014a; 2014b) focused on isomorphic problems only, to investigate novice ability to reason reliably at the Neo-Piagetian concrete operational stage. In any case, the subjects usually do not identify the above problems classifications especially in solving the programming task since the time the learner spent to learn programming and practise problem solving is simply limited (Palumbo, 1990). But, if they are able to point out the relationship between the prior problems and the new problem, then they are likely to be able to solve the problem successfully (Simon & Hayes, 1976). The design of the code writing problems for this research will take into consideration prior knowledge, prior problems and be delivered in a sequence which means that the participants should be able to recognise the connection to earlier problems.

Analogy is a powerful cognitive mechanism for promoting the understanding of an unfamiliar situation (i.e. target analogy) in terms of a familiar one (i.e. source analogy) (Muller, 2005). When reasoning by analogy the target analogy is seen as “*the same kind of situation*” as the source analogy. Reasoning by analogy provides a way of focusing on identical sub goals in common.

The process of reasoning by analogy can be usefully decomposed into several basic constituent processes: “*identifying relevant analogue in memory, mapping the correspondences between an existing schema and the new instance, consequently, making inferences about the target analogues, and eventually, grasping a better understanding of the novel situation as a whole. As an important outcome, the analogy between specific analogues may lead to the formation of a new schema that encompasses them, to the addition of instances to memory, and to better understanding of old instances and schemas that allow better access in the future*” (Muller, 2005, p.59).

Broadly speaking, transfer by abstraction is similar to reasoning by analogy. To *abstract a solution* is to identify identical sub goals in common (i.e. patterns) and avoid contextual specificity. Explicit teaching of the connection between the new problem and previously solved problems increases metacognition and promotes schema abstraction as long as the cognitive load is controlled (Cooper & Sweller, 1987). Analogical reasoning is an important practice in the computer science domain. Therefore, realizing similarities and differences between problems and reuse of previously solved problems are essential for schema acquisition and automata.

2.5. Summary

This chapter has described the general theories of cognitive development and learning which may prove useful for the design and analysis phases of this research. Cognitive theories emphasize that learning does not occur suddenly but instead gradually expands this points to the need for research that is longitudinal. Much of the research that investigates the learning of novice programmers consists of research that looks at the learners at a single point in time. This research will follow students closely over their first year of learning to program and observations will be made of novice programmers writing code using a think aloud protocol. The overall aim of this research is to find some sort of explanation or understanding of the way in which the students adjust cognitive schemas when undertaking code writing tasks. One way of approaching this is to look at existing theories of cognitive development and learning (discussed in this chapter) or combination of theories and see if one or more of these theories or aspects of these theories can explain the observations made in this research.

Chapter 3. Research Methodology

3.1. Introduction

This chapter begins with an overview of methodological principles and a discussion of the philosophical perspective for this thesis. A detailed overview of the methods adopted in the research is then presented. This research takes a mixed methods approach because different methods are required in order to build a framework of question difficulty with which to construct the research instrument and to observe novice programmers learning.

3.2. A Pragmatic Research Approach

In order to conduct research, it is important to understand the underlying philosophical principles on which the research is constructed. Such perceptions can be subjectively based on the distinctions between *positivist research* and *interpretivist research* at the *paradigm*² level, the *ontological*³ level, the *epistemological*⁴ level, and the *methodological* level (Fitzgerald & Howcroft, 1998).

At the *paradigm level*, the *positivist* researcher assumes that an understanding of the world can be obtained from objective measurements that are repeatable and independent of social constructions (Fitzgerald & Howcroft, 1998). The *interpretive* researcher assumes an understanding of the world comes from the subjective experiences of individuals (Reeves & Hadberg, 2003). Thus, *interpretive* researchers may accept an intersubjective approach, which is the ontological and the epistemological belief that reality can only be described through social construction.

At the *ontological* level, the *positivist* researcher assumes knowledge is independent of social construction (Fitzgerald & Howcroft, 1998), and opposes the views of *interpretivist* researchers that multiple social realities exist that can be explored through human interaction in order to discover how and why individuals make sense of the world as situations emerge (Fitzgerald & Howcroft, 1998).

At the *epistemological* level, *positivists* assume an objective reality which can be described using quantitative properties to identify facts and draw inferences in an attempt to increase the predictive understanding of phenomena. *Interpretive researchers* assume

² *Paradigm* refers to “a pattern, structure, and framework or system of scientific and academic ideas, values and assumptions” (Olsen, Lodwick, & Dunlop, 1992; p. 16).

³ *Ontology* refers to nature and structure of the world or reality (Bryman, 1984).

⁴ *Epistemology* refers to the nature of human knowledge and understanding, which may be obtained through different types of inquiry and alternative methods of investigation (Fitzgerald & Howcroft, 1998).

reality is influenced by interaction with social factors including the researcher who recognises the meanings of individual actions within specific social contexts.

At the *methodological* level, the positivist tends to use methods, such as surveys and experiments that are quantifiable in nature – analysed using mathematically based methods (statistical and descriptive analysis) (Creswell, 2009). In contrast, interpretivists study things in their natural surroundings in order to understand or interpret phenomena in terms of the meanings participants bring to researchers (Harwell, 2011). Accordingly, researchers who employ this approach typically use methods such as grounded theory, interviews, and think aloud protocols to provide richer accounts of the phenomena (Creswell, 2009). Interpretivist researchers are the primary instrument for collecting the qualitative data. Therefore, to gain a flexible and open research process, social interaction is discouraged between participants and researchers (Harwell, 2011). The subsequent qualitative data is analysed based on the identification of a major theme (Creswell, 2009).

The *positivist* and *interpretivist* viewpoints shape the way methodologies are adapted by researchers. The two perspectives are philosophically distinct and there is some debate as to whether these two philosophical perspectives are conflicting and therefore should not be combined or may be combined using a mixed method approach (Onwuegbuzie & Leech, 2005).

According to Rossman and Wilson (1985), three major camps of thought have evolved from the qualitative and quantitative methodological approaches namely *purists*, *situationalists* and *pragmatists*. The difference between these three points of view relates to the extent to which the quantitative and qualitative paradigms co-exist and can be combined. The purists' camp believes in "*mono-method*" studies where quantitative and qualitative approaches cannot and should not be mixed. In contrast, the situationalists' camp believes that certain research questions are more suited to quantitative approaches, whereas other research questions lend themselves more to qualitative approaches. Situationalists believe that you should choose the method or approach which best suits the research question. The situationalist researcher only mixes the two methods when drawing conclusions at the end of the study during the overall interpretation; therefore the two approaches are treated as being complementary. The pragmatics' camp, unlike purists and situationalists, assert that a false dichotomy exists between quantitative and qualitative approaches (Newman & Benz, 1998). They believe that integrating quantitative and qualitative methods within a single study is appropriate because both approaches have their strengths and weaknesses and hence the researcher should adopt

both methods by employing the strengths of each of the techniques in order to better understand social phenomena (Creswell, 1995).

The research undertaken in this thesis adopts a pragmatic approach, where the nature of the research questions suggests that integrating quantitative and qualitative methods within a single study will add reliability and should allow the researcher to highlight the pertinent factors that impact novices learning to program.

3.3. Research Instrument Design

In order to conduct this research a research instrument must be designed. This instrument will consist of a series of programming or code writing tasks which will be presented to participants in a series sessions using a think aloud protocol. The aim is to design the questions in sequences such that each sequence is a series of increasingly difficult questions that build on each other in terms of programming concepts and cognitive schemas (or programming blocks/concepts). Thus, within a sequence questions will become progressively more complex.

To build such a sequence of questions a framework or method is required to measure the difficulty and complexity of code writing tasks given to novice programmers.

In order to build the framework the potential components need to be identified and evaluated. A quantitative method will be adopted, which allows the use of a statistical analysis for evaluating the usefulness of these components to measuring the difficulty and complexity of a code writing question.

A set of criteria will be established which will be used to select a set of code writing problems from a large repository. The repository of questions consists of problems from a series of controlled, summative practical programming tests held throughout the P1 course. Each of the selected code writing question's difficulty will be measured or estimated using the potential difficulty predictors. Participant performance on a question will then be used as a measurement of the "real" or observed difficulty of that question and compared with the difficulty of the question as measured by potential difficulty predictors (e.g. software metrics or taxonomic levels). In the computer science education literature the use of data from course assessments, aka naturally occurring data, and the use of student performance as a proxy for the measurement of a difficulty of a task is well documented and an accepted approach (Lister et al., 2007; Sheard et al., 2008; Shuhidan, Hamilton, & Souza, 2009).

The evaluation of software metrics as a predictor of difficulty will consist of a quantitative data collection stage and a quantitative analysis stage. In the data collection stage the performance of the participants on each question will be calculated. Relevant software metrics for the instructor model sample code will be calculated and be used as a measure of the complexity of the question, and then the correlation between the calculated software metrics data and the participants' performance will be calculated in order to measure the strength of the difficulty predictor (for further details see Section 4.6.2.1). This approach allows us to answer research question RQ1.1 and evaluate the hypothesis that software metrics provide a reliable and objective predictive measurement of the relative difficulty of a novice code writing question.

It is likely that a number of metrics will correlate with difficulty. This is due to the fact that many software metrics attempt to measure the complexity of code and it is logical to assume that the more complex the code the more difficult it is to write. In order to simplify the question of difficulty measurement it may be necessary to create a single measure of difficulty rather than a set of individual measurements. In such a situation, a statistical approach such as factor analysis, may be used in order to build a predictive model of a group of inter-correlated variables (for further details see Section 4.6.2.2).

An alternative to software metrics is a more subjective measure based on educational taxonomies such as Bloom's taxonomy (Bloom, Krathwohl, & Masia, 1956) and SOLO (Biggs & Collis, 1982). This approach has been investigated and reported in the literature with guidelines for the classification of novice code writing tasks using both of these taxonomies (Whalley et al., 2006). Researchers have reported that academics are able to reliably classify novice programming problems using the SOLO taxonomy (Lister et al., 2009) and that the cognitive level of the task reflects the actual difficulty observed. The use of taxonomies to evaluate the difficulty of a code writing task is discussed in more depth in Chapter 4 Section 4.3.

The method for evaluating software metrics as a measure of difficulty is purely quantitative and stems from a positivists perception. While the assignment of level of a taxonomy is more subjective, and therefore qualitative, the correlation of the assigned level with the observed difficulty (student performance) is quantitative. It may be that in order to measure the difficulty of a code writing problem that both the level of thinking required to solve the problem and the complexity of the code together form a richer way of describing and understanding the difficulty of a code writing problem. In investigating both software metrics and taxonomies as potential predictors of difficulty this research

will adopt a positivist stance by using triangulation (Bogdan & Biklen, 2006), using more than one theoretical scheme in the interpretation of the phenomenon, to increase the credibility and validity of the research instrument designed using the proposed difficulty framework.

In this research, a series of code writing questions, will be designed so that it progressively builds on programming concepts and for each progressive question to become slightly more difficult or complex to solve. The participants will be asked to attend a series of think aloud sessions. During a session, each participant will be observed attempting to solve these questions.

Because the question difficulty framework will be used to inform the design of the think aloud questions the framework may also be used to inform the interpretation and analysis of the think aloud data. Verbal protocol analysis will be used in order to extract patterns of behavior which will then be categorized according to a coding schema to draw conclusions about the relationship between the verbalizations (cognitive processes) and the task solution (final product quality)(Atman & Bursic, 1998). Figure 3.1 gives an overview of the philosophical perspective and methods adopted in this thesis.

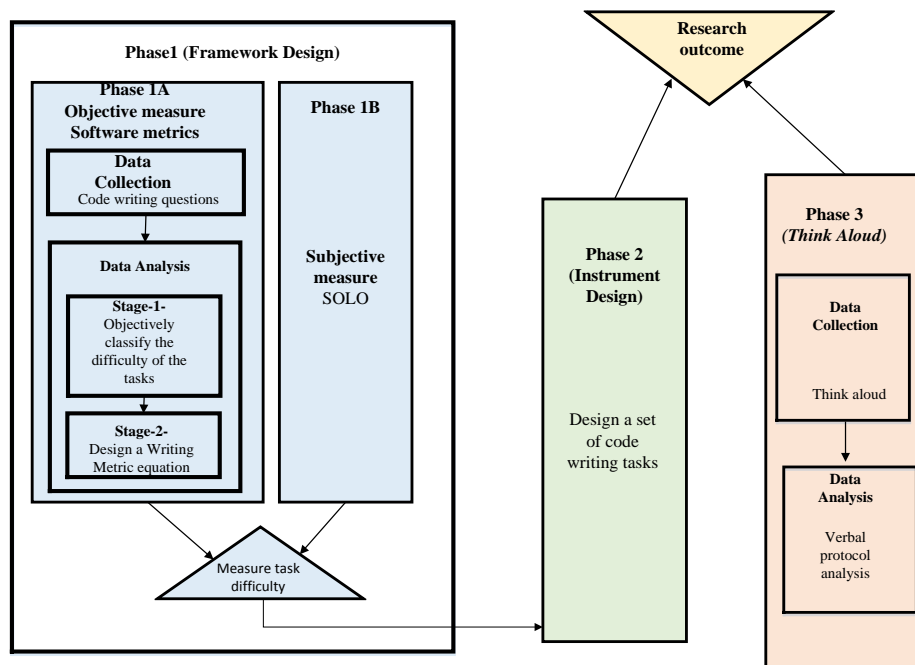


Figure 3.1 Philosophical perspective of this thesis

3.4. Ethics Consents

Ethics consents were granted for this PhD research by the Auckland University of Technology Ethics Committee (AUTECH).

For the first consent allowed for the collection of naturally occurring data and for conducting think aloud sessions using a smart-pen to record the student writing code. (Appendix B). A later amendment was made to this ethics application to allow for the use of video recordings of the think alouds and retrospective interviews (Appendix B).

3.5. Research Participants

The participants for this research came exclusively from the population of students studying P1 and P2 at Auckland University of Technology (AUT).

3.5.1. Recruitment

Recruitment involved a process which ensured informed and voluntary consent. The researcher appeared for ten minutes at the first lecture of the P1 course to outline the purpose and nature of the research. The consent forms and background information sheets⁵ were circulated to all students. Students who consented, henceforth called participants, returned a completed consent form and students were free to withdraw from the study at any time prior to the data analysis phase of the research.

On giving consent each participant was asked to fill a questionnaire⁶ that was used to establish the participants' prior programming knowledge.

3.5.2. Sampling Methods - Participant Selection

"A sample is a proportion or subset of a larger group called a population...A good sample is a miniature version of the population of which it is a part – just like it, only smaller" (Fink, 2003; p.82).

Two different sampling methods will be used in this study. One for the selection of participant data for developing and evaluating the difficulty framework and one for the selection of participants for the think aloud observations.

3.5.2.1. Sampling for the Difficulty Framework

For the first phase of the research that of framework design, it is the intent that the naturally occurring data from all students who volunteered will be used. This is because sufficient data is required so that statistical analysis of the correlation of difficulty predictors is valid. An analysis of the success of the participants (the sample) vs. the

⁵ See Appendix B, for the background information sheets and consent forms provided to students which were approved by AUTEK.

⁶ The background information questionnaire which was completed by the participants in the think aloud phase of this study is provided in Appendix C.

overall success of the cohort (the target population) will be used to ensure that the participants are representative (*non-random samples, purposive*) of the cohort. This means that the results of this phase of the study will be generalizable and that it will be possible to use the research instrument designed using this framework to future cohorts of the P1 paper. An assumption is made that the profile of the cohort from which the participants are recruited is representation of a typical novice computer programming class. The purposeful non-random sampling strategy used for quantitative data, which Patton(1990) referred to as *maximum variation sampling*, is also known as *homogeneous sampling* because the sample used for this part of the study were students all attending the same course. This type of sampling “*aims at capturing and describing the central themes or principal outcomes that cut across a great deal of participant or program variation*” (Patton, 1990, p. 172). Here “variation” refers to the students’ grade distribution. *Maximum variation sampling* is an accepted approach that has been widely adopted by computer science education researchers (for example see work arising as a result of the ITiCSE working groups Lister et al.(2004) and McCracken et al. (2001), and the BRACElet project Whalley et al. (2006).

3.5.2.2. Selecting Participants for Think Aloud Observations

The process adopted for sampling for think aloud observations will be the *critical case* sampling method. Critical case sampling is a type of purposive sampling technique that is particularly useful in exploratory qualitative research and is based on the premise that a small number of cases can be decisive in explaining the phenomenon of interest. The selected cases should be “*important*” cases - cases that are likely to “*yield the most information and have the greatest impact on the development of knowledge*” (Patton, 1990, p.236).

In order to identify the critical cases (desirable participants) it is necessary to first identify the dimensions that make a case critical. The main dimension for the selection of participants for analysis will be based on their dual ability to think aloud while simultaneously performing another task. Moreover, the participants should have no prior knowledge of programming. The performance of the participant on the programming course is also an important dimension. Critical cases could be consider as participants who excelled in the course and participants who fail the course. Selecting critical cases from each quartile of the cohort will provide a sufficiently broad view of the learning phenomena across the target population. Another dimension is likely to be the amount and richness of the verbal data collected (i.e. interesting cases), and also the participants’

commitment to the project in terms of attending the majority of the sessions scheduled. Although sampling for critical cases may not yield findings that are broadly generalizable it does allow for the development of logical generalizations from the rich evidence produced when studying a few cases in depth. However, such logical generalisations should be made carefully.

3.6. Think Aloud Method

The think aloud method has been used effectively in the areas of psychology and education to investigate cognitive processes. Think aloud requires participants to talk or think aloud their thought processes while solving a task. In other words, the think aloud method provides a description of cognitive processes or activities, ordered in time (Flower & Hayes, 1980). One of the strengths of the think aloud method is that, when conducted appropriately, it yields rich verbal data about reasoning during a problem solving task and can provide insights into cognitive processes, thoughts and feelings. Think aloud verbalisation is considered to be one of the most effective ways to assess higher-level thinking processes and it may be used to study individual differences in performing the same task (Olson, Duffy, & Mack, 1984). According to Ericsson and Simon (1993), the cognitive processes that generate verbalisations are a subset of the cognitive processes that generate behavior or action. Using think aloud and protocol analysis allows researchers to identify what information is concentrated on during problem solving and how that information is used to facilitate a solution to the problem. From this information, inferences can be made about the reasoning processes used during problem-solving.

Because think alouds are immediate, and performed concurrently with the task, the information is believed to be retrieved from working memory. Thus, in general think alouds are preferable to interviews post problem solving. Retrospective verbal reports require retrieval of information retained in long term memory which may be incorrect or incomplete. On the other hand, gathering real time data has issues because the load associated with problem solving and speaking simultaneously may be too difficult for some participants (Branch, 2000). Therefore, the goal of any method for thinking aloud is to minimise the cognitive effort in verbalisation in order to enable participants to articulate their thinking process and to ensure undue bias is not introduced by the interviewer.

3.6.1. Think Aloud Data Collection Protocol

Ericson and Simon (1993) provide guidelines on how to carry out valid and complete think alouds. These suggestions will form the protocol for the think aloud observations and data gathering for this research.

3.6.1.1. Training

Training participants to think aloud is a technique which was successfully employed by Rowland (1992) and it is recommended that participants are trained to think aloud to improve the quality of the data acquired. Therefore, in this research two individual think aloud train sessions were conducted to help the participants become more fluent with the thinking aloud process. In these sessions the technique will be explained to them emphasising that the idea was to verbalise whatever went through their minds and they will be given the opportunity to practice thinking aloud while solving some practice code writing tasks.

3.6.1.2. Instruction

Clear instructions will be given to the participants by the interviewer prior to the task. For example: *“Please keep talking out loud while solving the problem”*. The primary goal is to maintain focus on the task, with the articulation of thoughts as a secondary goal. Social interaction will be discouraged in order to keep the participant focused on the specific task at hand. When conducting a think aloud verbal method it is suggested that the interviewer should only intervene when the participant stops talking, and to simply say: *“Keep on talking”* (Van Someren, Barnard, & Sandberg, 1994). However, in this research, such situations should be avoided if possible because the interviewer is the sole researcher. Any intervention may lead the participants to believe either that something they are doing is wrong or that the researcher will help them and therefore they do not try to solve any problems encountered independently. It will be explained to the participants that the researcher’s role during interview sessions would be as an observer only. On completion of the think aloud session the interviewer will discuss with the participants their solutions and further explore their experiences in the study and in trying to solve the problems. Additionally, in response that discussion the researcher will be available to offer them individual guidance with respect to any of the programming concepts that were required to complete the relevant think aloud tasks, or any issues arising from their current programming studies outside the think aloud session that they wished to discuss.

3.6.1.3. Setting

“The first thing to do when one wants to get a subject to think aloud is to make sure that the setting is such that the subject feels at ease” (Van Someren, Barnard, & Sandberg, 1994, p.41). Taking into account this principle, and with the help from a department faculty member, a suitable room was allocated for conducting meeting sessions. There was large desk equipped with a desktop computer for the participants to use. Behind that desk, there was a wide area that allowed the interviewer to set up a camera with a built microphone to record the whole scene without disturbing the participant. There was a second desk where the interviewer could sit with the participant after the think aloud and question them about their thought processes during the solving of a problem.

3.6.1.4. Recording Think Aloud

When producing think aloud protocols, the researcher should not depend solely on the direct observation to collect data. It is very important to always use a recording device so the data can be viewed multiple times in order to ensure accurate and complete transcriptions (Brown & Rodgers, 2002). At the beginning of this research, the intention was to use a smart-pen and dot paper for collecting the data. Digital Evernote XML (.enex) files are produced using the smart-pen. These files can be played and reviewed, with synchronized visual and audio output, using Evernote software.

During the think alouds the researcher will also record field notes that consist of descriptions of major events (as detailed in Section 3.6.4.1). At the end of each session, the researcher will add additional notes reflecting on the session from their viewpoint.

In order to ensure that the methodology adopted for think alouds had a viable protocol and that the researcher was able to capture the data required for this research a pilot study was conducted. As a result of this pilot study it was expected that there would be some refinement required of the research method, for use with novice programmers, despite following the guidelines recommended in the literature.

3.6.2. Pilot Study & Data Collection Method Refinement

An initial group of 18 students studying P1 volunteered to be involved in the pilot study. From this group of four students had previous knowledge of programming and were not included in the sample group. Fourteen students attended the initial training sessions and on being informed of what the study entailed, four of those students withdrew their consent because they anticipated that they would not have enough time to fully commit to the study. After collecting and examining the initial think aloud data from the remaining 10 participants, seven were selected to take part in the pilot study. These

students had shown the dual ability to think aloud while attempting to solve the programming tasks. Two participants withdrew from the study before the last session but their data was included in the study because they had completed the majority of the study and provided critical cases for analysis. The findings of this pilot study pointed to two areas for improvement in the method, firstly the use of video to record think aloud and secondly the need for retrospective interviews. The data from this pilot study is not included in this thesis, but the results of the preliminary analysis of the data collected from this phase were published (Whalley & Kasto, 2014).

In the pilot study, it was noted by the researcher that as the complexity/and difficulty of the programming tasks increased, the participants started to encounter difficulties when solving the programming tasks and requested the use of a computer. In the post think aloud discussion, many of the participants reflected on the fact that using the Robot World on the computer in their class labs enabled them to visualise the execution of the program and gave them immediate feedback. The participants felt that programming on the computer was easier than using pen and paper to write code. They commented that in the case of an error on the computer, they could just experiment and change the code until they got it right and to check their solution they could run the program and check the output – something they could not do with pen and paper. Reflecting participants' experience during the pilot study it was decided that it would be more appropriate to video the students writing code on the computer in the development environment used in the context of their course rather than writing using smart-pen and paper. This conclusion was reached not only because the participants experiences suggested that this would be a better mechanism but also because writing code on the computer may be considered to be a more authentic task (Paperblanks, 2012).

The researcher trained the participants to verbalize their thoughts rather than to interpret them as suggested by Van Someren, Barnard, and Sandberg (1994). However, despite this coaching, during the pilot study, some of the participants found it nearly impossible to communicate their thoughts and had to be continually prompted to thinking aloud. Even with prompting they were unable to think aloud. It was observed by the researcher that as the difficulty of programming tasks increased the problem of lack of verbalisation increased and the majority of the pilot study participants found it nearly impossible to communicate their thoughts. As a result the think aloud data was sporadic and incomplete. It was theorised that the lack of verbalisation was not due to the participants being unable to verbalise but that it was due to the high cognitive load imposed on novice programmers

by code writing tasks. This conjecture was reached because even students who verbalised well on “easy” early tasks found it increasingly difficult to verbalise as the problems became more difficult and complex. This conjecture, and the effect of cognitive load, will be explored in more depth in the context of the full study and is discussed in later chapters. What is important at the method level is that way is found to compensate for situations where the think aloud data collected proves insufficient. As a result of the pilot study a retrospective interview phase was added to the data collection method (as detailed in Section 3.6.3). This phase will be held after the think alouds has been completed and will make use of the ability to review video and smart-pen data as a mechanism to trigger the retrospective interview discussions.

In order to film the think aloud sessions a camera attached to a tripod that allowed the interviewer to smoothly pan and zoom. The camera will be focused on the computer screen rather than on the participant. The camera was able to be locked in a fixed position so that the researcher can record observation notes when necessary. A Sony HDV 1080i video camera with a microphone, long-life battery and a wide-angle lens will be used. The video (.wmv) file produced by the camera can then be replayed using iMovie software.

3.6.3. Retrospective Interviews

In the case of retrospection method, the participants are questioned afterwards about the thought processes during the solving of a problem. The retrospection interview will be focused on portions of the think alouds which were incomprehensible, incomplete or confusing during the first phase as suggested by Van Someren, Barnard, and Sandberg (1994). The video provides an essential tool for triggering recall of these events.

Retrospection data are not the primary data source, but will be used to supplement any unclear data derived from the think aloud phase.

One drawback of this approach relates to the duration of the participants’ sessions. Because of the retrospective interviews the sessions will be longer. During the retrospection phase participants will often be asked to revisit their solution and to revise it as a result of nay insights they gain during the retrospective interview.

3.6.4. Stages of Verbal Protocol Analysis

Data analysis of verbal protocols typically consists of three stages: transcription, segmentation and encoding.

3.6.4.1. Transcription and Segmentation

In this study, the researcher is responsible for the transcription process for the following two reasons: Firstly, this process gives the researcher opportunity to revisit the think aloud sessions before encoding and analysis, and secondly it avoids issues associated with a third party transcribing the verbal reports (Jordan & Henderson, 2015). Such issues include the need to train a third party (Johnson, 2011), the third party must understand the research context (MacLean, Meyer, & Estable, 2004), the resultant omission of the researcher's role in interpreting the sessions (MacLean et al., 2004). Thus the selective transcription of a third party is unlikely match that of the researcher (Davidson, 2009), and the use of a third party can introduce data privacy and security issues (Jordan & Henderson, 2015).

A preliminary step in transcription is to view video while it is being collected. This strategy is useful in this research because the researcher herself conducts the interviews and therefore the data she wrote during meeting includes time-indexed *field notes* that consist of a description of major events, which is especially important in order to revisit fragments where the think aloud is incompressible, incomplete or confused. These notes will help to link retrospective interview data with the think aloud events.

The next step is for the researcher to quickly review the videotape soon after it is recorded in order to create a *content log*, which, like the field notes, will provide a time-indexed outline of the events on the videotape. Content logs can be extremely detailed, consisting of a description of major events that took place, or they can consist of a description of the content itself. Field notes and content logs allow the researcher to develop a sense of the corpus of data and facilitate the selection of episodes for further detailed analysis. Field notes and content logs are categorized under the indexing approaches which help in developing representations of video data (Jordan & Henderson, 2015). In later stages, the verbal report is transcribed and segmented by the researcher in details for the selected participants and questions.

The transcription guidelines adopted for this research are:

- The participant verbalisation and program code were transcribed and recreated in a table which consisted of a verbal statements column and a program code (i.e. program code episodes) column. The program code episodes were then embedded with two columns: program order column (i.e. the final order of the solution code), and generation order column (i.e. the order the program instructions appear in the code), in order to create the whole episode. Transcription guidelines for this stage were

adapted from Rist (1989; 1991). For the verbal statements column, preamble words “student” and “interviewer” were used to clearly distinguish the participant’s utterances from the interviewer’s utterances.

- Behavioural observations were registered as *action protocols* therefore an action column was included in the transcription table. Examples of an action include: “the participant uses his/her hand to point to the direction of movement of the robot”, “the participant deletes this line of code” and “the participant compiles his/her program for the second time”. Transcription guidelines for this stage were adapted from Van Someren, Barnard, & Sandberg (1994).
- Time-stamped notes were created and recorded under the *verbal statements* column for the participants’ utterances that were difficult to hear and understand. These time-stamped notes were revisited later by the researcher.
 - Breaks and hesitations in speech would be marked under the verbal statements column. For example: up to 5 seconds with [pause], and over 5 seconds with [long pause]. Recording such events is important as they are “good predictors of shifts in processing of cognitive structures” Ericsson and Simon (1993, p. 225).
 - In cases where participants simply verbalise exactly the code that they are typing, only the first few utterances are recorded in the *verbal statements* column during the think aloud session. These words are followed by [...] to indicate that this is the beginning of a direct code verbalisation. This relieves the immediate burden of encoding during the session. After the session the researcher will revisit this encoding to determine the end point of this duplicate verbalisation.
 - Where there were pauses in utterances, but the participant could be seen writing, the written words were recorded in the *program code* column while the *verbal statements* column was left empty. Silences, while carrying out another task (e.g., writing), were not likely to be indicative of shifts in processing of cognitive structures, so there was little value in recording a time stamp.
- Non-speech words such as noise, coughs and sneeze were not transcribed.
- The additional comments, code writing and explanations given during retrospection interviews were transcribed in the same manner as the think alouds.

Table 3.1 provides the transcription template which will be used to analyse each programming task.

Table 3.1 The transcription template

Program order	Generation Order	Program code	Participants Actions	Verbal statements

3.6.4.2. Transcript Encoding Techniques

The encoding of the verbal data follows the following coding schema

Learning behaviors were encoded using the novice programmer behaviors identified by Perkins et al.(1989):

- *Stoppers* – stop when confronted with a problem or a lack of direction.
- *Movers* – keep trying, experimenting, and modifying their code until they succeed. They are able to solve the problem independently or able to use interviewer feedback effectively and subsequently go on to solve the problem.
- *Tinkerers* – make changes by random permutation (i.e. trial and error) and they are typically unable to trace/track their code. Their behaviour like stoppers has little chance of progressing their ability to program.

Programming Strategies were used to encode the way in which the participants wrote and constructed their code. These classes were derived from the existing literature on learning to program as follows:

- *Stepwise design* (Soloway, 1986) – dividing an unknown problem into manageable sub-problems based on prior knowledge of similar problems. The solutions to these sub-problems are then recomposed in order to solve the unknown problem. Stepwise design is a divide and conquer technique (Sakhnini & Hazzan, 2008).
- *Familiar first* – code is written in two stages, first the participants focus on writing only the familiar parts or aspects of the problem. Their solution is then refined by

dealing with the parts not dealt with during the first phase of the solution (Sakhnini & Hazzan, 2008).

- *Trial and error* – program by random permutations in the hope of reaching a solution by chance.
- *Sequential* – code is written line by line in its complete and largely correct form. The initial code may contain a minor syntax error, such as a missing bracket or incorrect variable declaration, but it is easily fixed on compilation.

Activities encode other relevant code writing activities as:

- *Planning*
 - *Verbalise* – participants think aloud about aspects of their solution before actually writing the code.
 - *Pen and Paper* – participants use the smart-pen and paper to plan aspects of their solution in pseudo code or diagrammatically in the form of doodles.
- *Tracing* – participants work through the code they have already written in order to understand how the code is operating or to fix a problem (bug) in their code.
 - *Mental Tracing* – think aloud and reason about the written code or directly read aloud the code written.
 - *Pen and Paper*– using pen and paper to desk check or trace through the code, complete a trace table, or draw the changing state of the robot and its world.
 - *Visual debugging* – using the visual outputs of the Robot World and the robot’s animation to try and identify and fix a bug.
 - *Hand gestures* – hand waving to work out how the code is working, this is most likely when participants are trying to follow or track a robot’s direction and motion.
 - *PRINT debugging* – writing lines code which print the current state of variables out to console.
- *Unit test* – using the supplied unit tests to support programming.
 - *Reading* – reading and trying to interpret the unit test outputs/messages.
 - *Reading Test Code* – reading the actual unit test code in an attempt to understand either the test or the requirements imposed by the test.
 - *Reading PRINT debug* – reading and interpreting the results of PRINT debugging.
- *Time on task* – total time on the tasks, not including retrospection interviews.
- *Total Number of Compilations*

- *Total Number of Code Executions*

Timing point at which think aloud was conducted in terms of the number of weeks of learning programming.

Scaffolding a classification of the types of assistance provided, discussed in detail in Section 3.7, to the student by the researcher.

- *Interviewer intervention* – the assistance is provided by the interviewer when they feel that the participant has worked on the problem long enough and the participant shows no sign of being able to complete the task independently.
- *On request* – the assistance is provided at the request of the participant.

Scaffolding includes:

- *Clarify*
- *Generational Prompts*
- *Hint*
- *Exact solution*

Emotions:

- *Confused*
- *Indiscernible*
- *Surprised*
- *Relieved*
- *Happy*
- *Frustrated*
- *Hesitant*
- *Angry*

3.7. Intervention

The purpose of think aloud in this research was to gather data that closely reflected the mental processes used by a participant for solving the programming tasks.

Using the intervention model, proposed should give adequate opportunities for all the participants to successfully complete the task and for learning to take place.

In order to collect accurate data it is important to record these interventions and to track their effect on the participants' subsequent thought processes and to observe the effect of these interventions in helping the participant overcome difficulties. The use of such

intervention has its roots in Vygotsky's ZPD which defines the optimal level of challenge for a student's learning in terms of a task that the student cannot perform successfully on his/her own but could perform successfully with some help from knowledgeable other. Interviewer intervention measures will be recorded using a classification based on the kind of assistance supplied during the intervention. All interventions were considered to be a type of scaffolding and *"can range from doing almost the entire task for them [i.e. subjects] to giving them occasional hints on what to do next"* (Collins, Brown, & Holum, 1991, p. 7). The first three types of scaffolding (clarify, "general prompt" and hint) are directly adapted from Perkins and Martin (1985). While, the fourth type scaffolding, that may be given involves providing the participant with an exact and correct solution. This exact solution scaffold is based on the premise that *"In teaching programming - and problem solving in general a key objective is to develop useful methods of abstraction: If every problem a student must solve appears to be new and different, then there is little reuse of experience. A hallmark of expertise is the ability to view a current problem in terms of old problems, so that solution strategies can be transferred from the old situation to the current situation"* (Soloway, 1986, p. 852).

The intervention classes/types are defined below:

- **Clarify:**
Task requirements are explained to the participant in order to clarify the code writing task. For example, clarifying the terminology in the question text, removing ambiguity from the question text, or confirming the purpose of the task.
- **"General Prompt":**
Redirecting to encourage progress by allowing the participant to re-examine their solution, recognize errors if any, and fix those errors without any support or instruction being provided by the researcher. For example, suggesting that the participant may be able to solve the problem if they manually execute the code using smart-pen and dot paper.
- **Hint:**
If a couple of generational prompts does not help a participant to overcome their difficulty then the researcher may resort to providing some guidance in the form of a hint. For example, proposing a possible programming construct or indicating the location of a syntactic bug.

- **Exact Solution:**

If the participants simply cannot solve a problem, even after providing hints and “general prompt” or if they abandon the task an exact solution will be provided for them to review before the next session.

A *stepwise refinement* technique (Soloway, 1986) will be considered in this research in situations where participants are unable to progress. In this stepwise refinement, the researcher will firstly attempt to redirect the participants to think aloud using the smart-pen and the dot paper to solve the current problem (target code) by reminding the participants of problems which they have solved successfully before and helping them to arrive at the idea that such problems are part of the solution. Essentially the researcher is helping the participant to breaking the problem into sub-problems; these sub-problems can be merged or build on one another, each paving the way towards a solution of the overall problem. The fundamental motivation for a *stepwise refinement* technique is to redirect the participant to transfer solutions from previously solved problems to new problem (i.e. the ability to abstract similarities and apply previously solved problems to new situations). According to Rist (1989), the novice programmers tend to place the minimum possible load on the their memory; therefore explicitly teaching the connection between a new problem and previously solved problems increases metacognition as well as promoting schema abstraction as long as the cognitive load is controlled (Cooper & Sweller, 1987).

If the participant could solve the programming task on their own, the scaffolding provided to the participant was classified under “general prompt”, otherwise it was categorised under providing an exact solution; at this stage the role of the researcher started to provide the participant with the exact solution. The researcher also used the *stepwise refinement* technique instead of writing the code line by line to the participant.

3.8. Summary

This chapter has described the adoption of a pragmatic approach to research using mixed methods for this study. Three phases were identified namely; developing and verifying a task difficulty framework, instrument design using this framework and verbal protocol analysis with think aloud protocol and retrospection interviews, using the instrument, to collect data related to novice programmers learning to write code.

The next chapter focuses on the development and evaluation of a framework for predicting the difficulty of novice programmer programming tasks.

Chapter 4. Framework Design

4.1. Introduction

This chapter focuses on the design of a framework for describing programming tasks and their difficulty level. This framework will be used to design a sequence of code writing problems which are of increasing difficulty and complexity. The work discussed in this chapter can be broken down into two parts. Firstly, an examination of potential approaches to measuring the difficulty of novice code writing tasks and secondly an empirical evaluation of selected approaches. As a result of this work a novel framework for measuring the difficulty of such programming tasks was developed and this framework is presented here. The research discussed in this chapter addresses the first research question and related sub-questions.

4.2. Task Complexity vs. Task Difficulty

Many researchers have hypothesised that there is a difference between *task complexity* and *task difficulty* (for example: Braarud, 2001; and Campbell, 1988). Both authors agree that task complexity consists of two different dimensions. Firstly, *subjective complexity*, which relates to the kind of thinking, action, and knowledge needed in order to complete a task. And secondly, *objective complexity*, which is a characteristic of the task itself.

Task difficulty, on the other hand, is people performance on the task. Such tasks are defined as easy or hard, whether or not they are hard or easy is determined by how many people can accomplish the task correctly or successfully (Hunkins, 1995). Campbell (1988) concluded that while complex tasks are always difficult, difficult tasks are not always complex. For example, tracing a path through a maze with a pencil can be very complex, but is rarely difficult. Nevertheless, complexity may be a key concept in determining a task's difficulty. This view is supported by Börstler, Caspersen and Nordström (2007) who reported that measures of difficulty for exemplar computer program code, that are suitable for use in an educational context, must take into account factors such as the complexity of the code itself as well as the level of thinking required to understand that code. It is highly likely that this must also be the case for code writing tasks. Therefore, two types of measures of difficulty were considered in this research. *Subjective measures* and *objective measures*. The subjective measures considered are based on educational taxonomies. The use of taxonomies as a means of determining both the difficulty and the cognitive complexity of novice programming tasks has been well documented. The objective measures considered were selected from those used

commonly in the software engineering domain. Software metrics have been established as a way of obtaining objective, reproducible and quantifiable measurements, and have numerous uses from project planning and cost estimation to quality assurance and program complexity. Sections 4.4 to 4.6 detail some common software metrics, the selection of potential software metrics and the evaluation of these metrics as measures of novice code writing task difficulty.

4.3. Educational Taxonomies

Computer science educators have used various educational taxonomies to describe programming task complexity, approaches to solving programming tasks and to classify solutions to programming problems. The most widely adopted taxonomies to date have been the Bloom (1956) and SOLO (Biggs & Collis, 1982) taxonomies.

In 1956, Bloom produced a taxonomy that consisted of a multi-tiered set of learning objectives ordered according to their expected cognitive complexity (Figure 4.1). The taxonomy is a behavioral classification system of educational objectives. Many variants of the taxonomy have been proposed, but the most widely accepted is the revised Bloom's taxonomy (Anderson et al., 2001). This version of the taxonomy adds a knowledge dimension (factual knowledge, conceptual knowledge, procedural knowledge, and metacognitive knowledge), which specifies the type of information that is processed, to a revised version of the original cognitive process dimension.

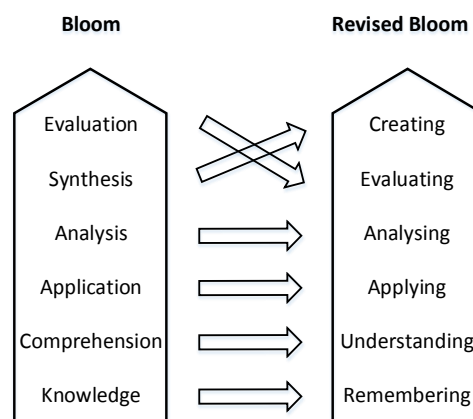


Figure 4.1 The cognitive process dimension; (left) Bloom's and (right) revised Bloom's taxonomy (adapted from Pohl, 2000, p.8)

A significant number of studies have used Bloom's and the revised Bloom's taxonomies to categorize the difficulty of novice programming tasks (e.g. Shuhidan, Hamilton, & Souza, 2009; Thompson et al., 2008; Lister & Leaney, 2007; Whalley et al., 2006). A number of researchers (e.g. Shuhidan, Hamilton, & Souza, 2009; Thompson et al., 2008;

Fuller et al., 2007) have pointed out that there are some problems related to the use and interpretation of Bloom's and the revised Bloom's taxonomy. Johnson and Fuller (2006) found that it was difficult to reach any sort of consensus when academics were asked to classify novice programming tasks using the Blooms taxonomy. Gluga et al. (2013) suggested that many of the ambiguities in the application of Bloom's taxonomy occur because the academics classifying novice programming tasks often have misunderstandings of the taxonomy categories as well as differing views on the difficulty of programming tasks. It is probable that, as the academics came from different institutions, these differing views occurred due to variations in their instructional focus. It has also been reported that the ordering of cognitive tasks in Bloom's taxonomy does not map easily to the learning paths of many novice programmers. "[S]tudents performing poorly on lower levels can still perform well on higher taxonomy levels" (Lahtinen, 2007, p. 23). As a result of these difficulties, several variants of Bloom's and revised Bloom's taxonomies have been proposed specifically for computer programming education (e.g. Bower, 2008; Fuller et al., 2007; Marzano et al., 2000). These variants have not been widely adopted by computer science educators and researchers. Perhaps this is partially due to the fact that the suitability of the Bloom's and revised Bloom's taxonomies to the design of learning and assessment activities has been disputed.

In a recent study, it was reported that the majority of the programming tasks in exams were at the *Application level* (Simon & Sheard, 2012). It is therefore likely that Bloom's levels may not be at a suitable level of granularity to be useful for determining the difficulty of novice (CS1) programming tasks.

Because of the lack of consensus in the computer science education domain on the use of Bloom's taxonomy, for the purpose of this research, it will not be adopted.

In 1982, Biggs and Collis developed a taxonomy called the Structure of Observed Learning Outcomes (SOLO). The SOLO taxonomy consists of a hierarchy of five stages or levels, which describe increasingly integrated thinking in a student's understanding of a subject (Figure 4.2). "[SOLO] is based on a quantitative measure (a change in the amount of detail learned) and a qualitative measure (the integration of the detail into a structural pattern). The lower levels focus on quantity (the amount the learner knows), while the higher levels focus on the integration, the development of relationships between the details and other concepts outside the learning domain" (Thompson, 2008, p.27).

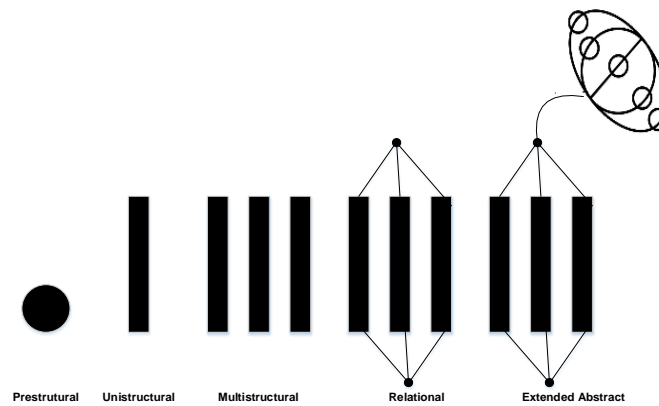


Figure 4.2 SOLO taxonomy (taken from Hook, 2016, p.1)

At the lowest SOLO level, *prestructural*, the student is able to learn about or acquire bits of information. They are unable to see or make connections between these bits of information. At this operational stage the student has bits of information which essentially have no structure and therefore make little sense. When the student begins to see simple connections between these bits of information but they do not fully understand the significance of the connections they are operating at the *unistructural* level. As the student progresses they are able to start to see more and more connections between bits of information but have yet to understand the importance of these connections (still unable to fully abstract ideas from these connections or to see meta-connections). At this stage the students is considered to be at the *multistructural* level of thinking. At the *relational* stage students are able to see the connections and their significance in the context of the subject area as a whole. Extended abstract is reached when the student is able to generalise, transfer and connect their knowledge both within and outside of the subject area.

Hattie and Purdie (1998) provided a number of examples or vignettes, not in computer programming, to help guide educators in the use of the SOLO taxonomy and it is this work on which much of the work related to using SOLO in computer science education is based, with their vignettes and examples guiding the development of an interpretation of SOLO for classifying novice computer programming tasks and student answers to code reading and writing tasks.

Computer science education researchers have reported greater success in using SOLO, rather than Bloom's taxonomy, to classify students' responses to code tracing and comprehension tasks (Sheard et al., 2008; Clear et al., 2008; Philpott, Robbins, & Whalley, 2007; Whalley et al., 2006; Lister, Simon, Thompson, Whalley, & Prasad,

2006), and code writing tasks (Ginat & Menashe, 2015; Seiter, 2015; Whalley, Clear, Robbins, & Thompson, 2011; Lister et al., 2009; Shuhidan et al., 2009).

Thompson et al. (2008) noted that, in order to design a richer model for programming tasks and their difficulties, the Bloom category for a programming task can be meaningfully mapped to a number of categories in the SOLO taxonomy. Inspired by Thompson's observation many researchers, for example Jakoš and Lokar (2015), and Meerbaum-Salant, Armoni, and Ben-Ari (2013), proposed a hybrid taxonomy for the classification of programming task difficulty. However, none of these hybrid taxonomies have been explored further so there is little empirical basis for their use as a measure of code writing task difficulty. These hybrid taxonomies are therefore not explored further.

The body of research into using SOLO for classifying code writing tasks has consistently reported that the higher the SOLO level of a task, the more difficult it is, as measured by student performance (Whalley et al., 2011; Clear et al., 2008; Sheard et al., 2008). This body of empirical research has resulted in accepted guidelines and rubrics for the classification of novice solutions to code writing tasks using SOLO.

A classification of task difficulty using the SOLO classification was therefore chosen as a subjective measure of code writing task difficulty because it has been found to be a reasonably reliable measure of code writing task cognitive complexity and this complexity has been found to be a measure which correlates to task difficulty.

Although the SOLO classification process is a subjective one, reliant on the classifiers knowledge of SOLO, the task being classified and the context of the task within the greater course of study, this cognitive complexity dimension of the framework is needed because software metrics are unable to account directly for the cognitive complexity of the code writing tasks. Using SOLO provides a means of accounting for the way in which the novice programmer must structure their knowledge in order to solve a problem; from more surface to deeper constructs.

4.4. Software Metrics

When academics were asked to rank the difficulty of programming problems, from examinations, they found it difficult to agree on the difficulty of the problems (Simon et al., 2012). The degree of agreement among the academics in estimating difficulty was only 40%, so the inter-rater reliability was poor. This finding indicates that there is a need for an objective measure of difficulty in this research.

In this research software metrics are proposed as a means of objectively measuring the complexity, size and structure of the code. This approach seemed reasonable because one aspect of the difficulty of a code writing task is the code itself, (e.g. the syntax, structure and size) and so software metrics are likely to be a useful dimension to the “*difficulty framework*”. It is expected that combining SOLO with software metrics will more reliably measure the difficulty of code writing tasks than either would in isolation. SOLO provides a richer way of describing and considering the complexity and difficulty of a task while metrics are arguably more precise but provide less information about the nature of the task itself.

Kaner et al. (2004) defined a software metric as “*a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality*” (p.2). Software metrics were introduced to support the most critical issues in software development and offer support for planning, predicting, monitoring, controlling, and evaluating the quality of both software products and processes (Briand, Morasca, & Basili, 1996). A large number of software metrics have been developed for the measurement of relatively large-scale commercial software development projects. ISO 9126 (ISO, 2001) divides software metrics into two categories: *static metrics* and *dynamic metrics*.

Static metrics are the group of software metrics used to capture the static properties of code. These metrics are usually computed using static analysis tools, such as Checkstyle (2016), PMD (2016), and Rationale® Software Analyzer tool (2016). Static metrics are invariant and do not change regardless of whether or not the program executes. Most existing software metrics are static. Some metrics measure the size of a program such as *lines of code*, *number of methods*, *Halstead’s metrics* (Halstead, 1977) and *number of parameters*. Other metrics measure the structural complexity of the program code such as *Cyclomatic complexity* (McCabe, 1976), and *average nested block depth*. Last but not least, some static metrics measure object oriented properties of the program code such as *number of children* and *depth of inheritance*.

Dynamic metrics are usually computed from data collected during program execution (i.e. at runtime) (ISO, 2001). Dynamic metrics directly reflect the quality attributes of code in operation because they capture the actual values.

While there are many potential software metrics available that can be used for measuring various aspects of code (i.e. size, structure, and readability) these metrics are typically focused on a particular feature of a program and are often devised with a single

programming paradigm in mind. Table 4.1 provides a set of commonly used software metrics classified by metric type and their applicability to three main programming paradigms (i.e. imperative, procedural, and object oriented).

Table 4.1 Software metrics and their applicability across programming paradigms (taken from Kasto & Whalley, 2013, p.60)

Metric Type	Metric	Programming Paradigm		
		imperative	procedural	object oriented
Basic	Number of lines of code	✓	✓	✓
	Number of <i>blank</i> lines of code	✓	✓	✓
	Number of comment lines of code	✓	✓	✓
	Number of comment words	✓	✓	✓
	Number of statements	✓	✓	✓
	Number of methods		✓	✓
	Average line of code per method		✓	✓
	Number of parameters	✓	✓	✓
	Number of import statements		✓	✓
	Number of arguments		✓	✓
	Number of methods per class			✓
	Number of classes referenced			✓
	Average number of attributes per class			✓
	Number of constructors			✓
	Average number of constructors per class			✓
	KLCID	✓	✓	✓
Complexity metrics	Cyclomatic complexity	✓	✓	✓
	Nested block depth	✓	✓	✓
Halstead metrics	Number of operands	✓	✓	✓
	Number of operators	✓	✓	✓
	Number of unique operands	✓	✓	✓
	Number of unique operators	✓	✓	✓
	Effort to implement		✓	✓
	Time to implement		✓	✓
	Program length		✓	✓
	Program level		✓	✓
	Program volume		✓	✓
	Maintainability index		✓	✓
Object oriented	Weight method per class			✓
	Response for class			✓
	Lack of cohesion of method.			✓
	Coupling between object classes			✓
	Depth of inheritance tree			✓
	Number of children			✓

4.5. Software Metrics and Learning to Program

This section focuses on using software metrics to support research related to the improvement of teaching and learning of computer programming and explores the relatively small body of work in this area. This research is limited to using software metrics as a way of aiding the design assessments and exemplar code, or as a means of providing feedback on code.

Some early work investigated the usefulness of software metrics as a form of formative feedback for novice programmers and as a diagnostic assessment tool for instructors (Cardell-Oliver, 1995). This work used the following measurements as forms of automated feedback: - program size metrics such as *lines of code*, *number of methods*, and *number of fields*, unit tests, and program style violation counts, but did not investigate these metrics as a way of estimating the difficulty of a novice programming task or guiding the design of code examples and tasks.

A preliminary study by Kasto & Whalley (2013) focused on the use of software metrics for determining the difficulty of code comprehension tasks. They found that dynamic metrics, Cyclomatic complexity and average block depth correlated significantly with the difficulty of code tracing tasks. They concluded that such an approach might also work for code writing tasks.

The remainder of this section focuses solely on software metrics and their use for measuring the difficulty and complexity of programming code tasks.

One of the obvious problems in using software metrics to inform the design of code writing tasks is that there is no software metric that measures code that has yet to be written and the aim of this work is to develop an objective means of measuring the difficulty of a novice code writing task *prior* to the students undertaking the task. For this research the choice was made to use the instructor's model answer as the code from which the metrics are calculated. Of course in theory, the model answer should provide a better quality solution when the task is sufficient enough for there to be variation in the possible solutions. In many cases an instructor's solution might actually have less complex code than many of the solutions produced by the students. Students, especially students who find programming challenging tend to produce code which includes redundant code. In order to produce a "good" solution students need to produce a more generalised, connected or integrated solution that reduces redundancy (Whalley et al., 2011).

This research explores code complexity and readability metrics; both of these types of metrics are potential components for the difficulty framework.

4.5.1. Complexity Metrics

McCabe proposed a complexity metric based on the structure of a piece of code as a control flow graph, making use of graph theory, which directly measures the number of linearly independent paths through a program's source code; he named this metric *Cyclomatic complexity* (McCabe, 1976). McCabe postulated that a program code with a large number of possible control paths would be more difficult to understand, maintain, and test. One limitation of the *Cyclomatic complexity* metric, is that code structures such as ELSE-statements and backward branches are not considered (Shepperd, 1988; Piwowarski, 1982; Magel, 1981) and it is highly likely that such structures contribute to the complexity of a code writing task for novice programmers.

Driven by the limitations of *Cyclomatic complexity* metric, Magel (1981) proposed a complexity metric making use of graph theory and *regular expressions*. Magel represented the structure of a piece of code as a control flow graph and then derived a regular expression from the control flow graph to calculate his *regular expression metric*. The symbols in the regular expression were then counted to give a structure complexity metric. Examples of the calculation of this metric can be found in Figure 4.3. The more nested the code the higher the value of the metric (Figure 4.3, A vs. B). Backward branches also contribute to a higher value than forward branches (Figure 4.3, D vs. C), and increasing complexity in selection statements (Figure 4.3, E and F) also results in higher values.

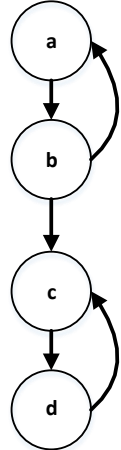
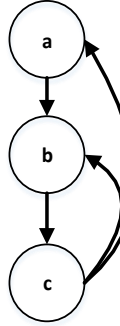
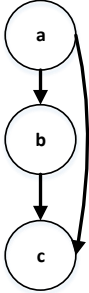
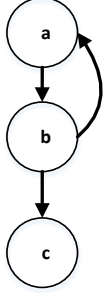
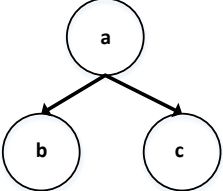
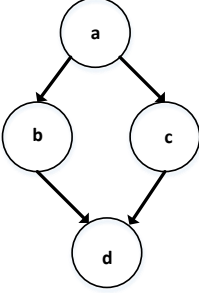
A	B	C
		
$ab(ab)^*c(dc)^*d = 14$	$abc(bc)^*(abc(bc)^*)^* = 19$	$a(b+n)c = 6$
D	E	F
		
$ab(ab)^*c = 8$	$a(b+c) = 6$	$a(b+c)d = 7$

Figure 4.3 Regular expression metric calculation and control flow graphs ($n = \text{no node}$)

A study conducted by Mathias et al. (1999) used software metrics in order to examine the underlying nature of code designed to study the process of program comprehension for novice programmers. A correlation was found between the nature of the task (measured by *lines of code* and *Cyclomatic complexity*) and the comprehension strategies used by novice programmers. However, using *lines of code* to measure code complexity of the program code is no longer accepted in the software engineering community because there is no agreed standard approach for counting the lines and also because the number of lines of code does not necessarily reflect the quality of the code. The easiest way to obtain a *lines of code* metric is by simply taking a count of all the physical lines in the source code. This raises issues when considering languages such as C and Java that allow statements to be split across multiple lines (Mathias et al., 1999). The authors concluded that, despite

these issues, *lines of code* as well as *Cyclomatic complexity* might be useful in measuring the difficulty of novice program comprehension tasks.

Parker and Becker (2003) employed *Halstead's metrics* to measure and compare the effectiveness of students' solutions of two different types of code writing assessment (constructivist and behaviourist). The authors did not reach any conclusions about the use of metrics beyond the fact that constructivist tasks appeared to require more effort as measured by both the metrics and student performance on the assignment.

A preliminary investigation by Klemola (1978) that measured student solutions to code writing tasks using *lines of code* and *Halstead's metrics* and that compared those measures with student performance, concluded that neither metrics were able to explain the error rate in the students' solutions. Follow-on research by Rilling and Klemola (2003) involved developing a software metric called the "*Kind of Line of Code Identifier Density*" (*KLCID*) metric as a means of analysing the cognitive complexity of program comprehension tasks. The proposed metric used the *Identifier Density (ID)* measure (proposed by Rilling & Klemola (2003)):

$$ID = TNoI/LOC$$

where *TNoI* is the total is number of identifiers and *LOC* is the total number of lines.

In order to compute *KLCID*, lines that have the same type of operands with the same arrangement of operators are considered equal and are counted as one unique line. For example, the lines (a = b + c) and (d = e + f) when a, b, c, d, e and f are of the same type are considered to be equal and are only counted once. *KLCID* is computed as:

$$KLCID = ID \in \text{unique } LOC / \text{number of unique lines containing identifiers}$$

A correlation was found between increasing *KLCID* and decreasing student performance for code comprehension tasks in a final examination of an introductory programming course. Klemola and Riling (2003) concluded that *KLCID* was "*a good candidate to measure the complexity of code comprehension assessment tasks within the same course*"(P.165). This finding was not surprising as in text comprehension it has been found that a higher density of *concepts* (i.e. *topic knowledge*) decreases the rate of comprehension (Kintsch, Teun, & Van, 1978). However, *KLCID* is considered to be limited in its application because it is time consuming to calculate. Because in this study the internal control structure for the different code writing task are the same therefore *KLCID* is not worth considering.

Petersen et al. (2011) investigated the relationship between cognitive load and the concepts used in the exam questions; the majority of the questions in their study were code writing tasks. The authors assessed cognitive load simply by counting the number of distinct concepts. They concluded that the more concepts the students needed to deal with to answer the question, the more difficult the question was and hence the higher the cognitive load. In this study, concept count is essentially being used as a measure of cognitive complexity. In order for this measure to be useful it would be necessary to have an accepted list of concepts, with definitions, so that it could be used reliably.

4.5.2. Readability Metrics

Readability refers to the ease with which text can be read. *Readability* can be considered to be a basic requirement for *understandability* (Börstler et al., 2007). It is difficult to understand a piece of text that is hard to read. It is reasonable to include a metric that measures the readability of code since empirical research has found that there is a strong relationship between the ability to explain the code and write code (Lopez et al., 2008).

Many well-known readability measures have been used in the assessment of English literature. These measures typically count the number of syllables, words and sentences in a piece of text and produce a single numeric value. For example, Flesch Reading Ease (Flesch, 1948), Gunning fog index (Gunning, 1952), SMOG (McLaughlin, 1969), Bormuth readability index (Bormuth, 1971), and Flesch-Kincaid Grade Level (Kintsch et al., 1978). Such measures simply parse text and do not measure understandability. Flesch-Kincaid Grade Level is a recalibration of the Flesch Reading Ease metric. While they both use the same core measures (word length and sentence length) they use different weighting factors and as a result are inversely related. A text with a comparatively high score on the Flesch Reading Ease test, meaning that the text is more readable, will give a low score on the Flesch-Kincaid test. The Gunning fog index and SMOG grade level both estimate the years of education needed on average to understand a piece of writing. The Bormuth readability index calculates a reading grade level required to read a text based on, firstly, average length of characters, and average number of familiar words in the text.

Readability measures have been the subject of considerable criticism because they fail to take into account factors such as prior knowledge of the reader, knowledge presumed by the writer, and the complexity of the written text (Bruce, Rubin, & Starr, 2015). Despite this “*readability formulas, unquestionably, have some utility. They have reasonable utility and predictability as a starting point for determining the level of challenge of a text*” (Pikulski, 2002, p. 10).

In 1998, Kurt Starsinic (Starsinic, 1998) explored the idea of the use of readability measures to assist him in training junior programmers during time-critical projects. He selected the Flesch-Kincaid Grade Level measure to produce a script called Fathom that was designed to automatically measure the readability of the programmers generated code (in Perl) and grade their work. Starsinic found it was difficult to map syllables, words and sentences to corresponding syntactic structures in code. How many syllables are there in ++ or { or \$_? Is *select* easier to read than *getHostByName*? As a result of this mapping difficulty; Starsinic developed a similar metric where he elected to measure the number of tokens per expression, the number of expressions per statement, and the number of statements per subroutine:

$$\text{code readability} = (\overline{elt} \times 0.55 + \overline{sle} \times 0.28) + \overline{srl} \times 0.08$$

where, *elt* is the expression length in tokens, *sle* is the statement length in expressions, *srl* is the subroutine length in statements, and where for example:

- tokens are ++ , ; , { , && , \$foo::bar, or any keyword
- expressions are 0.8, (\$a+6), wantarray?@a:0
- and statements are \$x++, \$a = \$foo::bar * 7.

Starsinic concluded that a low readability metric value indicates a more readable piece of code and that a piece of code with a readability of 2.91 was easy to read, whereas code with a readability of 6.85 was considered to be very complex and therefore hard to read. No justification or explanation is provided for the weightings given to each operand in the formula or for the thresholds that were used to determine the relative level of complexity of the code readability. It should be noted that Starsinic's code readability metric was not empirically evaluated and that the author makes no claims as to the effectiveness of the measure.

In a later study Börstler, Caspersen and Nordström (2007) proposed that some cognitive aspects of code reading can be expressed using common software measures and explored this idea in the context of two novice code reading tasks. Their aim was to develop a reliable means of selecting appropriate code examples to help guide novice programmers' learning and to distinguish between good and bad examples. They surmised that a good example must be readable and comprehensible, and they designed a framework based on these principles. Their framework consisted of *Cyclomatic complexity*, *Halstead's difficulty metric*, and an interpretation of the English language *Flesch Reading Ease*

measure they called a Software Readability Ease Score (SRES). SRES reinterpreted the Flesch Reading Ease metric parameters as follows:

- syllables → lexemes of the programming language
- words → statements
- sentences → units of abstraction

One of the limitations of the SRES measure is that syllables are interpreted as a character count. This is justified with the argument that words with fewer syllables also have fewer characters and might be meaningful in the context of the readability of natural language; while in the context of programming code meaningful variable names (which could have more characters) make the code more readable. Hence, while both measures will be investigated, it is anticipated that Starsinic's method might be better suited to measuring code readability not just in the context of this thesis but also for novice code writing tasks in general.

4.6. Selecting the Metrics: A GQM Approach

One of the key challenges faced by researchers when measuring software processes and products is the choice of appropriate measurements. This is also a challenge for this research. In the software engineering discipline the most widely known approach is to apply a goal oriented method called the *Goal Question Metric* (GQM) (Van Solingen & Berghout, 1999; Basili, Caldiera & Rombach, 1994) (Figure 4.4). The GQM measurement is a top-down system beginning with a focus on goals and consists of a set of rules for interpretation of measurement data. It is generally accepted that *“a bottom-up approach will not work because there are many observable characteristics in software..., but which metrics one uses and how one interprets them is not clear without the appropriate... goals to define the context”* (Basili, Caldiera & Rombach, 1994, p.528) The GQM has three levels, the *conceptual* level where goals are established, the *operational* level where questions are established, which help refine the goals and to assess whether or not the goal has been met, and the *quantitative* level where the metrics and data are selected which enable the questions to be answered. In this research because the data itself is quantitative only the objective dimension of the quantitative level of GQM need be considered.

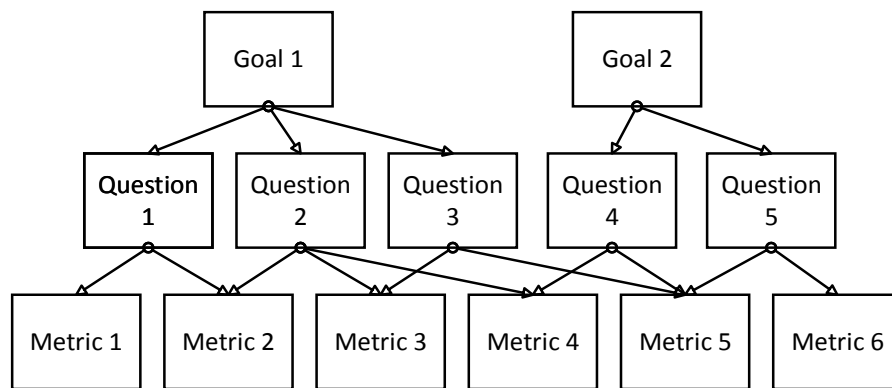


Figure 4.4 The GQM paradigm (taken from Basili, Caldiera, & Rombach, 1994, p. 3)

In order to select appropriate metrics, in the objective dimension, the GQM method was applied to ensure that only potentially useful metrics are evaluated as measures of code writing task difficulty.

There are some metrics that can be eliminated, prior to GQM selection. Most of the code writing tasks set for the students in P1 are procedural even though the code is encapsulated in classes and methods. For this reason, many of the object oriented metrics are not appropriate because the same measurement would be obtained for almost all novice tasks. Therefore procedural programming metrics were used in this research because they are more appropriate given the context of this research and the programming courses the students were studying. In addition, readability metrics were included as potential difficulty measures because the literature in the field provides evidence that it is difficult to write code if you cannot also read and understand code. The completed GQM template is shown in Table 4.2.

Table 4.2 GQM template

Goal		To measure the difficulty of novice programmer code writing tasks
Question	1	Does the size of the writing task influence the difficulty of code writing tasks?
Size- metrics	M1	– Number of commands
Question	2	What contributes most, if at all, to task difficulty operators or operands? Is it the number or the type of operands/operator which influence task difficulty?
Size- metrics	M2	– Number of operators
	M3	- Number of unique operators
	M4	- Number of operands
	M5	- Number of unique operands
Question	3	Does the structure of the code relate to how difficult it is to write the code?
Structure–metrics	M6	- Cyclomatic Complexity
	M7	- Average Block Depth
	M8	- Regular Expression
Question	4	Is code that students find more difficult to write also less readable?
Metrics-Readability	M9	- Adapted Starsinic’s metric the parameters to this method are the following: <ul style="list-style-type: none"> ○ Number of Keywords ○ Number of Expressions ○ Number of Operators (M2) ○ Number of Commands (M1)

In order to answer the first question in the GQM template (Table 4.2), *number of commands* was used because almost all of the procedures written required the students to call methods on objects. The heavy use of methods is an artefact of the use of a robot micro-world and the fact that in order to move and place the robot and beepers (the objects) methods must be called on the relevant object. Beyond this calling of methods or procedures the central aspects of most of the tasks were code structures such as selection and iteration statements and arrays.

The *number of commands* was calculated as the total number of executable lines of code rather than the total lines of code. A command line, was identified by a terminating semi-colon or an opening brace for a block of code. In the case of a FOR-loop statement, all code relevant to the loop was counted as a single statement (i.e. the initialization of the stepper variable, the test condition, and stepper update).

For the second GQM question four metrics were selected from the widely adopted *Halstead metric* set. These were the metrics specifically related to operators and operands. The *number of operators* will be directly proportional to the *number of operands* but both of these were selected as potential metrics because it maybe that novices actually perceived operators to be much harder to understand than the accompanying operands. It is also likely, but not proven, that code requiring the use of more types of operators contributes more to the difficulty of a task than the total number of operators and for this reason the unique count of operators and operands was included.

To answer the third question in the GQM, structural complexity metrics were included in an attempt to capture the inherent difficulty in certain structural properties of the code, such as an *average nested block* (i.e. a program code with too many levels of nested blocks can be difficult), as well as *Cyclomatic complexity*. Because of the limitations noted in the literature related to *Cyclomatic complexity* (Section 4.5.1), a *regular expression metric* also included.

To answer the last question in the GQM, number of the keywords, number of expressions, number of tokens and number of commands were selected to measure the difficulty of the readability. These are all parameters for Starsinic's readability metric (Starsinic, 1998).

In order to adopt Starsinic's metric the tokens and expression parameters were redefined as follows. Tokens were defined as a construct made up of Java operators (e.g.: ++, [], ||) and keywords (e.g.: int, return, void). Expressions were defined as a syntactic construction that has a value. Expressions are formed by combining variables, constants, and method returned values using operators. For this research the manner in which expressions were counted was altered. In Starsinic's method an expression such as $n=n+1$ was counted as one expression. But for this research it was counted as two expressions in an attempt to more closely map the way in which a novice might read the expression; it is likely that most novices would break this down into two expressions: - Firstly evaluating $n+1$, and then evaluating the assignment. Similarly, $(x+y)/10$ involves firstly an evaluation of the sub-expression $(x+y)$, and then an evaluation of the division. An example of the count of the basic units for a readability metric, namely the tokens, expressions and statements, is given in Figure 4.5.

```
public int method(int a, int b)
{
    int result = a - b;
    if(result < 0) {
        result *= -1;
    }
    return result;
}

tokens = 15
expressions = 8
statements = 5
readability = 1.88
```

Figure 4.5 Example readability metric calculation

The next two sections explore whether or not the software metrics might be a suitable way to estimate the difficulty of novice programming writing tasks.

4.6.1. Evaluating the Metrics

In order to evaluate the metrics selected using GQM a set of novice solutions to code writing tasks were selected for analysis. The software metrics were calculated using the instructor's model answer, and the correlation between each metric and the level of difficulty of the question based on participant performance was calculated. Metrics which show a strong correlation with difficulty were considered to be appropriate for use in the framework.

4.6.2. Data Set

The eleven Robot World code writing questions analysed in this chapter (See Appendix F) were selected from a series of controlled, summative practical programming tests held throughout the P1 course. For each question, the participants were provided with starting code (a BlueJ project) and relevant unit tests and required to add a method to a Java class in the project. The unit tests which were supplied tested the new functionality and provided the participants with an opportunity to check the correctness of their answer and fix any errors. Sixty participant responses were analysed for each question. These students gave ethical approval for their tests to be analysed and were a representative non-random sample of the entire cohort (Section 3.5.2.1).

The questions analysed in this section are limited to previously "unseen" questions presented to participants in a test situation. Unseen questions involve writing entirely new code for which the language constructs had been taught during the course but where the students had not previously seen the code during the course either as an example or as an exercise. If previously seen questions were included, it is likely that the correlation between the software metrics and difficulty would be insignificant. This insignificance is

likely to be due to the difficulty of the question being affected by the level of thinking required. A problem for which the participants have already seen the code may mean that participants can simply answer the question by recall which would compromise the effectiveness of the question for this research.

4.6.2.1. Data Analysis and Results

Table 4.3 gives the software metrics for the instructor’s model and participants’ performance for each of the questions analysed. It should be noted that the difficulty was measured as the percentage of fully correct answers. Question 11 was therefore the easiest question with a percentage difficulty of 100%, whereas question 1 was the most difficult question with 14% of participants giving a correct working solution.

Table 4.3 Metrics for instructor’s model and a percentage difficulty for each question

	Questions										
	1	2	3	4	5	6	7	8	9	10	11
Difficulty (%) (n=60)	14	24	39	52	55	63	84	84	90	98	100
Cyclomatic complexity	12	5	5	5	6	5	4	3	2	2	1
Average nested block depth	4	2	3	2	4	2	2	2	2	2	1
Number of operators	18	15	4	14	8	8	3	6	1	1	0
Number of unique operators	5	8	2	6	4	4	2	6	1	1	0
Number of operands	9	17	0	14	4	4	0	6	0	0	0
Number of unique operands	4	8	0	6	2	2	0	4	0	0	0
Number of commands	49	13	14	27	20	20	9	7	3	4	4
Regular expression metric	60	24	24	29	31	25	20	14	8	8	3
Readability metric	5.78	4.88	2.74	1.78	2.38	4.20	1.69	1.90	1.14	1.33	1.28

The significance of the correlation of each metric to the difficulty of each question was then tested using a Pearson’s correlation (Table 4.4).

Table 4.4 The correlations between metrics and difficulty

Software Metrics	Pearson’s correlation	
	<i>r</i>	<i>p</i> (one-tailed)
Number of operators	-0.87**	< 0.0001
Readability metric	-0.85**	< 0.0001
Regular expression metric	-0.85**	< 0.0001
Cyclomatic complexity	-0.85**	< 0.0001
Number of commands	-0.78**	0.002
Number of unique operators	-0.67*	0.012
Number of operands	-0.67*	0.012
Average nested block depth	-0.65*	0.015
Number of unique operands	-0.59*	0.035
** Correlation is significant at the 0.01 level (one-tailed)		
* Correlation is significant at the 0.05 level (one-tailed)		

According to Evans (1996) a correlation is considered to be very weak ($0 < r \leq 0.19$), weak ($0.20 \leq r \leq 0.39$), moderate ($0.40 \leq r \leq 0.59$), strong ($0.60 \leq r \leq 0.79$), and ($0.80 \leq r \leq 1$) indicates a very strong correlation. As shown in Table 4.4, all of the selected metrics

are either very strongly or strongly correlated to the difficulty of the task. Not unsurprisingly the correlations are negative, for example the higher the number of operators the lower the performance of the participants on the question and therefore the higher the difficulty of the question.

Unexpectedly, in the case of the questions analysed in this study, the *number of operators* correlates more strongly with difficulty ($r = -0.87$, p (one-tailed) < 0.001) than the *number of unique operators* ($r = -0.67$, p (one-tailed) $= 0.012$). The fact that more operators correlated with difficulty seems to support an idea central to CLT, which is that the more concepts novices need to deal with, the higher the cognitive load that will be imposed (Petersen et al., 2011). The repetition of operators perhaps gives novice programmers more opportunity to make mistakes.

The *number of operands*, in the case of the questions analysed in this study, also correlates strongly with the difficulty ($r = -0.67$, p (one-tailed) $= 0.012$) than the number of *unique operands* ($r = -0.594$, p (one-tailed) $= 0.035$). Perhaps the repetition of operands, like the repetition of operators, gives novice programmers more opportunity to make mistakes.

The *number of commands* correlates strongly to difficulty ($r = -0.78$, p (one-tailed) $= 0.002$), the higher the number of Java commands required, the more difficult the question is.

The *readability* measure was found to correlate very strongly with difficulty ($r = -0.85$, p (one-tailed) < 0.001). This means that as expected, in the case of these questions, the easier the code was for the students to write, the easier (according to the readability metric) the code is to read and understand. It is possible that there is a causal relationship between the ease of writing and readability of code. This idea has been explored in the literature and although no cause has been identified to date there is empirical evidence that there is some sort of relationship or correlation between code reading and code writing (Simon, Lopez, Sutton, & Clear, 2009).

The very strong correlation between the difficulty of a question and increasing structural and data flow complexity ($r = -0.85$, p (one-tailed) < 0.001), as measured by the *regular expression metric*, supports the conjecture that many students cannot write code that requires more complex structures and that there must be some relationship between the ability to structure code and being able to produce working code regardless of the quality of the code. Dijkstra (1997) claimed that the simpler the sequence control of the code the easier the code will be to read. This conclusion can be extended to the simpler the sequence control and the more readable the code the easier the writing of the code is.

Similarly, the higher the *Cyclomatic complexity*, the more complex the control flow of the program code is and the more difficult the question is ($r = -0.85$, p (one-tailed) < 0.001). As postulated by McCabe (1976) code with a large number of possible control paths should be more difficult to understand. The findings of this thesis research suggests that this statement may now be extended to include that the code is also harder to write.

The more deeply nested the branches of the code are, the higher the *average nested block depth* is and the more difficult the question was for the students ($r = -0.65$, p (one-tailed) $= 0.015$). This is not really surprising as research investigating student responses to code writing questions found that students find questions that can be solved by writing the code line by line with limited reference to the previous lines of code are easier than those that require the students to understand the relationship between the chunks or blocks of code that they have written (Whalley et al., 2011).

4.6.2.2. Factor Analysis-Principal Axis Factor

The nine metrics found to strongly correlate with task difficulty are further investigated in this section in an attempt to build a predictive model of a group of inter-correlated variables that may be used to estimate the difficulty of a code writing task at the task design phase.

Principal Axis Factor (PAF) is an approach to finding the least number of factors which can account for the common variance (correlation) of a set of variables. Thus, in this research PAF was used to simplify common variance amongst the set of possible variables (the nine software metric values (Table 4.3)).

Recommendations on the appropriate sample size of PAF vary considerably. However, Hogarty et al. (2005, p.222) noted that, “*our results show that there was not a minimum level of N or $N:p$ ratio to achieve good factor recovery across conditions examined*”. Where N was the number of participants and p the number of variables. Hogarty et al.’s finding is also consistent with that of MacCallum et al. (1999). This finding suggests that PAF is a suitable method for this analysis where the sample size is small. In this case N is the number of questions and p is the number of metrics.

Before applying the *PAF*, an *R-matrix* was calculated to check for *multicollinearity*. Multicollinearity exists when multiple factors are correlated not just to the response variable, but also to each other. When you have multicollinearity it can result in factors that are redundant. The issue with multicollinearity is that it increases the standard errors of the coefficients. This inflation of the standard errors makes some variables statistically

insignificant when they should be significant. Without multicollinearity (and thus, with lower standard errors), those coefficients might be significant.

Table 4.5 provides an *R-matrix*, for the data set of nine metrics, which consists of three rows, the first row of the table contains the results of a pairwise Pearson's correlation coefficient (r) between all of the metrics, the second row gives the one-tailed significance of these coefficients (p -value), and the final row lists the *Determinant* = $1.900E-015$. Field (2009) provides the guideline that the Determinant value of the R-matrix should be greater than 0.00001; if it is less than this value, it means that multicollinearity does exist. In this case the Determinant is less than the necessary value of 0.00001. Therefore, multicollinearity does exist in the data. One approach to mitigating the effect of multicollinearity is either to remove any variables that are not highly correlated with other variables or one of the two highly correlated predictors from the model. Further examination of the metrics and their relationship was therefore required in order to reduce multicollinearity before performing a PAF. As shown in Table 4.5, most of the metrics were actually *very strongly* correlated or *strongly* correlated with each other.

The *number of operators* was found to be *very strongly* correlated with the *number of operands*. As a result, the number of operators was proportional to the number of operands; therefore, there was a need to consider eliminating one of these two variables which clearly contribute to the *multicollinearity* problem. The operators and operands metrics measure essentially measure the same thing; as demonstrated by their very strong correlation. In deciding which of these two should be removed. The Determinant value was calculated twice. Firstly using the data set with operands and unique operands counts removed (Determinant = $2.721E-7$). The second instance using the data set with the operators and unique operators metric were removed (Determinant = $2.050E-7$). The Determinant value was similar in both calculations, therefore it was decided to remove the operands and unique operand metric as the strength of their correlation with task difficulty was much lower that for operators.

As the Cyclomatic complexity and regular expression metrics both convey essentially the same information; so it was important to consider eliminating one of these two variables to reduce *multicollinearity*.

The PAF was calculated twice more (after deleting the number of operands and the number of unique operands metrics) in order to determine whether to remove Cyclomatic complexity or the regular expression metric from the PAF.

1. Eliminating the Cyclomatic complexity gave a Determinant of 8.891E-005.
2. Eliminating the Regular Expression metric, gave a Determinant of 9.903E-005.

Both options resulted in an acceptable Determinant value but removing the regular expression metric gave a higher determinant value thus, *Cyclomatic complexity* was retained and the *regular expression* metric was removed.

The remaining six software metrics (*Cyclomatic complexity, average nested block depth, number of operators, number of unique operators, number of commands, and readability metric*) were used as the data set for calculating PAF.

Table 4.5 R-matrix

		Correlation Matrix ^a								
		Cyclomatic complexity	Average nested block depth	Number of operators	Number of unique operators	Number of operands	Number of unique operands	Number of commands	Regular expression metric	Readability metric
Correlation	Cyclomatic complexity	1.000	.758	.782	.376	.382	.339	.955	.994	.793
	Average nested block depth	.758	1.000	.368	.027	-.021	-.046	.657	.741	.418
	Number of operators	.782	.368	1.000	.819	.871	.839	.822	.809	.774
	Number of unique operators	.376	.027	.819	1.000	.927	.965	.404	.409	.556
	Number of operands	.382	-.021	.871	.927	1.000	.988	.454	.423	.530
	Number of unique operands	.339	-.046	.839	.965	.988	1.000	.395	.374	.517
	Number of commands	.955	.657	.822	.404	.454	.395	1.000	.973	.705
	Regular expression metric	.994	.741	.809	.409	.423	.374	.973	1.000	.760
	Readability metric	.793	.418	.774	.556	.530	.517	.705	.760	1.000
Sig. (1-tailed)	Cyclomatic complexity		.006	.004	.142	.138	.169	.000	.000	.003
	Average nested block depth	.006		.147	.470	.477	.450	.019	.007	.115
	Number of operators	.004	.147		.002	.001	.001	.002	.002	.004
	Number of unique operators	.142	.470	.002		.000	.000	.124	.121	.047
	Number of operands	.138	.477	.001	.000		.000	.094	.112	.057
	Number of unique operands	.169	.450	.001	.000	.000		.129	.144	.063
	Number of commands	.000	.019	.002	.124	.094	.129		.000	.011
	Regular expression metric	.000	.007	.002	.121	.112	.144	.000		.005
	Readability metric	.003	.115	.004	.047	.057	.063	.011	.005	

a. Determinant = 1.900E-015

The *Kaiser-Meyer-Olkin (KMO)* measure verifies the sampling adequacy for PAF analysis. Kaiser (1974) recommended accepting values less than 0.5 as barely acceptable. Hutcheson and Sofroniou (1999) provide a classification system for KMO values over 0.5. The classes are: mediocre (0.5 - 0.7), good (0.7 - 0.8), great (0.8 - 0.9), and superb values over 0.9. For this data set; the value of KMO was 0.642, which falls into the range considered to be mediocre. None the less the PAF was conducted because according the Kaiser criterion this level is acceptable.

A PAF analysis was run to obtain *eigenvalues*. Kaiser (1960) recommended retaining all factors with *eigenvalues* over than one. One factor in had *eigenvalues* over Kaiser’s criterion of one and can therefore be retained, and this factor accounted for 69.431% of the total variance (as shown in Table 4.6).

Table 4.6 Total variance explained

Total Variance Explained						
Factor	Initial Eigenvalues			Extraction Sums of Squared Loadings		
	Total	% of Variance	Cumulative %	Total	% of Variance	Cumulative %
1	4.412	73.541	73.541	4.166	69.431	69.431
2	.957	15.946	89.487			
3	.352	5.864	95.351			
4	.245	4.080	99.431			
5	.021	.356	99.788			
6	.013	.212	100.000			

Extraction Method: Principal Axis Factoring.

Figure 4.6 gives a scree plot of the principal factors and can be used to visually assess which components or factors explain most of the variability in the data. The scree plot confirms that most of the variability can be explained by the first factor and this factor should be retained. Factor 2 is at the point of inflexion and that and any subsequent factor should be discarded.

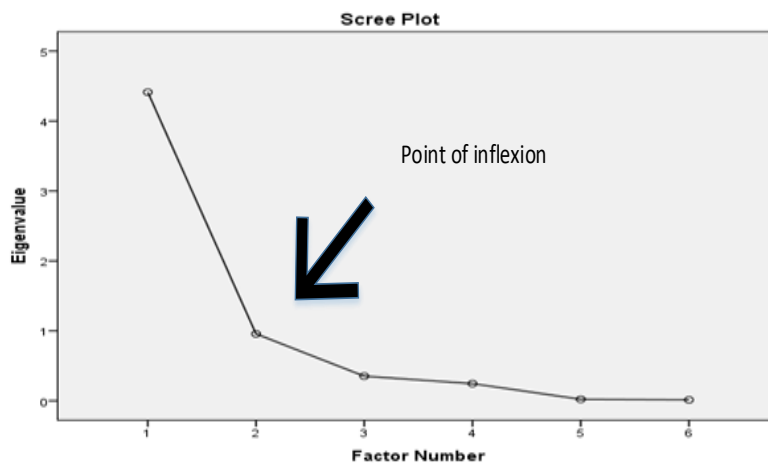


Figure 4.6 Scree plot for the analysis

Table 4.7 presents the factor loadings, which represents the degree of correlation between a specific observed variable and a specific factor. The higher values indicate a closer relationship.

Table 4.7 Factor matrix for the analysis

Factor Matrix a	
	Factor
	1
Cyclomatic complexity	0.974
Number of operators	0.930
Number of commands	0.920
Readability metric	0.834
Average nested block depth	0.647
Number of unique operators	0.627
Extraction Method: Principal Axis Factoring.	
a. 1 factors extracted. 6 iterations required.	

This principal factor suggests that structure, size and readability software measures combined are sufficient to capture the most important aspects of the difficulty of writing code for novice programmers. Hence, for this thesis the proposed objective metric included in the framework is the factor determines here and is called the Writing Metric (WM) and is defined equation:

$$\begin{aligned}
 WM = & (Cyclomatic\ complexity * 0.974) + (Number\ of\ operators * 0.930) \\
 & + (Number\ of\ commands * 0.920) + (Readability * 0.834) \\
 & + (Average\ nested\ block\ depth * 0.647) \\
 & + (Number\ of\ unique\ operators * 0.627)
 \end{aligned}$$

When interpreting the WM (based on an instructor's model answer), a program code with a lower WM value is considered an *easy code writing task* and a higher value is considered a *difficult code*.

4.7. Summary

This chapter has focused on the design of a framework for describing programming tasks and their difficulties. Two types of measures were suggested the subjective measure and the objective measure. Different subjective measures were discussed in detail in this chapter, and it was concluded, based on the literature review, that the SOLO taxonomy is a suitable measure for the cognitive complexity of the code writing tasks to be used in this study. For the objective measure, the GQM approach was used to select a suitable metrics. An evaluation of these metrics was undertaken that compared software metrics extracted from an instructor's model answer to a task with the participants' performance

on the tasks. A simple statistical approach was used to determine the degree to which each metric correlated with difficulty. The results of this analysis showed that most of the selected metrics were strongly correlated with the difficulty of the code writing task. Finally, the correlation between metrics was investigated further using factor analysis (PAF) and a novel *writing metric* for predicting the difficulty of code writing questions was developed.

Chapter 5. Research Instrument Design

5.1. Introduction

In the previous chapter, a framework for describing programming tasks and their difficulties was designed and evaluated. This chapter focuses on the design of a set of programming tasks (the research instrument) using this framework. These tasks are designed within the context and constraints of the pedagogy used in the programming courses (namely P1 and P2) at AUT University. Therefore, this chapter begins with a discussion of the delivery, pedagogical approaches, development tools and content of these programming courses.

This chapter then goes on to present the tasks within the difficulty framework. The tasks are in delivery sequences, which have been designed to trigger a reorganisation of a participant's cognitive structure. This restructuring can be thought of as being scaffolded by connecting each task with a previous task in a sequence and/or with some aspect of the participants' existing knowledge, and should result in the participants creating a product of learning as a new cognitive structure.

In order to create any new cognitive structure/schema participants need to be able to transfer their existing knowledge (from previously seen tasks in the sequence) to the new task at hand. Each sequence always starts with a minor variation of a task, which the participants have already seen in their programming course or in a previous sequence in this research instrument and therefore there is an assumption in this design that they should have prior knowledge/existing schemas to build on when faced with a task. Therefore, as part of building this set of program writing tasks the schemas required to reach a solution are considered.

5.2. Programming Courses at AUT

The programming courses from which participants were recruited were two introductory courses (Programming 1 (P1) and Programming 2 (P2)) in the first year of a bachelor's degree in computer science. The two courses focus on the development of problem solving skills and foundation programming concepts. The student cohort is diverse. While some students may go on to study further programming in software design and engineering courses, many of the students choose to major in topics in which no further programming is taught.

5.2.1. The P1 Teaching Approach

The P1 course at AUT University was designed with the assumption that the students have no prior knowledge of computer programming. The course was taught using the Java programming language⁷. The course was 12 weeks in length with three one-hour lectures per week. During this time, the lecturer introduced the key concepts (see Table 5.1) using slides, demonstrations, and question and answer sessions.

Table 5.1 Main P1 topics

Week	Lecture	Lab Task/ Homework Concepts
1	Course overview	Using a BlueJ project Writing a sequence of instructions using Robot World methods.
2	Basic Syntax Selection (IF/ELSE) WHILE-loops Nested (WHILE/IF, WHILE/WHILE)	Simple selection and iteration (not nested) only using Robot World methods.
3	Code tracing (desk checking) variables & data types	WHILE-loops, selection and nesting using primitive data type variables to store values.
4	Arithmetic Operators String Variables Input and string comparison Output-PRINT statements Comments	Print statements and using loops to count things.
5	Boolean expressions Methods, parameters and return values	String variables Concatenating and comparing strings
6	Planning and Developing an algorithm	Practicing previous lab & homework assignments
7	Methods calls with parameters Characteristics of good programs	Planning algorithms Writing methods
8	Object references 1D arrays (creating and accessing) For-loops	1D arrays and using for loops to iterate through the array
9	Handling errors & Exceptions Javadoc documentation Character expression	Using exceptions Documenting methods with Javadoc
10	Objects & classes	Objects & Classes
11	Constructors Mutator and Accessor methods Private methods	Objects & Classes
12	Packages & classes	Practicing previous lab & homework assignments

In addition to the lectures, the students attended one two-hour small class practical programming laboratory a week, which was supervised by teaching assistants (typically PhD students). In these lab sessions, students were given code writing problems to solve.

⁷ See Appendix D for a list of the learning outcomes for the P1 course.

For the majority of the course, students were not required to write their own classes but instead were asked to write methods for existing classes.

A number of tools were used in the teaching and learning during P1. These were selected by the course leader in order to facilitate teaching and learning. There has been extensive research into the advantages of using various software tools to assist in the teaching and learning of programming by, reducing the effect of complex programming environments, incorporating visualisation tools, and reducing or simplifying teacher workload (Gómez-Albarrán, 2005). Some studies have shown that these software tools have had positive impacts on student learning (Dougherty, 2007; Kölling, 1999; McIver & Conway, 1996; Cardell-Oliver, 1995; Pattis, 1981). While others have discovered no clear advantage in using such tools (Thomas, Ratcliffe, & Thomasson, 2004).

For P1, BlueJ version 3.1.4 was the courses prescribed integrated development environment (IDE). BlueJ provides a simplified mechanism with which students can edit, compile, and then execute their programs using a minimal interface. It has been argued that this simplified environment enables students to concentrate on solving programming problems without becoming distracted by the mechanics of compiling and executing programs (Kölling & Rosenberg, 2001). Moreover, BlueJ provides support, in a simplified way, for unit testing (Kölling & Rosenberg, 2001). Many researchers have reported on the advantages of using BlueJ in introductory programming courses (Haaster & Hagan, 2004; Ragonis & Ben-Ari, 2005).

In P1, the students were provided with unit tests for all code writing tasks. These tests allowed them to check whether or not their program provided a working solution to a specific task. These tests/test cases were designed to cover, as much as possible, all potential errors in a student's program. It should be noted, however, that a poorly structured solution and a well-structured solution might both pass the unit tests provided. Some research has been undertaken which supports the use of unit tests for example, Cardell-Oliver et al. (2011) found that when students were supplied with unit tests most students were able to write fully compiled and functionally correct code. Whalley and Philpott (2011) found that unit tests support the majority of students but that the weaker students still tinkered with their code because their knowledge of programming was very fragile. The benefits reported for most students motivated the introduction of unit tests to the course as a teaching and learning tool.

The P1 course, during the time this study was undertaken, also used an in-house micro-world, called "Robot World", which could be compiled and run in BlueJ. Robot World

was inspired by “Karel the Robot” (Pattis, 1981). Such micro-worlds are considered to be simple, interactive environments for student learning (Xinogalos, 2010). The advantages of micro-worlds have been well documented in the literature. These advantages include:

- Reducing the complexity of a language by providing a limited instruction set with simple syntax and semantics (Pattis, 1981).
- Enabling students to visualise the execution of the program by giving immediate feedback and assisting them in the debugging process (McIver & Conway, 1996).
- Increasing the focus on problem solving and algorithm design (Kölling, 1999).
- Facilitating learning better than text-based (non-visual) systems (Dougherty, 2007).

The Robot World used at AUT was designed specifically to be used in the first few weeks of the course in order to teach students the basics of sequential code, selection and repetitive structures. Other concepts such as strings and arrays were taught without Robot World.

At the beginning of the course, the students were provided with a small number of predefined methods that allowed robots to check their status, move within the Robot World and to pick up and drop off beepers (Table 5.2 gives the library of methods for the robot class).

Table 5.2 The provided Robot class methods

Robotworld createRobotworld()	boolean isRobotCarryingItems()
Robotworld createRobotworld(int,int,String,boolean)	boolean isRobotCarryingItems(Robot)
Robot createRobot()	boolean isItemOnGroundAtRobot()
Robot createRobot(int,int)	boolean isItemOnGroundAtRobot(Robot)
Robot createRobot(String,int,int)	void turnRobotLeft()
boolean isSpaceInFrontOfRobotClear()	void turnRobotLeft(Robot)
boolean isSpaceInFrontOfRobotClear(Robot)	void pickUpItemWithRobot()
boolean isGroundClearAtRobot()	void pickUpItemWithRobot(Robot)
boolean isGroundClearAtRobot(Robot)	void moveRobotForwards()
boolean isRobotDead()	void moveRobotForwards(Robot)
boolean isRobotDead(Robot)	void dropItemFromRobot()
boolean isRobotFacingWall()	void dropItemFromRobot(Robot)
boolean isRobotFacingWall(Robot)	void killRobot()

Students were required to complete ten weekly online quizzes consisting of code comprehension tasks such as reading, tracing and code completion questions. The homework consisted of code writing tasks. The students had a strict submission deadline to complete the lab online quizzes and homework assignments. The time allowed to complete each quiz and homework assignment was one week. Table 5.1 includes details of the main topics for the lab, online quizzes, and homework assignments. For the P1

course, the assessment consisted of marks for completion of lab online quizzes (10%), homework assignments (30%) and three controlled online code writing tests (test 1 (15%), test 2 (15%) and final exam (30%)).

All the tests and final exam were computer-based and open book. The main aim of the open book tests was to ensure that students did not spend too much time recalling or focused on syntax and to allow them to focus on the semantics and problem solving aspects of the tasks.

The workload, as determined by the course leader, required from each student in order to complete the course is provided in Table 5.3.

Table 5.3 The workload expectation for the P1 course (taken from the course descriptor)

Item	Weeks	Hours
In class	12	60
Independent study	12	90
Total learning hours		150

5.2.2. The P2 Teaching Approach

The main goal of the P2 course was to further develop the programming skills of the student by introducing Object-Oriented Programming (OOP) concepts such as inheritance and simple graphical user interfaces⁸. This course continued to use the Java programming language. During the course, students were transitioned from the simplified BlueJ environment to the fully-fledged development environment of NetBeans (NetBeans IDE 8.0.1).

P2 was taught by a different lecturer to P1 and with this change in teacher there was also a change in lecturing style. As for P1, P2 was 12 weeks in length, but with two one hour lectures per week and 108 hours of independent study time allocated. Again the lecture was used to introduce the core concepts (Table 5.4), using a similar approach to that of P1 but with less frequent examples and discussion and more of a focus on communicating the core topics using slides.

During the semester, the students were required to submit three take home assignments due in weeks five, eight, and twelve of the course (25%). Additionally, students received marks for completing the weekly lab code writing exercises (10%). There were four controlled programming/code writing tests (test1 (10%), test2 (10%), test3 (5%) and the final exam (40%)).

⁸ See Appendix E for a list of the learning outcomes for the P2 course.

Table 5.4 Main P2 topics

Week	Main topics
1	Objects and Classes Introducing inheritance
2	Class hierarchies and polymorphism Methods and Overriding
3	Primitive arrays (1D, 2D) and ArrayList Enumerators Information Hiding
4	Writing unit tests Abstraction & Interfaces
5	Revision of week1 to week4 topics
6	Java Collections Framework
7	GUI (Graphical User Interfaces)- Introducing Java Swing & NetBeans GUI: Layout managers and more
8	GUI3: Events UI Design & Usability
9	Debugging skills & tactics
10	Recursive algorithms Handling Errors & Exceptions
11	Designing Unit Tests Identifying 'good' test data
12	Review & Exam preparation

5.3. The Design Process

The tasks were specifically designed to encourage participants to think of new ways to apply previously learned concepts to a new task. In other words to trigger the restructuring or expansion of an existing cognitive schema. To do so effectively, the tasks need to include concepts and principles from previous tasks and embed them in new scenarios. If this is done correctly, participants will be forced to adapt their knowledge to new situations. According to Mayer (1977) in order to study “*the perception of the to-be learned material*”, “*the availability of a cognitive structure to which the new material may be assimilated*”, and “*the activation of the structure during learning*” (p. 370), the instructional design should be focused on three factors: sequencing, ordering, and organisation. Therefore, in designing this research instrument careful attention was given to the sequencing, ordering and organisation of both the course the students were studying and also to the design of the tasks. The tasks were designed within a sequence of conceptually similar but increasingly difficult questions. Several sequences were constructed to cover a range of programming concepts from assignment of variables through to selection, iteration and simple logic. In addition to applying the research framework each task in the sequence was also contextualised in terms of the schemas and prior knowledge/programming concepts required to solve the task in order to determine an appropriate timing and order for presentation of the tasks during the period of this research and the courses. Therefore, each sequence was designed around the teaching material and the timing of the delivery of the material in the course. Sequences one to

three consisted of tasks related to the content of the P1 course, while the programming tasks covered in sequences four and five were based on the course content of P2.

As discussed in Chapter 4, two types of measures were selected and used in this study to estimate the difficulty of the programming tasks, a *subjective measure* based on the SOLO taxonomy and an *objective measure* based on software metrics. The SOLO taxonomy is used to measure the cognitive difficulty of the task, while software metrics are used to measure the structural difficulty of the code.

5.4. SOLO Classification

As the tasks were designed within each sequence, the researcher classified each question using the SOLO taxonomy and using the difficulty metric in order to ensure that for each sequence the tasks became progressively more difficult.

In order to classify the tasks using SOLO the guidelines reported in the literature were considered (Table 5.5 and Table 5.6). Most of the work reported in the literature uses SOLO to classify novice programmer solutions to code comprehension and code writing tasks (Ginat & Menashe, 2015; Sudol-Delyser, 2015; Whalley et al., 2011; Thompson, 2010; Shuhidan et al., 2009; Lister et al., 2009; Clear et al., 2008). Lister et al. (2009) examined how SOLO might be used to classify novice solutions to code writing tasks and used the examples provided by Hattie and Purdie (1998) for language translation to inform their classification system. This classification system for novice code writing tasks was investigated further by Whalley et al. (2011). Whalley et al. used a grounded theory approach to identify salient elements in novice programmers' code. These salient elements were basic syntactic elements such as IF-statements and FOR-loops. Because the code writing tasks themselves were very simple they did not go beyond these basic syntactic elements and consider programming schema, plans or patterns. Whalley et al.'s SOLO classification was then based on the degree to which a student produced code that removed redundancy and provided a generalised solution. At the *unistructural* level students simply produced code which was a direct line by line translation of the task specification. A *multistructural* solution was still essentially a direct translation of the task specification but in translation some lines of code were rearranged in order to provide a working solution. This rearrangement may lead to a more integrated solution. At the *relational* level all redundancy (unnecessary repetition of lines of code e.g., multiple IF-statements) has been removed and the specifications integrated to form a logical whole. While Whalley et al. (2011) focused on using SOLO to classify novice programmers' solutions to code writing tasks their definition of levels considered the way in which the

code writing tasks was presented or phrased. Accordingly tasks which were phrased or described in detail and gave a line by line description lent themselves to solutions which were *unistructural* and a direct translation – this does not necessary mean that a valid *multistructural* or *relational* solution could not be produced but all that is required is a *unistructural* response. This research suggests that SOLO could be used to describe the code writing tasks itself. In an earlier paper by Thompson et al. (2010) a classification system was devised for code writing questions (Table 5.6) which was based solely on the way in which the task was specified.

Table 5.5 SOLO categories for code reading

	Code reading Clear et al. (2008)	Code reading Thompson (2010)	Code reading Sudol-Delyser (2015)
Relational	Provides a summary of what the code does in terms of the code’s purpose. (The “forest”).	Provides a summary of what the code does in terms of the code’s purpose. Provides a summary of the code that recognises applicability of the code segment to a wider context.	The parts of the problem are integrated into a coherent structure.
Multistructural	A line by line description is provided of all the code. (The individual “trees”).	A line by line description is provided of all the code. Summarisation of individual statements may be included.	Answer demonstrates a correct understanding of the parts of the problem, but there is no evidence of connection between problem parts.
Unistructural	Provides a description for one portion of the code.	Provides a description for one portion of the code (i.e. describes the IF-statement).	Answer demonstrates an understanding of some but not all aspects of the problem.

Table 5.6 SOLO categories for code writing

	Student solution Lister et al. (2009)	Student solution Shuhidan, Hamilton, and Souza (2009)	Code writing question Thompson (2010)	Student solution Thompson (2010)	Student solution Whalley et al. (2011)	Student solution Ginat and Menashe (2015)
Relational	Provides a valid well-structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.	Fully correct or almost right. Novices appreciate significance in relation to the whole program and can generalise outside of program.	Requires interpretation and decomposition in order to arrive at a suitable solution. Although the specification provides all of the details for a solution, it provides few clues that would hint at the structure of the solution.	Provides a valid well-structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.	Provides a valid well-structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.	A valid well-structured solution that involves the composition of two or more design patterns, integrated in a non-simple, interleaved manner, to form a logical whole.
Multistructural	Represents a translation that is close to a direct translation. The code may have been reordered to make a valid solution.	There are numbers of connections made. Novices can create code for loops and comparisons, but there are a few minor slips, leading to failure to connect the whole idea. They may fail to convert arguments, use incorrect operators, and not interpret general explanation.	Requires some interpretation in order to arrive at a suitable solution. Some parts of the specification may be directly translatable into the solution.	Represents a translation that is close to a direct translation. The code may have been reordered to make a valid solution.	Represents a translation that is close to a direct translation. The code may have been reordered to make a more integrated and/or valid solution.	A translation of the specifications into flexible manipulation of a generic design pattern; or a simple, elementary composition of more than one generic pattern.
Unistructural	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.	Simple connections are made. Novices can compare, or write loops, but fail to implement or derive the connections of loops in relation to manipulation of arrays or usage of further structures.	Requires a direct translation of the specification into a possible solution.	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.	Direct translation of the specifications into a straightforward implementation of a generic design pattern.

For this research either the Thompson et al. (2010) classification might be adopted or a simple translation of the descriptions from Whalley et al. (2011) reframed in terms of the task itself might be sufficient. However, the tasks in this research are different those used to develop their SOLO classification systems. The tasks in this research are designed explicitly to require multiple schemas and merging or nesting of those schemas to form an integrated and generalised solution. The level of syntactic elements which was used by Whalley et al. (2011) to develop classifications of code writing tasks using SOLO is not appropriate for the tasks reported in this study. None of the tasks in this study will be expressed such that a direct line by line translation is required because the intention is to trigger retrieval of abstract schemas. This means that if we were to use existing classification systems all the tasks would be relational. On reflection the published classification systems are very limited in their applicability to novice programming tasks which are focused on building abstract plans and problem solving skills. For these reasons a new SOLO classification system is used in this research based on the notion of cognitive schemas (Table 5.7). In order to have confidence in the validity of the SOLO classification for the tasks developed for this research the tasks were also classified by two academics experienced in using SOLO. Any differences were discussed until an agreement was reached.

In order to clarify the classification process and commentary of the classification of three tasks from sequence 1 (see Section 5.7 for full details of these tasks) follows.

Table 5.7 Novel SOLO classification categories for code writing tasks using schemas

SOLO category	Description
Relational [R]	Task requires the merging or nesting of more than one schema or parts of schemas in order to produce a generalised working solution.
Multistructural [M]	The task requires the use of either the same schema repeated or two or more simple or familiar schemas that combined using a simple construction process such as schema abutment (sequential concatenation) to produce a solution.
Unistructural [U]	The task requires the use of a single schema (not repeated and familiar) which can be used directly without modification.

- **Length of a corridor – Unistructural Task**

This question asked the participants to calculate the length of a corridor. In Robot World this is achieved by moving the robot to the end of a corridor and counting the number of moves it has to make in order to get to the end of the corridor. Solving this problem requires the use of a single schema which is familiar to the participants and

requires no modification. In order to program any code in the Robot World moving the robot through the world is essential. The participants have prior this tasks repeatedly used this programming schema to solve code writing tasks and indeed have actually written code to count the length of a corridor:

```
While robot is not facing a wall
```

```
    Move forward
```

```
    Increment Counter
```

- **Compare the length of two corridors – Multistructural Task**

In order to solve this problem three simple schemas are required one to calculate the length of a corridor (as above), one to move the robot to the next corridor, and one to compare two numbers. In order to solve this task the length of a corridor schema is repeated twice and then the comparison of the lengths is made, the schemas do not require anything beyond minor tailoring and can essentially be used directly and are combined in a simple abutment or concatenation process. The corridors are always in the same position in the world. This meant that the task lends itself to being solved using a *multistructural* approach and does not mean that a relational solution is not possible. Thus the classification is made to indicate the lowest possible SOLO level at which the novice programmer can operate in order to solve the task and is not dependant on the way the task is phrased.

Calculate length

```
While robot is not facing a wall
```

```
    Move forward
```

```
    Increment Counter1
```

Move to next Corridor

Calculate length

```
While robot is not facing a wall
```

```
    Move forward
```

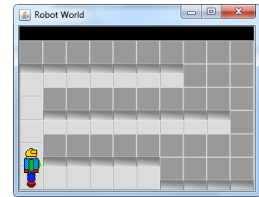
```
    Increment Counter2
```

Compare values of Counter 1 and Counter 2

Print result

- **Find the longest corridor – Relational Task**

This task required finding the longest corridor where there may be any number of corridors in the world. The corridors alternate and are always adjacent to each other. The image to the right gives an example scenario with three corridors. In



order to solve this problem the schemas used in the previous task were required but in order to generate a valid solution the schemas must be merged and adapted. The schema could not just be sequentially joined or concatenated and produce a correct solution.

Create most-wanted variable

calculate length of the first corridor and store the result in most-wanted

While there are more corridors

calculate length and store the result in current

if current > most-wanted

most-wanted = current;

Return most-wanted

As an addendum in 2015, after this phase of the research and the think aloud data collection was completed a paper was published which investigated using SOLO for the classification of code writing tasks and student solutions that required the development of simple CS1 level algorithms (Ginat & Menashe, 2015). Their classification system for classifying the tasks is given in Table 5.6. Their encoding focused “*on the selection and manipulation of [algorithmic] design patterns*” (Ginat & Menashe, 2015; p.456). They classified seven code writing tasks using their SOLO level descriptors. These tasks were designed to make students abstract basic pattern structures, and to help students see these patterns as templates that can be manipulated. Ginat and Menashe’s classification system is remarkably similar to the one conceived for this research in 2013 – simply exchange pattern for schema – and gives us confidence that our interpretation of SOLO for determining the level of thinking (hence difficulty) of a task is reasonable.

5.5. Transfer Learning: Classification of the Tasks

It is well accepted that learning requires the *transfer* of prior knowledge and/or skills to a new situation. Indeed, the tasks for this study were designed so that students should be able to transfer what has been learnt in a previous task (the *transfer* source) to the new task at hand (the *transfer* target).

There are a number of different views of *transfer*. The traditional view is that of Thorndike (1923) who believed that *transfer* depended on the original and the transfer tasks having identical elements or similar features (stimuli) and where a clear and known relationship exists between the tasks. Skinner (1953) suggested that *transfer* involves generalization of responses from one discriminative stimulus to another. More recently it has been suggested that *transfer* involves activating knowledge in memory networks and requires links between links between pieces of information in memory (Gagné, Briggs, & Wager, 1992). The more links there are the more likely activating one piece of knowledge will activate another. Knowledge *transfer* is therefore a complex process for which some taxonomies have been developed to define different types of *transfer* (Table 5.8).

Table 5.8 Transfer types (taken from Schunk 2012, p.319 Table 7.4)

Type	Characteristics	Source
Near	Much overlap between situations; original and transfer contexts are highly similar.	Royer (1986)
Far	Little overlap between situations; original and transfer contexts are dissimilar.	
Literal	Intact skill or knowledge transfers to a new task.	
Figural	Use of some aspects of general knowledge to think or learn about a problem, such as with analogies or metaphors.	
Low road	Transfer of well-established skills in spontaneous and possibly automatic fashion.	Salomon and Perkins (1989)
High road	Transfer involving abstraction through an explicit conscious formulation of connections between situations.	
High road Forward reaching	Abstracting behaviour and cognitions from the learning context to one or more potential transfer contexts.	
High road Backward reaching	Abstracting in the transfer context features of the situation that allow for integration with previously learned skills and knowledge.	

Typically *near transfer* refers to transfer within a domain where the source and target are drawn from the same domain (Vosniadou & Ortony, 1989), for example the transfer of knowledge between programming languages. When tasks have similar elemental structures they are often called *isomorphic tasks* (adjective | iso·mor·phic being of identical or similar form, shape, or structure) and transferring from one task to another involves *near transfer*. On the other hand *far transfer* involves transfer between domains where what is transferred is drawn from a different domain (Vosniadou & Ortony, 1989). For example a far transfer task might involve transferring algebraic knowledge to a task

that requires the development of a computer program (Olson, Catrambone, & Soloway, 1987).

In the context of computer programming a *far transfer test* has been defined as “*the design and construction of programs for new programming problems that require solutions not encountered before*” (Van Merriënboer, 1997, p. 277) and “*near transfer test*” as “*a test that measures knowledge of commands, syntax and standard language constructs of the programming language*” (Van Merriënboer, 1997, p. 276). A similar distinction was made by Scholtz and Wiedenbeck (1990). In their study they gave students with knowledge of two programming languages (C and PASCAL) a problem in a new programming language (ICON). This distinction is too broad when considering the novice programming tasks designed for this research. However unarguably the notion of *transfer* is a useful mechanism for distinguishing between the designed tasks and might also be useful in helping explaining why a participant might be able to solve one task and not solve another task. Therefore the following novel knowledge transfer classification consisting of three types was developed by the researcher to distinguish between near and far transfer novice programming tasks:

1. *Isomorphic Tasks*

Tasks that share the same programming concepts and code structure (in terms of salient elements, and order of those elements), but are superficially different. Thus, the underlying solution rationale can be extracted and represented in the form of an abstracted solution schema. This abstracted schema enables learners to correctly transfer learned solutions to problems with superficial changes using the same structure. For example consider code that finds the lowest number in a one-dimensional array and code that finds the highest number in a one-dimensional array. Both solutions have the same code structure and very similar algorithmic logic. In essence, a small change in syntax (switching less than operator to a greater than operator) is required.

2. *Glued Isomorphic Tasks*

Multiple schemas for programming tasks that have been solved before are combined and/or adapted. In this case the *target* does not have the exact same underlying solution rationale as the *transfer sources* but the transfer *target* task has sub goals that are the same as the goal or sub goal of each *transfer source*. More than one *source* is *transferred*. For example, a program that reads input integers and outputs their average, requires the retrieval of the existing abstract solution schema for

summing of integer numbers, and the abstract solution schema for counting the number of integers, and the merging of these two plans – which in turn should lead to a new abstract cognitive schema.

3. *Far Transfer Tasks*

The *target task* requires the use of different programming concepts and code structure than the *source task*. For example, code that calculates the summation of elements stored in a one-dimensional array and the program code that calculates the summation of elements stored in a two-dimensional array, ignoring that the first problem is solved using the concept of a one-dimensional array and the second problem is solved using the concept of a two-dimensional array, both problems have identical sub goals in common so that certain sub goals or at least relevant sub goals can be transferred; set the gather variable to zero, and then increment that value based on the new value read from an array. Thus, identifying identical sub goals involves identifying multiple abstract features (or “schemas”), which can be transferred.

It should be noted that as for other classification systems, such as the Bloom and SOLO taxonomies, an intimate knowledge of the programming courses and programming tasks students which constitute the participants prior knowledge is required in order to classify the tasks.

The tasks designed for this research were classified according to this three class degree of knowledge *transfer* taxonomy and the results of this classification are given in Table 5.9. The classification was undertaken with the assumption that the students programming knowledge had been acquired during their course of study in P1 and P2. The determination of prior knowledge and was dependant on the timing of the task delivery in the think aloud sessions and the topic delivery and programming exercises in these programming courses. The tasks were classified by the researcher and two independent experts – where disagreements existed the classifiers discussed the process in order to reach a consensus.

5.6. An Overview of the Tasks

Nineteen code writing tasks were created for this research. These code writing tasks were classified into five categories based on the programming **concepts** required to solve the problem and the **context** of the task (either a Robot World or a native Java task (one outside of the Robot World using the standard Java libraries)).

An overview of these tasks and their classification within the difficulty, conceptual and context frameworks is presented in Table 5.9.

*Table 5.9 Overview of the tasks**

Sequences	Tasks	Conceptual knowledge								Difficulty measure		Knowledge transfer classification
		Variable	Selection	Iteration	Method	Array (1D)	Array (2D)	ArrayList	Class	SOLO	WM	
1	Q1	✓		✓						U	18.58	Isomorphic
1	Q2	✓	✓	✓						M	61.03	Glued Isomorphic
1	Q3	✓	✓	✓	✓					R	75.55	Glued Isomorphic
1	Q4	✓	✓	✓	✓					R	75.55	Isomorphic
2	Q1	✓	✓	✓						M	32.56	Far
2	Q2	✓	✓	✓						M	35.47	Isomorphic
2	Q3	✓	✓	✓	✓					R	45.40	Far
2	Q4	✓	✓	✓	✓					R	48.33	Far
3	Q1	✓	✓	✓	✓	✓				R	36.71	Isomorphic
3	Q2	✓	✓	✓	✓	✓				R	38.34	Far
3	Q3	✓	✓	✓	✓	✓				R	39.23	Isomorphic
3	Q4	✓	✓	✓	✓	✓				R	40.02	Far
4	Q1	✓	✓	✓	✓	✓	✓			R	44.96	Far
4	Q2	✓	✓	✓	✓	✓	✓			R	56.72	Far
4	Q3	✓	✓	✓	✓	✓	✓			R	61.71	Isomorphic
4	Q4	✓	✓	✓	✓	✓	✓			R	67.33	Far
5	Q1	✓	✓	✓	✓	✓	✓	✓	✓	R	61.27	Far
5	Q2	✓	✓	✓	✓	✓	✓	✓	✓	R	65.54	Far
5	Q3	✓	✓	✓	✓	✓	✓	✓	✓	R	67.41	Far

* Where:

- U is unistructural, M is multistructural, and R is relational and U is the lowest level of thinking.
- For the writing metric the lower the value the easier the question is.
- The Knowledge transfer classification column gives a classification based on how the schemas required to solve each task need to be used or changed in order to solve the problem using ideas derived from transfer theory. Where isomorphic is the most direct transfer mechanism and by far the least direct, and therefore hardest to accomplish.
- An illustration of the sequences (1-5) is provided on the next page.

The conceptual relationships between the tasks, both inter and intra sequence, are illustrated in Figure 5.1. Table 5.10 provides a summary of the implied operative schemas required to solve each task. Figure 5.2 shows the chronological order in which the questions were intended to be presented to the participants. A detailed discussion of these tasks/questions and task/question sequences is presented in the next section of this chapter.

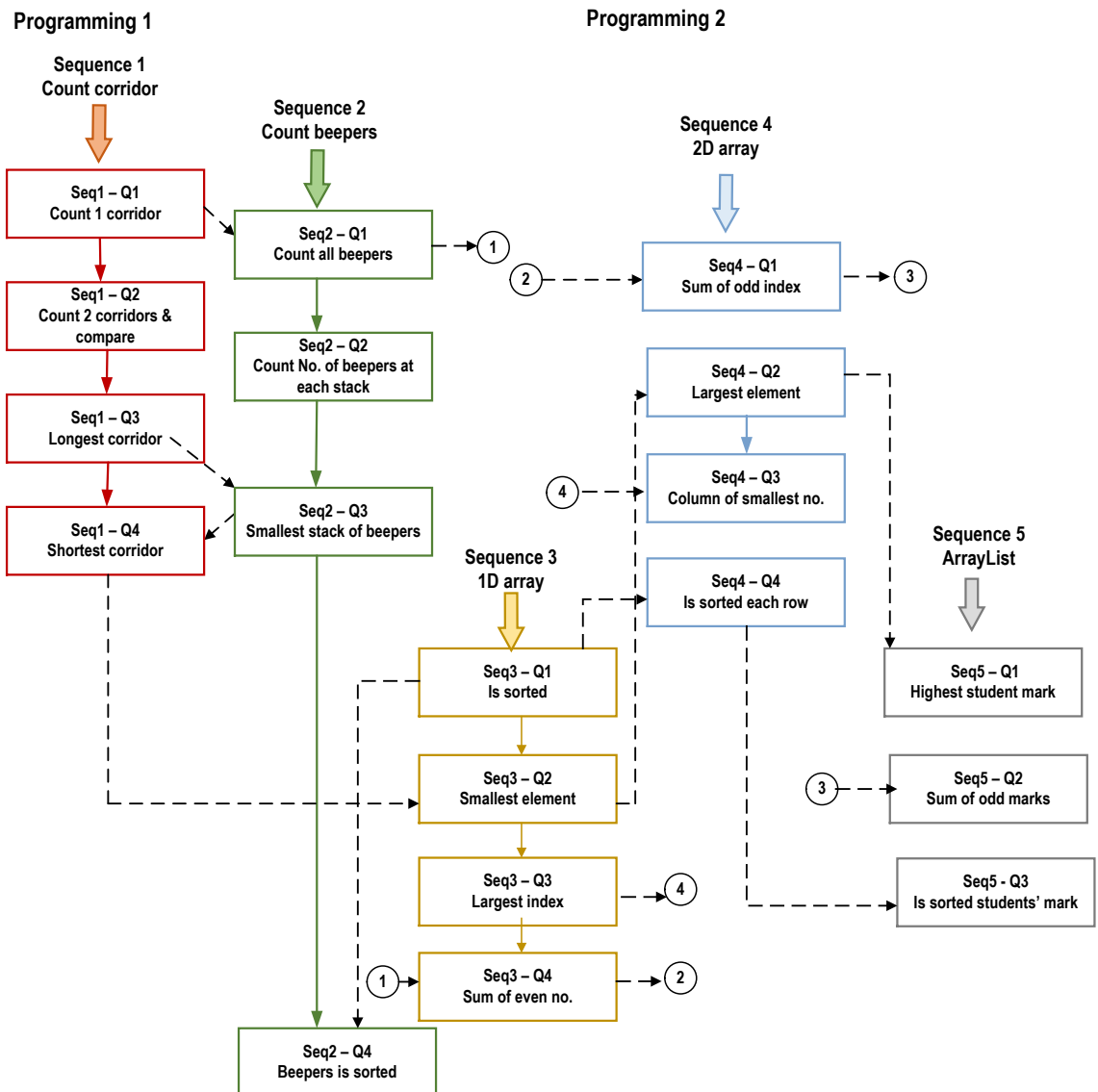


Figure 5.1 The conceptual relationships between the questions

- Questions in each sequence are represented using different box colours.
- Arrows of type \longrightarrow represents a conceptual relationship between tasks in the same sequence
- Arrows of type \dashrightarrow represents a conceptual relationship between tasks in the different sequences.

Table 5.10 Schemas required for solving the questions

Sequence	Question	Schemas required to solve questions
1	Q1	<ul style="list-style-type: none"> • Calculate the length of a single corridor. • Topic knowledge for Java commands used in Table 5.9.
2	Q2	<ul style="list-style-type: none"> • Calculate the length of a single corridor. • Compare two integer numbers (homework assignment). • Topic knowledge for Java commands used in Table 5.9.
3	Q3	<ul style="list-style-type: none"> • Calculate the length of a single corridor. • Compare two integer numbers (homework assignment). • Topic knowledge for Java commands used in Table 5.9.
4	Q4	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq1 – Q3, ○ Or, schema for solving Seq2 – Q3 • Topic knowledge for Java commands used in Table 5.9.
2	Q1	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for picking up all the beepers across a single corridor as well as counting the number of beepers at the first location (homework assignment), ○ Schema for counting the length of the corridor (Seq1 – Q1). • Topic knowledge for Java commands used in Table 5.9.
2	Q2	<ul style="list-style-type: none"> • Schema for solving Seq2 – Q1. • Topic knowledge for Java commands used in Table 5.9.
2	Q3	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq1 – Q3. ○ Or, Schema for solving Seq2 – Q2, and compare two integer numbers. • Topic knowledge for Java commands used in Table 5.9.
2	Q4	<ul style="list-style-type: none"> • Schema for solving Seq2 – Q2. • Schema for solving Seq3 – Q1. • Schema for comparing two integer numbers. • Topic knowledge for Java commands used in Table 5.9.
3	Q1	<ul style="list-style-type: none"> • Schema for solving values in the array is in ascending numerical order (homework assignment). • Topic knowledge for Java commands used in Table 5.9.
3	Q2	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq1 – Q3 ○ Schema for solving Seq1 – Q4, ○ Or schema for solving Seq2 – Q3, • Topic knowledge for Java commands used in Table 5.9.
3	Q3	<ul style="list-style-type: none"> • Schema for solving Seq3 – Q2, • Topic knowledge for Java commands used in Table 5.9.
3	Q4	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq2 – Q1, ○ Or, their existing schema that calculates the summation of all elements in a one dimensional array (homework assignment). • Schema for checking if the number is odd or even. • Topic knowledge for Java commands used in Table 5.9.
4	Q1	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq2 – Q1, ○ Summation of integer numbers (homework assignment), ○ Or, schema for solving Seq3 – Q4.

		<ul style="list-style-type: none"> • Schema for checking if the number is odd or even. • Topic knowledge for Java commands used in Table 5.9.
4	Q2	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq1 – Q3, ○ Schema for solving Seq1 – Q4, ○ Schema for solving Seq2 – Q3. ○ Or schema for solving Seq2 – Q2. • Topic knowledge for Java commands used in Table 5.9.
4	Q3	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq4 – Q2, ○ Or, schema for solving Seq3 – Q3. • Topic knowledge for Java commands used in Table 5.9.
4	Q4	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq4 – Q1, ○ Or, schema for solving Seq2 – Q4, • Topic knowledge for Java commands used in Table 5.9.
5	Q1	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq1 – Q3, ○ Schema for solving Seq1 – Q4, ○ Schema for solving Seq2 – Q3. ○ Schema for solving Seq3 – Q2. ○ Or, schema for solving Seq4 – Q2. • Topic knowledge for Java commands used in Table 5.9.
5	Q2	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq2 – Q1, ○ Summation of integer numbers (homework assignment), ○ Schema for solving Seq3 – Q4, ○ Or, schema for Seq4 – Q1. • Schema for checking if the number is odd or even. • Topic knowledge for Java commands used in Table 5.9.
5	Q3	<ul style="list-style-type: none"> • Either <ul style="list-style-type: none"> ○ Schema for solving Seq4 – Q4, ○ Schema for solving Seq3 – Q1, ○ Or, schema for solving Seq2 – Q4. • Topic knowledge for Java commands used in Table 5.9.



Figure 5.2 *The order in which the questions are presented to the participants*

5.7. Sequence 1 – Counting Corridors

The questions in this sequence focused on counting the length of a Robot World corridor. The participants were required to write code to *count corridor (concept)* using **Robot World (context)**. To find the length of corridors requires counting the number of cells travelled until a wall is encountered. These questions have small incremental increases in conceptual difficulty as determined by SOLO, and the writing metric measure of difficulty. Q1 requires a unistructural response, Q2 a multistructural response, and Q3 and Q4 a relational response. The same pattern of difficulty is seen in the code structure and readability as determined by the difficulty metric (Table 5.9).

Q1: For this question, the participants were provided with a method header. They were asked to complete the method by writing code to count the length of a single corridor. The length of the corridor changes each time the code is run. In order to solve this problem, they needed to move the robot, which was located at the start of the corridor and count the number of cells the robot travelled until it reached the end of the corridor. Examples of Robot World scenarios with a corridor of length 5 and 10 were provided for the participants (Figure 5.3).

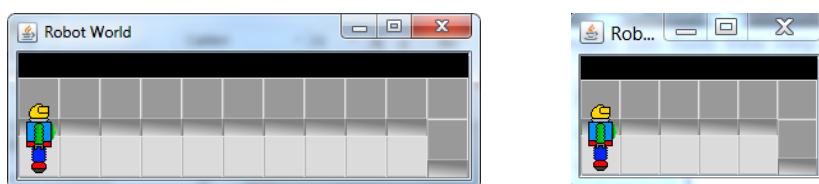


Figure 5.3 The scenarios provided for Seq1 – Q1

The participants had already been given this question as part of their coursework, so it should have been familiar to them. This question is the only question in the research instrument, which students had already seen. The purpose of this question was to check whether or not the participants had an existing cognitive schema for counting the length of the corridors.

Q2: For this question, the participants were provided with the method header. They were asked to complete the method by writing code to compare the length of two corridors and print out a message that either stated the corridors were of equal length or gave the length of the longest corridor. The corridors were always at the same location and were connected in the same way and there were always only two corridors; only the length of the corridors changed. Figure 5.4 shows the three different scenarios given to the students as part of the task description.

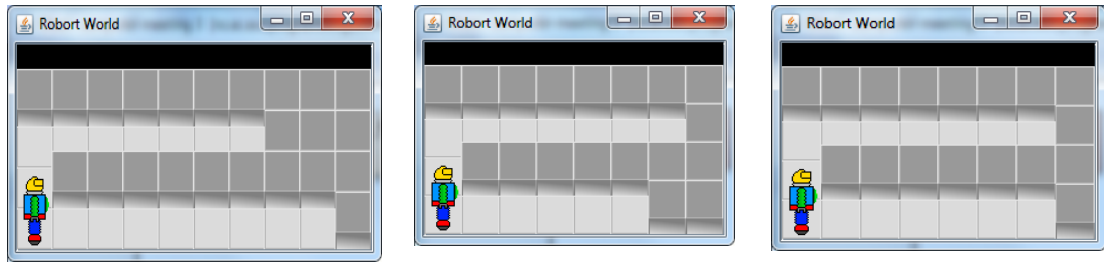


Figure 5.4 Three different scenarios for Seq1 – Q2

In order to solve this problem, it was anticipated that the participants would need to combine the two existing schemas. The first schema was one, which the participants should have already developed during their study in the P1 course. This schema provides a cognitive structure for finding the higher of two integer numbers and printing that number within an appropriate message. The second schema is the one, which was required to solve the previous task (Q1) in this sequence counting the length of a corridor.

Q3: In this question, there are any number (obviously limited by the dimensions of the world) of interconnected n corridors, but they are always connected at the same point (column 0, as shown in Figure 5.5). The length of each of the corridors changes randomly each time the Robot World is created. The participants were asked to write a program that used the robot to count and return the length of the *longest* corridor. The question text provided the participants with the three different scenarios illustrated in Figure 5.5. The main idea behind providing them with the three example scenarios was to suggest to the participants that the number and length of the corridors varied.

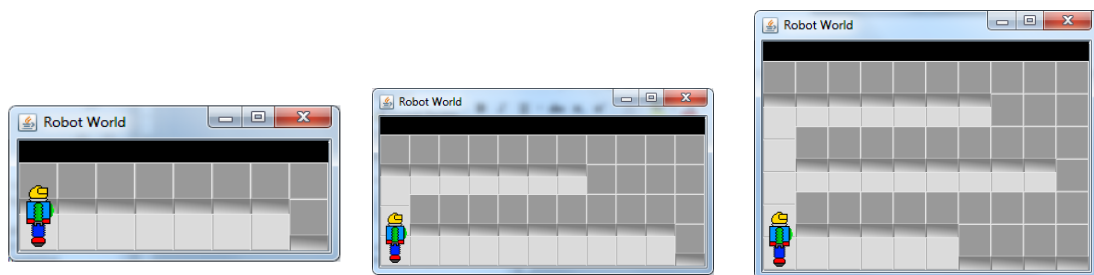


Figure 5.5 Three different scenarios for Seq1 – Q3

While this question would have been new to the participants it should have appeared familiar. It was designed so as to require them to utilise the knowledge that, ideally, ought to have been developed in the process of solving the previous two questions in the sequence. In this case the schemas had to be modified and adapted rather than just simply joined. At the stage that this task was delivered the participants had already been assessed, in the P1 course, on their ability to use the existing Robot World methods and to write methods with a return statement.

In order to develop a fully generalisable solution a loop should be written to count the length of the first corridor and this value stored in a most wanted holder variable. The process then continues and for any further corridors encountered then the length is counted. If the length is longer than that stored in the most wanted holder variable, then the most wanted holder variable is updated to be the current corridors length. Such an approach removes redundancy, which exists in less sophisticated potential solutions such as counting each corridors length and then comparing the lengths.

Q4: Again this question builds on the cognitive schemas which the participants should have developed during the generation of solutions to the previous questions in the sequence. In this case, the question is also reliant on the participant having successfully solved Q3 in sequence 1. In this question, there are any number of interconnected n corridors, always connected at the same point (column 0, as shown in Figure 5.6). The length of each of the corridors changes randomly each time the Robot World is created. The participants were asked to write a program that calculated the length of the *shortest* corridor, and then return the length of that corridor.

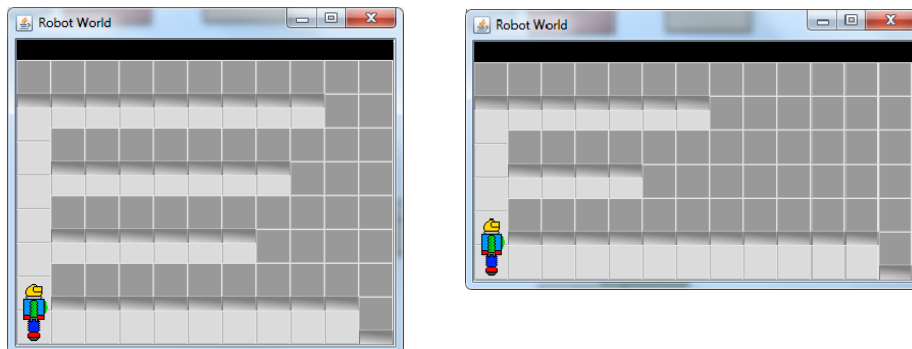


Figure 5.6 Two different scenarios for Seq1 – Q4

To solve this question, the participants were expected take one of two approaches:

1. Retrieve the schema developed in the previous question in the sequence and hence adapt longest to shortest. The only change necessary to the code for the longest should be in the comparison of the corridor lengths which changes from greater than to less than in order to find the shortest corridor.
2. Retrieve a similar schema developed when solving Q4 in Sequence 2. Q4 asked the participants to write code to find the *smallest* stack of beepers. In this case, the participants would be more focused on the notion of finding a lower/smaller/shorter value rather than on the process of counting corridors. To choose this approach they would need to be able to appreciate the differences in managing the robot movement for counting the length of a corridor vs. counting the number of beepers.

It is likely therefore that in this case they would need to select the relevant ideas from the schema for counting corridors, and finding the smallest stack of beepers.

5.8. Sequence2 – Counting Beepers

The questions in this sequence focused on counting the number of beepers. This task required code to **count beepers (concept)** using **Robot World (context)**. These questions were designed to impose small incremental increases in conceptual difficulty. The writing metric and the SOLO classification both show that the tasks are progressively more difficult through the sequence. The SOLO classification of the questions also suggests that the first two questions require a lower level of thinking (a multistructural response) than the others, which require the participants to be able to think relationally in order to generate a working, fully correct, solution.

Q1: After solving (Seq1 – Q1) counting the length of a single corridor, the participants were asked to write code to print the number of beepers from all the beeper stacks in a corridor, in other words calculate the total number of beepers in a single corridor. They were provided with the method header. From run to run the length of the corridor varies as does the number of stacks encountered and the position of the stacks (Figure 5.7). In order to solve this problem, participants are expected to have either:

- Started with an iteration statement that counted the number of beepers at a first location, followed by nested iteration statements that allowed the robot to move and count the beepers at each location, keeping a running total, until it reached the end of the corridor.
- Or, started with nested iteration statements that allowed the robot to count the number of beepers at its location and then move while counting beepers, followed by an iteration statement for counting the beepers at the last location.

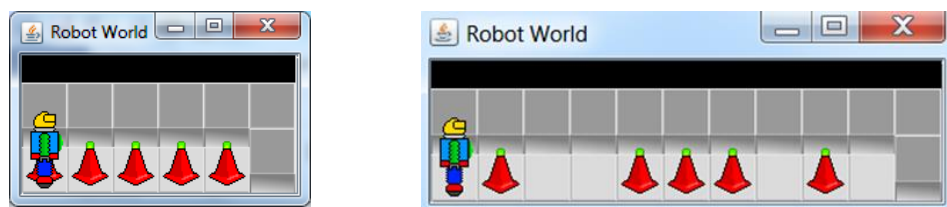


Figure 5.7 The scenarios provided for Seq2 – Q1

The lecturer had already taught the participants code examples which involved the robot picking up one of and all of the beepers at each stack across a single corridor during a lecture so, assuming the participants were paying attention, this problem should have been relatively familiar. In related homework, the students had been asked to write code, which

counted the beepers in the first cell of the corridor that had a stack of beepers. Merging a schema for picking up all the beepers in the corridor while keeping a running total of beepers picked up might be one way of solving this question. Alternatively, the differences between counting the length of the corridor and counting the number of beepers could be recognised and this knowledge applied to reach a solution to the problem.

Q2: The participants were provided with a method header and asked to complete a method to print the number of beepers in each beeper stack across a single corridor. Therefore, there would be a printed statement for every beeper stack encountered. The scenario examples provided were the same as those for Q1 (Figure 5.7). To answer this question, it was expected that the participants would tailor their existing schema for counting the number of beepers from all beeper stacks in a corridor.

One way to solve this question would be to conserve the code structure, which the participant developed for Q1. The main difference between Q1 and Q2 is that in solving Q2 the participants should call the print method to print the value of the gather variable (the variable storing the running total) after counting each stack and then reset that variable to zero before counting the beepers in the next stack. This difference meant that this question had a higher writing metric and therefore was for the purpose of this instrument considered to be more difficult than Q1.

Q3: The participants were asked to solve this question after solving Seq1 – Q3 and Q2 in this sequence. They were presented with a scenario in which there were a number of beeper stacks each one containing a different number of beepers. The participants were asked to write a method to make the robot move along the corridor and count the number of beepers at each beeper stack and return the number of beepers in the smallest stack (Figure 5.8 shows the scenarios provided). Obviously, as indicated in the question, the number of beepers in each stack is not fixed even though the scenario images do not show this aspect of the problem.

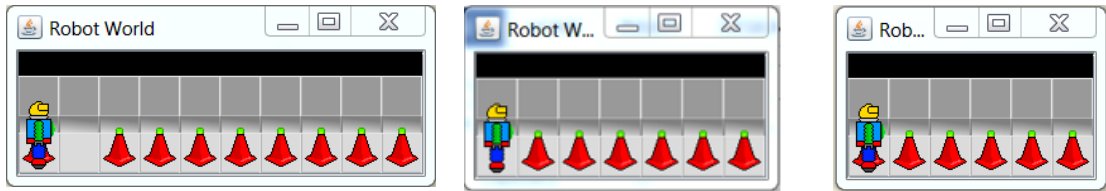


Figure 5.8 Three different scenarios for Seq2 – Q3

To solve this question, the participants were expected to take one of the following approaches:

1. Understand or recognise the differences between counting the length of the corridor and counting the number of beepers, and then apply that knowledge.
2. Modify their existing schema for counting the number of beepers at each beeper stack across a single corridor as well as a schema to compare two integer numbers.

Q4: The participants were asked to solve this question after solving Q3 and Seq3 – Q1. The question involves writing a method that makes a robot walk through a single corridor, count the number of beepers at each beeper stack, and return true if the beeper stacks are in order of an increasing size, otherwise it should return false.

Three scenarios were provided for the participants as shown in Figure 5.9 and they were told that if they ran the first and last scenario their code should return false, otherwise for the second scenario their code should return true.

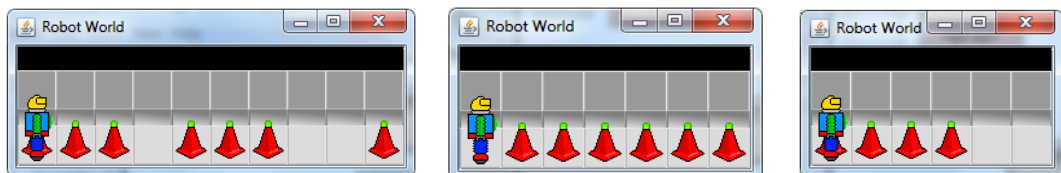


Figure 5.9 Three scenarios for Seq2 – Q4

This question and Seq3 – Q1 have the same sub goals; both of them require a sorting algorithm. However, they involve different concepts and task contexts.

To solve this question, the participants were expected to take one of these approaches:

- Modify existing schemas for the sorting algorithm (as developed solving Seq3 – Q1, which involved sorting integers in a one-dimensional array), and then somehow adapt that schema to the Robot World,
- Modify and combine schema for counting the number of beepers at each beeper stack across a single corridor and for comparing two integer numbers.

5.9. Sequence3 – One-Dimensional Array

The programming tasks in sequence3 involve a **one-dimensional array (concept)** using **native Java task (context)**. In the lab and homework assignments all the students, including the participants in this study, learnt about one-dimensional arrays. All the questions in this sequence are relational but becoming increasingly difficult according to the difficulty metric (Table 5.9).

Q1: This question asks the participants to write a method called `isSorted()` that takes an integer array as an input parameter, and returns true if the values in the array are sorted in descending numerical order. Eight unit test scenarios were provided for the participants. One of these scenarios checked the method parameter against Null array pointer exception.

In a homework assignment, the participants had been asked to write a method that took an integer array as an input parameter, and returned true if the values in the array were in ascending numerical order.

Q2: This question required writing a method `findSmallest()` that returns the smallest element in an array. The array is provided as a parameter to the method. Four unit test cases were provided for the participants. The first three scenarios consisted of positive and negative integer numbers, whereas the last scenario consisted of the positive integer numbers only.

This question and the questions solved in previous meetings, for example, calculate the longest corridor (Seq1 – Q3), the shortest corridor (Seq1 – Q4), and the smallest stack of beepers (Seq2 – Q3) are similar; they all need a most recent holder variable and require comparison of the stored value against a newer calculated value. However, none of the previous questions involved negative numbers. The main challenge for the participants would be in adapting their existing schemas to cope with negative numbers.

Q3: This question is very similar to Seq3 – Q2. The core differences are the adaption from less than to greater than comparisons and that comparisons are based on the value stored but the index of the array element is the value that must be held and returned. A method called `findLargestIndex()` that takes an array of integers and returns the index of the largest element in the array should be written. Four unit test cases were provided for the participants.

To solve this problem, the participants were expected to make use of their existing schema for finding the smallest number in an array and make minor changes to the code structure.

Q4: This question requires writing a method called `foundSum()` that takes an integer array and returns the sum of any even numbers stored in the array. A homework assignment had been undertaken which sums all the numbers in an array. This question is considered to be more complex than the homework one because a selection Java statement needs to be integrated into the schema for summing numbers in an array to check whether or not each number in the array is even. Four unit test cases were provided for the participants.

As part of practicing and coaching in the think aloud sessions, the participants had been evaluated to ensure that they were able to check if an input number is odd or even. In addition, in previous sessions as well as in the lab and homework assignments, all the participants had been evaluated to ensure that they were able to write procedures that calculate the summation of all the beepers in a single corridor, and the summation of all elements in a one-dimensional array.

5.10. Sequence4 – Two-Dimensional Array

This is the first task in the sequence of the programming tasks for the P2 course. The participants at this stage, in theory, should have a good knowledge of all programming tasks covered in the preceding sequences. The programming tasks in this sequence involve **two-dimensional array (concept)** in a **native Java (context)**. None of the questions had been seen before. Unit tests were provided along with the base code for all of the questions in this sequence.

The questions were designed to be similar to problems the participants had already encountered in the study. All the questions in this sequence are relational but becoming increasingly difficult according to the difficulty metric (Table 5.9).

Q1: Participants were asked to write a method called `foundSum()` that takes a two-dimensional array of integers and returns the sum of all the elements stored in odd indexed rows of the array.

At this stage, the participants had had plenty of practise programming tasks that request from them to count beepers and calculate the sum of some integer numbers. The participants at this stage should also be aware of the differences between the value of an element in an array and an index of an element in an array.

Q2: This question asks participants to write a method called `findLargestElement()`, that takes a two-dimensional array of integers and returns the largest element in a two-dimensional array. Three unit test cases were provided.

This question and questions solved in previous meetings, for example, calculate longest corridor (Seq1 – Q3), shortest corridor (Seq1 – Q4), smallest stack of beepers (Seq2 – Q3), and smallest element in a one-dimensional array (Seq3 – Q2) are similar in terms of schemas.

Q3: Participants were asked to write a method called `findSmallestIndexColumn()`, that takes a two-dimensional array of integers and returns the index of the column containing the smallest element in the array. Three unit test cases were provided.

This question was similar to Seq4 – Q2 as the index of the container of the smallest element was the value, which had to be returned rather than the element itself. But in contrast because this is a two-dimensional array the index is that of the column rather than of the element. To solve this problem the participants were expected to either:

- Use their existing schema for finding the largest number (Seq4 – Q2),
- Or, tailor their existing schema for solving the question (Seq3 – Q3) using a one-dimensional array and restructuring that schema to accommodate the two-dimensional array.

Q4: This question asks the participants to write a method called `isSortedElementRow()`, that takes a two-dimensional array of integers and prints out a message that states whether or not the values in each row are in ascending numerical order.

In previous sessions, participants had solved several related questions that required a schema for checking whether or not things are in an ordered sequence, for example checking if the values in a one-dimensional array were in descending order (Seq3 – Q1), and checking if beeper stacks were in order of increasing size (Seq2 – Q4).

5.11. Sequence5 – ArrayList

The programming tasks in this sequence use **ArrayList (concept)** in **native Java (context)**. As for sequence 4, the questions in this session were in order of increasing difficulty, measured using the difficulty metric value, and all were posed at the same SOLO level, which meant that to solve the problem the participants needed to be operating at a relational level.

The code base for these questions included a called `StudentDatabase` that stores an `ArrayList` of `Student` objects. The `Student` class consisted of a single constructor, which

assigned values to fields storing a name and an array of marks field. Student marks are within the range of 0-100. Unit tests were provided along with the base code for all of the questions in this sequence.

Q1: Participants were asked to write a method called `highestStudentMark()`, and print the highest mark and name of every student in the `StudentDatabase`. Participants by this stage in the study should have had a well-developed schema for finding the largest number and for iterating through an array, the biggest challenge should have been adapting these schemas to the relatively new concept of an `ArrayList` data structure.

Q2: This question asked participants to write a method called `sumOfOddMarks()`, that prints out a message that reports the sum of all the odd marks for each student.

This question was designed to build on schemas already familiar to the students such as: summing of integer numbers (homework assignment), all even numbers in a one-dimensional array (Seq3 – Q4), and the summation of numbers stored in odd indexed rows of a two-dimensional array (Seq4 – Q1).

Q3: Participants were asked to write a method called `studentsMarksSortedEach()`, and print out a message that states for each student whether or not their marks are stored in ascending order. The participants had been evaluated solving different sorting questions, for example, Seq3 – Q1, Seq2 – Q4, and Seq4 – Q4. These questions are superficially similar; all of them have an identical sub-goal in common (using a sort algorithm). All of these questions were solved using different concepts and contexts.

To solve this question, participants were required to take their knowledge and skills, which ideally had been developed in solving the sorting algorithm, and modify that knowledge and those skills into an `ArrayList` concept using native Java commands.

5.12. Summary

This chapter has presented the questions, which form the instrument for this research. Each question was placed in a delivery order that was dependent on a concept building sequence and on a measure of difficulty. The sequence and timing of the delivery of the questions has also been outlined. Because it was expected that participants would progress at different rates, and find different aspects of the tasks difficult, the timing was to be used as a guide only. The timing as presented in this chapter reflects the preparedness of the participants in terms of the programming course so that all the concepts used in the research questions would have already been encountered in their study and been practiced in lab classes. The research is designed such that the student would be able to solve dependent questions, or a variant of that question, prior to progressing to the next question in the study. It was expected that all but the best performing participants were unlikely to complete all of the instrument's questions. The next chapter provides the results of illustrative think aloud and retrospection interviews, which utilize the questions in this research instrument.

Chapter 6. Think Aloud: Encoding and Interpretation

6.1. Introduction

In this chapter think aloud data, the transcriptions of that data and a preliminary analysis of that data is presented. The data was transcribed and encoded using the method and coding schema presented in Chapter 3.

The information provided in this chapter is for a select set of the participants. An initial group of 21 students studying P1 volunteered to be part of this research. Two of these students had previous programming experience and were therefore excluded from the study. Nineteen students attended the initial think aloud training sessions. After being informed of what would be required in more detail and experiencing practice think aloud interviews five participants withdrew consent because they anticipated that they would not have enough time to fully commit to the study. Of the remaining 14 students, 13 were selected to take part in the research. These participants were the ones who demonstrated at least a minimal capability to think aloud while solving the simple training programming problems. Of these 13 participants one withdrew from the study after the second session. One other participant, despite the best efforts of the researcher, made very slow progress and was unable to think aloud or reflect on their thoughts during retrospective interviews. Although the researcher continued to work with this participant throughout their first semester the participant did not provide any useful data and therefore is their contribution is not included in this encoding and analysis phase.

Seven of the eleven remaining participants from P1 continued attending the meeting sessions during P2 and thus provided a full academic year of data⁹.

Because of space limitations the entire set of transcriptions, encodings and analysis of the think aloud sessions for all seven participants has not been included in this thesis. Instead the data for four participants is presented. In order to provide a useful picture the four participants selected included the two who performed best in this study (Andre and Luke) and the two who performed worst (Kasper and Matthew). The tasks presented in this chapter are the ones which are used in Chapter 7 to map the observations, reported in detail this chapter, with the constructivist cognitive theories discussed in Chapter 2. The

⁹ See Appendix H for a summary of the think alouds including hours of video recorded and diagrams outlining the progression of learning for each participant.

detailed data, encoding and interpretation for the remaining tasks are provided in Appendix A.

Data is grouped by participant. A brief overall summary of the participant's general approach to programming and performance is provided. The data is then grouped by question in the order in which the questions were presented to the participant. The information for each programming task/question is organised as follows:

- A table presenting the encoded think aloud and retrospective interview sessions. Detailing: programming behaviours, emotions, strategies used, associated activities, interview interventions, and timing with respect to weeks on course. Additionally any relevant observations from the participants attempt(s) on earlier tasks are included. Time on task is the time the participant spent on the task and does not include time spent assisting the participant (scaffolding) or the time spent on the retrospective interview.
- The transcribed think aloud data is then presented in temporal sequence along with images of the participants' code and any relevant or interesting notes/annotations the participant made. The code itself is annotated to illustrate the temporal order of the relevant programming activities. Some interpretations are made related to the researchers observations, think aloud data and retrospection interview data.
- The transcribed data from the retrospective interviews is then documented where relevant. In cases where the participant received quite a large degree of scaffolding during the code writing or where participants were able to easily solve the problem there was no need for a retrospective interview. The term interviewer is the used to refer to the researcher and author of this thesis, Nadia Kasto.

6.2. Andre's Think Aloud Sessions

6.2.1. Summary

Andre did not seem to move outside of his ZPD during the course of this research. In general it was observed that Andre consistently planned his solution prior to coding. He always attended the sessions on time and never cancelled a meeting. During the think aloud sessions, Andre showed independence with respect to the tasks and made considerable efforts to solve the programming tasks on his own without assistance. He also made use of the tools he had been taught when trying to solve the tasks by tracing his code, and reading and understanding unit test outputs and the unit test code. Andre was a high performing student and was in the first quartile for both the P1 and the P2 course.

6.2.2. Counting the Length of One Corridor (Seq1 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Sequential	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging, and hand gestures
	<i>Unit test</i>	
	<i>Time on task</i>	6 minutes and 13 seconds
	<i># of compilation</i>	3
	<i># of execution</i>	2
Intervention	“General prompt” scaffolding – provided on request	
Timing	Week four of the P1 course	
Important observations with respect to prior sessions		

Data

1. Think aloud:

Andre started to code his solution without hesitation or verbalisation. Andre initially created and set a gatherer variable to zero. He then wrote a WHILE-loop that moved the robot and counted the number of moves. The number of moves is the length of corridor. Finally, he added a PRINT-statement to print to the console the computed length of the corridor. Andre compiled his code which generated a syntax error – he had omitted the closing curly brace (}) for the WHILE-loop. After a short pause Andre asked for help.

2. Scaffolding:

The interviewer suggested that Andre compare the number of open and closed brackets. Andre started to count the number of opening and closing brackets and easily fixed his code.

3. Think aloud:

On running the unit tests they all failed, Andre realised there was an error but he did not read the error messages generated by the unit tests. Instead he moved his finger along the Robot World corridor displayed on the screen tapping and counting the corridor cells.

“One, two, three, four, five, I do not count the first location so I need to set it to one”

Seeing the Robot World image helped Andre fix his code – he set the initial value of the gatherer variable to one.

4. Retrospection:

The following is part of the conversation between interviewer and Andre:

Interviewer: *“Have you seen this question before?”*

Andre: *“No, I saw counting the beepers, so the beepers come with zero, and the initialise should be set to zero to count the beepers”*

Interviewer: *“Did you read the syntax error?”*

Andre: *“Yep”*

Interviewer: *“Did you understand what the message meant?”*

Andre: *“I did not think so [pause]. Most of the time while I was doing the homework assignment, I grew frustrated [pause] when I saw the messages shown below the screen.”*

Interviewer: *“Why were you frustrated?”*

Andre: *“Because, um, I think because I’m not always [pause] not always successful in carrying out the work.”*

At the end of the session, the interviewer explained to Andre, the usefulness of syntax error messages, what the message he saw meant in this task and how to interpret it.

6.2.3. Longest Corridor (Seq1 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Confused	
Strategies	Stepwise design	
Activities	<i>Planning</i>	Verbalise
	<i>Tracing</i>	Mental tracing, visual debugging, and hand gestures
	<i>Unit test</i>	
	<i>Time on task</i>	40 minutes and 13 seconds
	<i># of compilation</i>	3
	<i># of execution</i>	3
Intervention	1. Hint scaffolding –interviewer intervention 2. “General prompt” scaffolding – provided on request	
Timing	Week six of the P1 course	
Important observations with respect to prior sessions	Andre solved the comparing the length of two corridors task (see Appendix A) without any difficulty. He had practiced in isolation counting the length of a corridor and comparing the value of two integers, he just needed to concatenate these two programming plans. In the case of this question the same two programming plans were required but they had to be combined by merging and nesting rather than concatenating and he had difficulty doing this. There were additional robot navigation tasks as the robot had to be moved and orientated in order to get to the second corridor before the second corridor’s length could be counted.	

Data

1. Think aloud:

Andre began by reading the problem. Andre took 4 minutes and 19 seconds to formulate a plan for solving this question. He verbalised the plans as follows:

“I need to compare the numbers, the number of corridors changes each time it is created, so I need to find it out, whether there are more than one corridor, I need to compare the length of corridors, the first situation there are one corridor, so move the robot to the end of corridor, and count the numbers, and turn robot back, and to check whether there is wall ... the problem how to compare, ah, the problem how to memorise the long of corridor, this is the longest ah [long pause] ... how to compare, three is not enough is keep changing go to the first, go to the second corridor, [pause] but if there is more, four corridors, how to assign the integers, to assign the variables, what I can do.”

Andre read the question again before he attempted write a solution. He then began writing the code line by line. The first section of code mapped to his existing schema for counting the length of the corridor. He then wrote the commands to reorient the robot to face west and return to the start of the corridor (Figure 6.1, step1). Andre hesitated over the number of left turns required to allow the robot’s orientation to face north. This move was required to be in the correct direction to move to the second corridor. Andre started to read part of his code to verify the robot direction:

“After counting the length of the first corridor, after this the robot facing wall, turn robot left twice, turn left, turn left, move back, move back to the first location and it should be facing that way [moving his hand to robot direction], facing to the west, so he [the robot] should turn left.”

After a short pause, Andre started to write an IF-block that allowed the robot to check the existence of the next corridor and move to that position (Figure 6.1, step2). At this stage the code was incorrect but Andre was unaware of the logical error.

After another short pause, he verbalised:

“How I should make it automatically check it each time?”

He then enclosed the existing code inside a new WHILE-loop (Figure 6.1, steps3A & 3B) and re-read his code:

“We need while loop to make it. While not facing wall, so the robot moving, and after he went to the next line, this while is still work”

After a short pause, he verbalised:

“Now we need, how to compare this, only we have one variable, so I can check whether, the first one is bigger than this one.”

Andre had trouble working out how to store the length of more than one corridor in a way that would allow him to compare the lengths and work out which was the longest. Andre spent about 2 minutes and 20 seconds re-reading his code. The interviewer felt that Andre reached a dead-end and therefore intervened.

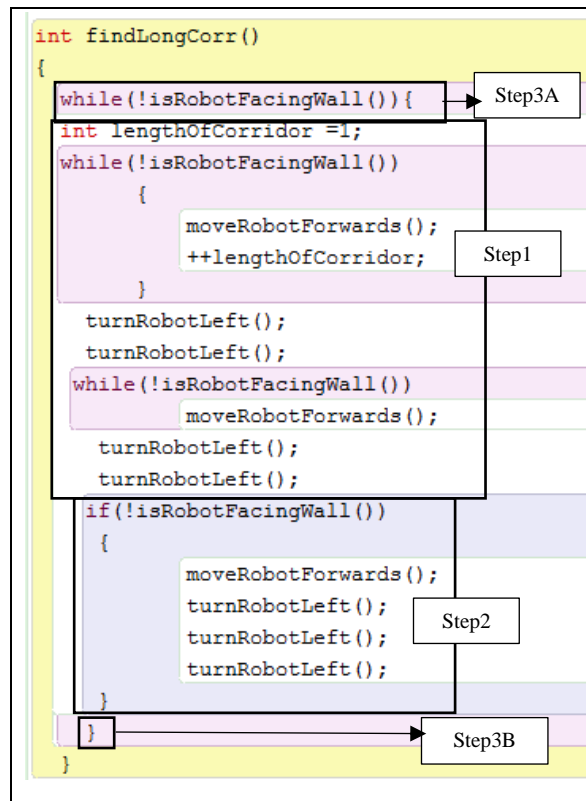


Figure 6.1 Andre’s first screen image for the longest corridor

2. Scaffolding:

The interviewer suggested to Andre that he might create as many variables as he needed to store the value of the variables (Hint scaffold).

3. Think aloud:

Andre started to update his code as shown in Figure 6.2, he defined another most wanted holder variable `thelongestCorridor` and a new gatherer variable `lengthOfPresentCorridor` (step5), he deleted a gatherer variable `lengthOfCorridor`, as shown in step6, etc. This process and the redundant code generated indicates Andre’s fragile knowledge of the way in which integer variables store data and how variable comparison works (as shown in Figure 6.2 step4 and step8). Andre started to read his code from the third WHILE-loop, at the same time examining the example Robot World images provided in the question specification.

“While not facing wall, so the robot’s moving, and after went to the next line, this while is still work”

Finally Andre verbalised:

“I’m confused how to compare these two?”

```
int findLongCorr()
{
  int lengthOfPresentCorridor=1;
  int thelongestCorridor;
  while(!isRobotFacingWall()){
    int lengthOfCorridor =1;
    while(!isRobotFacingWall())
    {
      moveRobotForwards();
      ++lengthOfPresentCorridor;
    }
    int lengthOfLargestCorridor = lengthOfPresentCorridor;
    if (lengthOfPresentCorridor>lengthOfLargestCorridor)
      thelongestCorridor=lengthOfPresentCorridor;
    else
      thelongestCorridor=lengthOfLargestCorridor;
    turnRobotLeft();
    turnRobotLeft();
    while(!isRobotFacingWall())
      moveRobotForwards();
    turnRobotLeft();
    turnRobotLeft();
    if(!isRobotFacingWall())
    {
      moveRobotForwards();
      turnRobotLeft();
      turnRobotLeft();
      turnRobotLeft();
    }
  }
}
```

The code is annotated with several boxes and arrows:

- Step5 - Add**: Points to the initialization of `lengthOfPresentCorridor` and `thelongestCorridor`.
- Step6 - Delete**: Points to the initialization of `lengthOfCorridor` inside the first while loop.
- Step7 - Update**: Points to the increment of `lengthOfPresentCorridor` inside the inner while loop.
- Step4 - Add**: Points to the assignment of `lengthOfPresentCorridor` to `lengthOfLargestCorridor`.
- Step8 - Add**: Points to the conditional assignment of `thelongestCorridor` based on the comparison.

Figure 6.2 Andre’s second screen image for the longest corridor

4. Scaffolding:

The interviewer redirected Andre (“General prompt” scaffold):

Interviewer: “If you think about the variables, you have one to store the value of the longest corridor encountered so far. Also, you have one variable that has the length of the current corridor. You want to know what the length of the longest corridor is. So you need to check if the length of the longest corridor is greater than the present corridor.”

Andre asked for pen and paper and started doodling (Figure 6.3). Andre’s drew three corridors and noted the length, he then drew boxes to represent the holder variables for the present (current) corridor and the longest corridor and then traced to work out how the comparison should work.

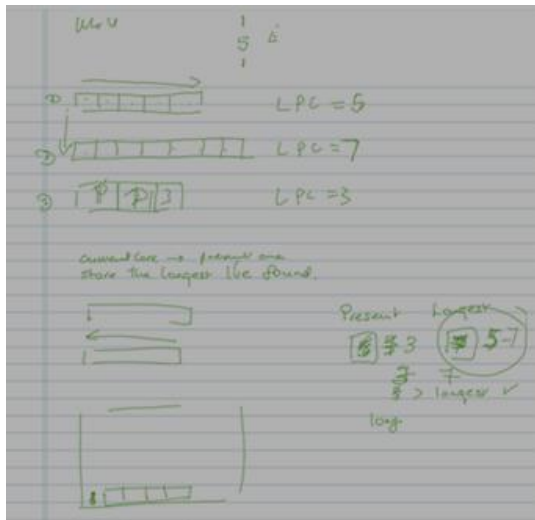


Figure 6.3 Andre's doodle for the longest corridor

5. Think aloud:

Andre corrected his code (see Figure 6.4 (left)) and then he started to mentally trace his code:

“Present equal one, longest equal zero, while not facing, last equal one, present equal one, the length of first corridor ah five greater than one longest one else, no I think I should [pause] delete else”

<pre> int findLongCorr() { int lengthOfPresentCorridor = 1; int theLongestCorridor = 0; while (!isRobotFacingWall()) { int lengthOfLastCorridor = lengthOfPresentCorridor; lengthOfPresentCorridor = 1; while (!isRobotFacingWall()) { moveRobotForwards(); ++lengthOfPresentCorridor; } if (lengthOfPresentCorridor > lengthOfLastCorridor) theLongestCorridor = lengthOfPresentCorridor; else theLongestCorridor = lengthOfLastCorridor; turnRobotLeft(); turnRobotLeft(); while (!isRobotFacingWall()) moveRobotForwards(); turnRobotLeft(); turnRobotLeft(); if (!isRobotFacingWall()) { moveRobotForwards(); turnRobotLeft(); turnRobotLeft(); turnRobotLeft(); } } return theLongestCorridor; } </pre>	<p>Step1</p> <p>Step2</p> <p>Step3</p>	<pre> int findLongCorr() { //int lengthOfPresentCorridor = 1; int theLongestCorridor = 0; while (!isRobotFacingWall()) { //int lengthOfLastCorridor = lengthOfPresentCorridor; int lengthOfPresentCorridor = 1; while (!isRobotFacingWall()) { moveRobotForwards(); ++lengthOfPresentCorridor; } if (lengthOfPresentCorridor > theLongestCorridor) theLongestCorridor = lengthOfPresentCorridor; turnRobotLeft(); turnRobotLeft(); while (!isRobotFacingWall()) moveRobotForwards(); turnRobotLeft(); turnRobotLeft(); turnRobotLeft(); if (!isRobotFacingWall()) { moveRobotForwards(); moveRobotForwards(); turnRobotLeft(); turnRobotLeft(); turnRobotLeft(); } } return theLongestCorridor; } </pre>	<p>Step4</p> <p>Step5</p>
--	--	--	---------------------------

Figure 6.4 Andre's third and final screen images for the longest corridor

He realised that in some cases he would not find the longest corridor because he was making a pairwise comparison. For example if there are three corridors he compared the first with the second and stored the longest of the two corridors in his `theLongestCorridor` variable then compared the second with the third and whichever was longer (the second or the third) overwrote the value stored in `theLongestCorridor`. This meant that if the first corridor was the longest it would always be overwritten by the next longest corridor. Andre updated his code (see Figure 6.4(right), steps1→3). Finally, Andre tested his solution and found two errors related to the robot's orientation and moving to the next corridor (as shown in Figure 6.1 – steps1→2). He was able to fix these bugs in his code by running the unit tests and follow the code executing by watching robot move in the Robot World window (software scaffolding) (see Figure 6.4(right), steps4 →5).

6. Retrospection:

Andre found it difficult to remember exactly how he solved the question. This may have been because he spent a lot of time solving the programming task or may have been because he floundered so many times during the problem solving session.

The following is a conversation between the interviewer and Andre:

Interviewer: *“Had you seen this question before?”*

Andre: *“No”*

Interviewer: *“Have you seen a question similar to this question before?”*

Andre: *“Nope [pause], but many times I solve questions about counting corridors and beepers.”*

Interviewer: *“What was the most difficult part when solving this question?”*

Andre: *“Comparing the length of corridors and repeating the process.”*

At the end of the session, the interviewer gave Andre feedback about the quality of his code and how he could further develop it and the interviewer explained to Andre the mistakes he made when using and comparing the variables. Andre and the interviewer worked on writing an improved solution to the problem before closing the session.

6.2.4. Shortest Corridor (Seq1 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Surprised	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing, and visual debugging
	<i>Unit test</i>	Read the unit test message
	<i>Time on task</i>	53 minutes and 16 seconds
	<i># of compilation</i>	4
	<i># of execution</i>	4
Intervention	None	
Timing	Week eight of the P1 course	
Important observations with respect to prior sessions	Andre previously solved an isomorphic question (the longest corridor). He had difficulty recalling how he had solved the longest corridor task but he was able to recall the fact that he needed to define a most wanted holder variable. The code quality of his solution for the longest corridor question may have hindered Andre's ability to transfer knowledge. He used initially used an incorrect pairwise comparison and in the end the longest corridor code was more complex than necessary and contained redundancy. Andre took a long time to solve the longest corridor task and was focused on the aspects which caused him difficulty rather than the overall purpose of the code, which may have made it difficult for him to transfer his knowledge to this new task.	

Data

1. Think aloud:

Andre saw the connection between the longest and shortest corridor tasks:

“So it is similar to last meeting, it was find out the longest, now it shortest, I think it is basically ah the same, ah, so first to identify the method called ah, we do not need return value. First as I remembered, we need to have ah, we need to have two, set up two integer variables to have comparison.”

Andre started problem solving with a mistake in the method definition. He assumed that the method did not return any value, he discovered this issue later. He then wrote two variable definitions, the gatherer variable `presentRow` for storing the length of the corridor, and the most wanted holder variable `shortestRow` for the storing the length of the shortest corridor. Andre initialised both of them to zero. Andre repeated his earlier mistake made when solving Seq1 – Q1 and Seq1 – Q2 and initialised the gatherer variable to zero. He realised his mistake directly after he defined the most wanted holder variable and fixed it. Andre then proceeded to write the code which allowed the robot to compute the first corridors length and return back and reorient and move to the start of the next corridor. He repeated the same mistake in robot orientation as he had made when solving the longest corridor task. He tested his code and watched the robot moving on the screen,

he then fixed his mistake (see Figure 6.5 (left) for Andre's code up until this stage). As shown in Figure 6.5 (left), Andre was able to use methods.

After a short pause, he started to read the last paragraph of his main method code and verbalise:

"While, let me see turn robot around , while not facing wall move robot forwards , and turn robot right, so yes , while not facing the wall, first turn robot right, um, and move robot forwards, and turn robot right, is better to use the big loop to make it always running"

As a result, Andre decided to add a while statement as he did in the longest corridor task to encapsulate the code he wished to repeat (Figure 6.5(right), step1 (A& B). Andre started to read his code again and verbalise:

"Okay check, not facing the wall yet, move robot forwards, plus plus yes, and turn robot around go back, and turn robot right yes, and move robot forwards, and turn robot right, um but this always, let me see, ah, so if we check moving back to the move robot forwards, he can move forwards and then turn robot right, not facing the wall, because there is ah, because there is"

Andre realised that there was a mistake in his code; as a result, he updated his code by adding an IF-statement as shown in the Figure 6.5(right), step2 (A& B). Until this stage; Andre had focused only on the plans that allowed the robot to move across the corridors and count the length of each of them. After a short pause, Andre verbalised:

"We need to compare, while robot is not facing the wall, after this is done the present row is already recalled at the beginning of it, um, at the beginning oh, I did a mistake"

Andre realised his mistake related to the method definition therefore he changed `void` to `int` Figure 6.5(right), step3 and he then verbalised:

"We need to make short row to store the value of present row, and then we make the present row to, maybe we can put it there, so we do not need"

After the above utterance, Andre updated his code as shown in Figure 6.6(left), steps4→9. At this stage, Andre started to repeat his code from the longest corridor solution in order to compare the current corridor length with the most wanted holder variable's value. He adapted the comparison to search for the shortest corridor but he made the same pairwise comparison mistake as he had in the longest corridor task Figure 6.4 (left).



Figure 6.5 Andre's first and second screen images for the shortest corridor

Andre had acquired the habit of tracing his code to check it before he attempted to compile it. So, he started to mentally trace through the IF-block:

"If this is five [the gathered variable] and that one six [most wanted holder variable], so it will be changed to this one, but if this is seven [the gatherer variable] so it is not bigger this one."

Andre finally added the RETURN statement for his method (Figure 6.6(left), step10) and then compiled the code and ran the unit tests:

"Oh, no, I need to move twice, because robot will not face the wall, move forwards twice."

Visualising the robot helped Andre to correctly update the block segment related to the robot moving across the world Figure 6.6(left), step11. Andre was surprised when the supplied unit tests failed for the second time. Andre read one of the unit test messages — *"Expected 7 but was 8"*

The unit messages did not support Andre into fixing his code. Andre spent about 6 minutes and 54 second trying to fix his code as well as mentally tracing his code. See Figure 6.6(right), steps11→12 for the last update at this stage. Andre's code failed to

return the length of the shortest corridor for the third time. Andre again re-read one of the unit test messages — “Expected 7 but was 1”



Figure 6.6 Andre’s third and fourth screen images for the shortest corridor

Andre spent about 9 minutes and 15 seconds trying to fix his code. Figure 6.6(right) steps13 through 16 shows the changes made to the code during this time. At this point Andre still had not realised that that his code compared the length of two adjacent corridors in a pair wise manner. Andre again started to mentally check his code:

“Let me check it again, the length of the first corridor one, two, three, four, five, six, seven, eight, nine, ten. Length of the second corridor one, two, three, four, five, six, seven. Length of the third corridor one, two, three, four, five, six, seven, eight. So ah last corridor equal one, present corridor equal ah ten. Ten [the value of the presentRow] less than one [the value of the lastRow], no, um so shortest row equal one [the value of the lastRow]. Now last row equal ten, present row equal 7, so if we compare seven with ten the values of the shorts row will be also seven, again last row will be seven, present row will be eight that mean we compare both of them

oh, no we do not compare with the smallest, let me check it again...., yes, oh no we do not compare it with the smallest.”

```
int findShortCorr()
{
    int presentRow = 1;
    int shortestRow = 1;
    while(!isRobotFacingWall()){
        presentRow = 1;
        //measure the length of present row
        while(!isRobotFacingWall())
        {
            moveRobotForwards();
            ++presentRow;
        }
        if(shortestRow == 1)
            shortestRow = presentRow;

        //compare present row with shortest row
        if(presentRow < shortestRow)
            shortestRow = presentRow;

        //go back and move to the next row
        turnRobotAround();
        while(!isRobotFacingWall())
            moveRobotForwards();
        turnRobotRight();
        if(!isRobotFacingWall())
        {
            moveRobotForwards();
            moveRobotForwards();
            turnRobotRight();
        }
    }

    return shortestRow;
}
```

Figure 6.7 Andre’s final screen image for the shortest corridor

Andre made the final changes to his code as well as adding comments (As shown in Figure 6.7). Andre started to read his code again. Finally when Andre re-ran the supplied unit tests, all the tests passed.

2. Retrospection:

The interview after solving this question took a considerable amount of time and the salient points are noted here. Andre discussed that fact that he had seen the link between the algorithm to finding the longest corridor and this task of finding the shortest corridor. He mentioned that the first algorithm, for the longest corridor, was easier because the initial value of the most wanted holder variable could be set to zero.

At the end of the session, the interviewer reiterated early feedback from previous sessions about using and comparing variables. It was also highlighted that he needed to focus on all possible robot scenarios in order to come up with a working solution. There was also some discussion about how to read and understand the unit test messages and the importance of reading and understanding those messages before starting to try and fix any bugs.

6.2.5. Checking if integers in a 1D Array are sorted in Descending Order (Seq3 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Happy	
Strategies	Sequential	
Activities	<i>Planning</i>	Pen and paper (doodles)
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	8 minutes and 30 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week ten of the P1 course	
Important observations with respect to prior sessions	Prior to encountering this problem, Andre had undertaken a similar exercise checking to see if numbers in an array were ascending as part of his P1 course work.	

Data

Think aloud:

Andre began by reading the problem and verbalised:

“First I need to return true or false [pause], if it sorted ascending, ah, if first place smaller than second one, second one is smaller than third one, ah, this time descending not ascending, the numbers are arranged from the largest to the smallest. Could I use a pen?”

Andre started to draw a one-dimensional array; this array contained four elements. He started to work out how he would need to compare the elements in the array to find out if they were in descending order (see Figure 6.8).

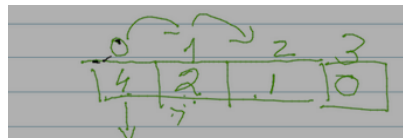


Figure 6.8 Andre’s doodle for checking integers in a 1D Array are sorted in descending order

Once he had worked out the logic he began writing the code. First he wrote the method header. He assumed that the method returned a Boolean value, and then he verbalised while writing his code (see Figure 6.9):

“For int [integer] i equal 0, i less than my array dot length [myArray.length] minus one, i plus plus, [pause], if my array i [myArray[i]] [pause] less than my array i plus one [myArray[i] + 1], I need to return false. Let me check. [Andre pointed to the Figure 6.8], four less than two, no, two less than one, no, one less than zero, no. that is right. If all okay I need

to return true. Let me check if first element is smaller than the second one return false, any time if my array i [myArray[i]] less than my array i plus one [myArray[i] + 1], return false”

Finally, Andre compiled and ran the supplied unit tests to verify the correctness of his solution.

```

boolean isSorted(int[] myArray)
{
    for(int i=0; i<myArray.length-1; ++i){
        if (myArray[i]< myArray[i+1])
            return false;
        }
    return true;
}

```

Figure 6.9 Andre’s screen image for checking integers in a 1D Array are sorted in descending order

Andre did not encounter any issues in trying to solve this task. From the beginning it was clear that he had recalled the class exercise which check to see if integers in a one-dimensional array were sorted ascending. Not retrospective interview was conducted. He was happy that he had managed to solve the problem successfully on his own.

6.2.6. Smallest Element in a 1DArray (Seq3 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Confused	
Strategies	Stepwise design	
Activities	<i>Planning</i>	Verbalise
	<i>Tracing</i>	Mental and pen and paper tracing
	<i>Unit test</i>	Read messages and test code
	<i>Time on task</i>	46 minutes and 13 seconds
	<i># of compilation</i>	4
	<i># of execution</i>	3
Intervention	“General prompt” scaffolding – provided on request	
Timing	Week ten of the P1 course	
Important observations with respect to prior sessions	From observations made for solving this question, it is evident that Andre’s lack of prior knowledge led to a pattern of continuous errors. For solving this question, Andre made the same mistake (an incorrect pairwise comparison) when he tried to transfer his knowledge from longest corridor and shortest corridor to this question. Also, the quality of the code he had used for the longest corridor, the shortest corridor, and the smallest stack of beepers (see Appendix A) questions may have hindered Andre’s ability to transfer his knowledge without difficulty for solving this task.	

Data

1. Think aloud:

Andre began by reading the problem and immediately started to plan his solution, and he verbalised:

“The method should return the integer smallest, so we need to find the smallest [pause], so if we set up, we need to set up, we need to set up the, ah, the integer call it the smallest. Then we need loop to read all the elements of the array and we need to compare each element which one, the smallest.”

He then proceed to write the method header which returned an integer value (Figure 6.10 (left)). He continued coding and wrote the definition and initialisation of the most wanted holder variable `smallest` followed by the FOR-loop that consisted of a stepper variable called `count`. Inside the FOR-loop block, Andre added an IF-block followed by the word RETURN as shown in Figure 6.10 (left).

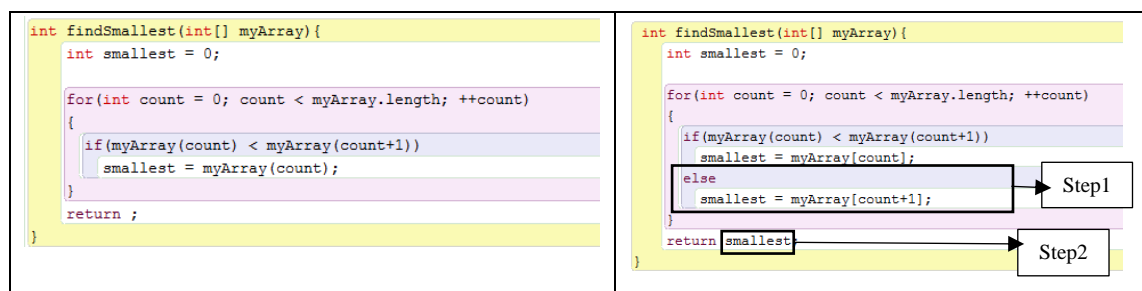


Figure 6.10 Andre’s first and second screen images for the smallest element in a 1D array

Andre started to read and check his code:

“Return the smallest, let me see, if this is smaller than that yes, if is bigger that no and let me see, I know the smallest ah [pause] else okay”

As a result, he decided to add an ELSE-block (see Figure 6.10(right), step1). As shown in Figure 6.10, Andre concentrated on two successive elements in the array each iteration, which meant that he repeated his old mistake of pairwise comparison (as already discussed for the tasks which required finding the longest and shortest corridor).

“First compare this position zero with one, so if number in zero is smaller than index one so we should put this one in this, temporary and if it’s bigger than that we should put the next one to the smallest, and then compare the next one when one to zero but altering from zero to one, we compare one and two so if one is the smallest then two so the one still be the smallest but if it bigger than two so I will change.”

Then he continued to type the RETURN statement (see Figure 6.10(right), step2). Andre compiled his code and he easily fixed the one syntax error. He had used round braces () instead of square braces [] when accessing an element of the array (Figure 6.10 before fix the syntax error and Figure 6.11 after fix the syntax error). Andre ran the supplied unit tests, all the tests failed. He started to read the test messages:

“Oh no minus nine, the smallest element is minus nine”

Andre opened the supplied unit test file and started to read the relevant test case which used an array containing the integer values {-9, 1, 2, 3, 4}. Andre started to verbalise his code:

“It is smaller than this, it should be minus nine, ah this smaller than this [Andre means one], so it should be minus nine, if this one is smaller than that one [Andre means two], and let me see, so, so, let me see [pause] if this one is smaller than that one, should be write one, one no”

Andre deleted the IF-ELSE block and introduced a second FOR-loop block as shown in Figure 6.11.

```
int findSmallest(int[] myArray)
{
    int smallest = 0;

    for(int count = 0; count < myArray.length; ++count)
    {
        for(int i = 1; i < myArray.length+1; ++i)
        {
            if(myArray[count] < myArray[i])
                smallest = myArray[count];
            else
                smallest = myArray[i];
        }
    }

    return smallest;
}
```

Figure 6.11 Andre’s third screen image for the smallest element in a 1D array

Andre compiled and ran his program and all tests failed for the second time, he started to read his code and verbalised:

“So if it is array smallest than this one and count greater than this, we chose different loop and we compare to the next one, which is the smallest”

Finally, Andre said *“I’m not sure”* and he asked for help.

2. Scaffolding:

The following is a conversation between Andre and the interviewer.

Interviewer: *“Have you seen this question before?”*

Andre: “I think so, it is similar to finding the least beepers in the stack. When I started with one-dimensional array, I actually start to think that the beepers in each box is the element of the array, but the dimensional array is still new”

The interviewer asked Andre to write an algorithm that could find the smallest stack of beepers using smart-pen and paper (“general prompt” scaffolding). Andre wrote the algorithm in pseudo code as shown in Figure 6.12.

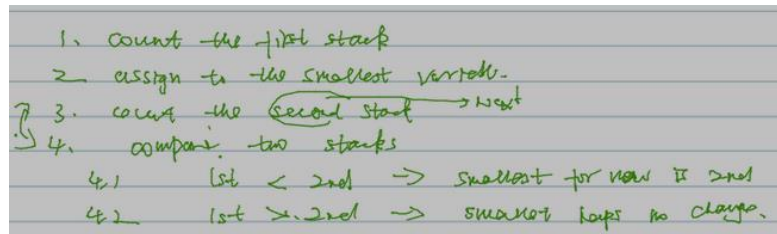


Figure 6.12 Andre’s doodle for the small stack of beepers algorithm

Then the interviewer asked Andre about the order of the third and the fourth elements. Andre responded by updating his doodle swapping the third and fourth step (indicated with a double headed arrow in Figure 6.12). After that, the interviewer asked Andre about the value of the second variable: does it change or remain fixed. Andre responded that what he meant by the ‘second value’ was the next value. The interviewer asked Andre to desk check his algorithm, giving him using the array {1, 0, -1, 2}. The trace generated is shown in Figure 6.13.

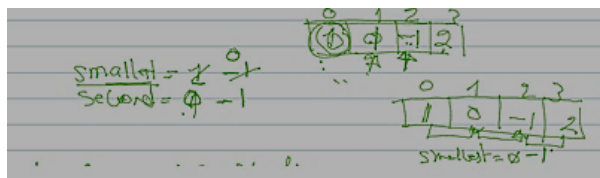


Figure 6.13 Andre doodle to trace the smallest stack of beepers algorithm

After tracing Andre verbalised:

“I compare the first two, I did not compare with this one because in the smallest I was using the WHILE-loop to check the end of the stacks but this time I need to use FOR-loop, I did not practise a lot using FOR-loops with the array.”

3. Think aloud:

Andre started to update his code as shown in Figure 6.14 and re-run the supplied unit tests.


```

int findSmallest(int[] myArray)
{
    int smallest = myArray[0];

    for(int count = 0; count < myArray.length; ++count)
    {
        if(myArray[count] < smallest)
            smallest = myArray[count];
    }

    return smallest;
}

```

Figure 6.14 Andre's fourth screen image for the smallest element in a 1D array

6.2.7. Index of the Largest Element in a 1D Array (Seq3 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Sequential	
Activities	<i>Planning</i>	Verbalise
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	6 minutes and 42 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	1
Intervention	None	
Timing	Week ten in the same session and immediately after solving the smallest element in a one-dimensional array problem.	
Important observations with respect to prior sessions	Andre had managed to solved the smallest element in a one-directional array with the with the interviewer's assistance.	

Data

Think aloud:

After reading the question, Andre started planning:

“This question is similar to the first question [find the smallest element in a one-dimensional array], the first question find the smallest one, and this question find the largest one, um, should return the index of largest element, the point is how to find the index, is the basically the same so, the difference the first one is asked to return the smallest element and now asked as to return the index of it, basically is the same returning, the most difficult part of this question how to find the index of an array”

He then wrote the method signature and defined the most wanted holder variable largest and assigned the value of the first element of the array to this variable. Andre started to verbalise his planning further focusing on how to define and work out the value of the index:

“Let me think about the largest, so first think to compare with, find out the largest, and I need to know the index of it, index of the largest element, if is bigger than that if it smallest than than the largest one, will be the okay, if it is bigger than that one, um [pause] I need index of result”

Andre decided to define another most recent holder variable and he called it `index` and initialised it to zero. Then he continued on and line by line wrote the code shown in Figure 6.15.

While writing the programming code, Andre paused twice to question the value of the stepper variable, and what it should be and how to compare it focusing specifically on the IF-block.

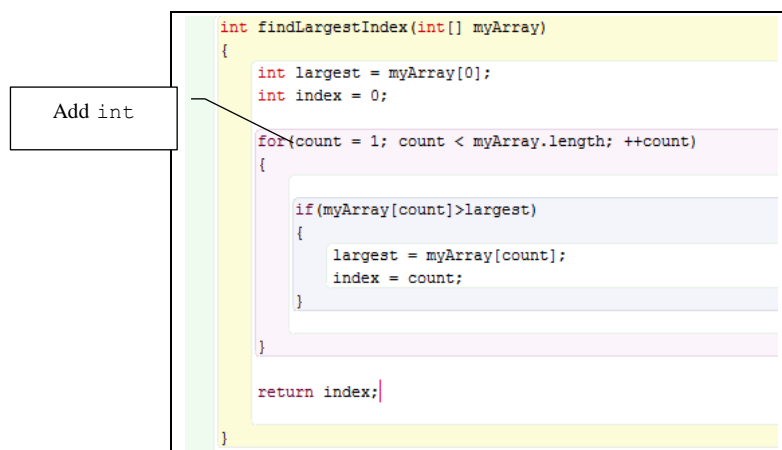
The first occasion was after setting the most holder variable `largest` to be current value in the array (the element at `count`):

“Largest equal array but if it smaller than that, I do not need to change it, oh yes, if it big than that”

Secondly, after setting the most holder variable `index` to the value of the stepper variable:

“If we comparing , the second one is bigger than the value of the index, the index will change will change to second ,if it still in that the index is always will be one, but if third is bigger than the value of index then will be changed again , otherwise the index will be one”

Andre compiled his code and immediately identified and fixed the syntax error present (Figure 6.15) by adding the missing data type `int` to the stepper variable declaration. Finally he ran the unit tests to verify the correctness of his solution, they all passed.



```
int findLargestIndex(int[] myArray)
{
    int largest = myArray[0];
    int index = 0;
    for(count = 1; count < myArray.length; ++count)
    {
        if(myArray[count]>largest)
        {
            largest = myArray[count];
            index = count;
        }
    }
    return index;
}
```

A callout box labeled "Add int" points to the `count` variable in the `for` loop's increment part, `++count`.

Figure 6.15 Andre’s code for find the largest index

6.2.8. Largest Element in a 2D Array (Seq4 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	Read messages and test code
	<i>Time on task</i>	9 minutes and 16 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	None	
Timing	Week five of the P2 course	
Important observations with respect to prior sessions	The previous problem that related to this question was finding the smallest element which was solved three months earlier. In solving this question, it appears that Andre’s faulty adjacent pairwise comparison schema still exists despite having fixed the error on three earlier occasions – a new correct schema does not appear to have been formed.	

Data

Think aloud:

Andre began programming the method name and the array of type integer as a passing parameter, and then he defined a nested FOR-loop block, followed by an IF-statement as shown Figure 6.16 (left). At this point he read that line of code and verbalised:

“So let me check it, so row zero zero, then row will be one no, so let me see.”

He made a decision to update the IF-statement as shown in Figure 6.16(right), step1. And then he read that line of code and verbalised again:

“Fix the row and let the column so that will be the row zero and row one so zero zero, zero one, zero two, and after this finished, before for loop.”

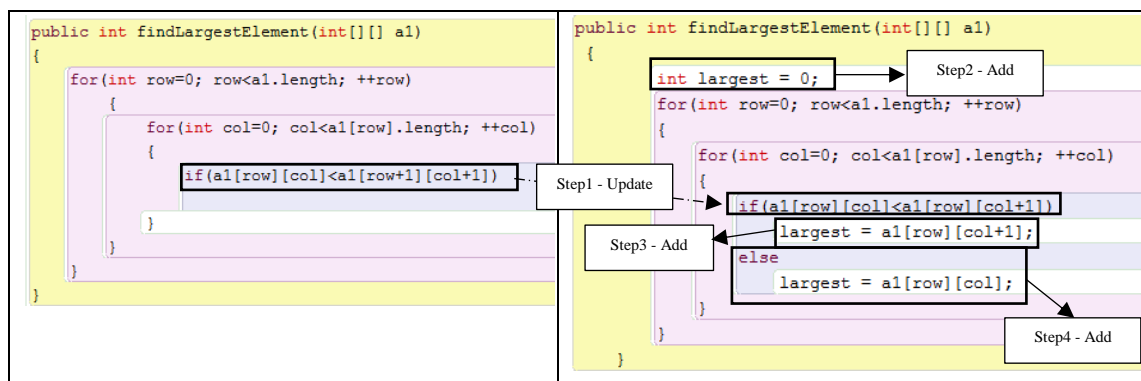


Figure 6.16 Andre’s first and second screen images for the largest element in a 2D array

After that, Andre decided to define the most wanted holder variable `largest` and set it to zero (Figure 6.16 (right), step2). Then he added an assignment command after the IF-statement as shown in Figure 6.16 (right), step3.

Andre started to trace the IF-block:

“If the second one is bigger than the first one store the second one in the largest, but if it not stores this one”

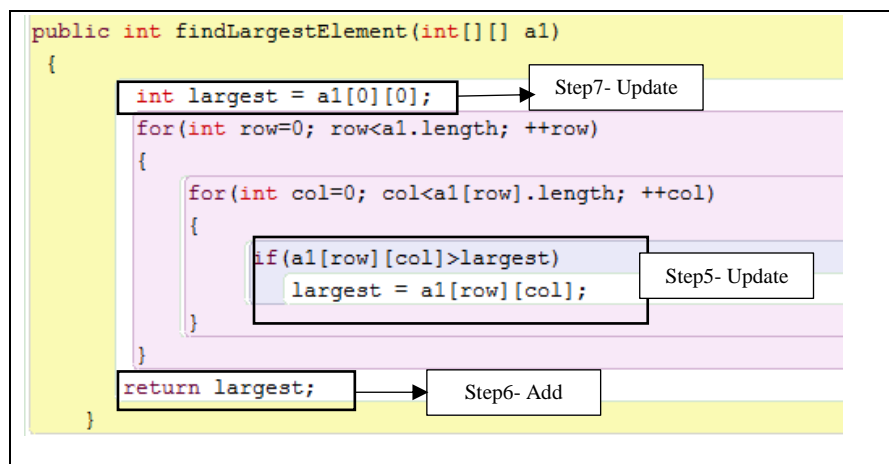
He decided to add an ELSE-block as shown in Figure 6.16 in step4 once introducing the adjacent pairwise comparison error made consistently during his first semester of think aloud sessions. Andre mentally checked his code again:

“Let me check it so first think we set row to row, and we set column to the row, and the first location which is zero zero, and then we check if zero one is bigger than zero zero largest will be zero one, but if it is not, we will set the largest one to um zero zero and so if then oh yes, if the column let me see we can change it.”

Andre started to update his code and added the RETURN Java commands (Figure 6.17, steps5→6).

Andre ran the supplied unit tests and one of three tests failed that consist of an array of negative numbers only. He started to read the code for the test failed, and he verbalised:

“Expected -1 but was 0, if it minus one let me see, so the largest one should be the largest, so I just need to modify the code because there is a minus number there -19, -1,-2,-9. Let me set the largest number to a1 zero zero so it will be the first, if the first one is largest than two if if its not.”



```
public int findLargestElement(int[][] a1)
{
    int largest = a1[0][0];
    for(int row=0; row<a1.length; ++row)
    {
        for(int col=0; col<a1[row].length; ++col)
        {
            if(a1[row][col]>largest)
            {
                largest = a1[row][col];
            }
        }
    }
    return largest;
}
```

The screenshot shows the code with several annotations: a box around `int largest = a1[0][0];` with an arrow pointing to a box labeled "Step7- Update"; a box around the `if(a1[row][col]>largest)` and `largest = a1[row][col];` lines with an arrow pointing to a box labeled "Step5- Update"; and a box around `return largest;` with an arrow pointing to a box labeled "Step6- Add".

Figure 6.17 Andre’s last screen image for the largest element in a 2D array

Not unsurprisingly Andre easily fixed the error as it was an error he had fixed several times in the past (see Figure 6.17, step7).

6.2.9. Print the Highest Mark and Name of Every Student in a Collection of Student Objects (Seq5 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	Read test output and test code
	<i>Time on task</i>	19 minutes and 7 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	None	
Timing	Week seven of the P2 course	
Important observations with respect to prior sessions	Andre showed evidence that he had learned from his earlier mistakes (see longest corridor, shortest corridor, smallest element in a one-dimensional array, and largest element in a 2D array). In the other words, the scaffolding and feedback given to him during the meeting sessions were effective in enabling him to move forward and solve subsequent programming tasks in each sequence.	

Data

Think aloud:

Andre read the question twice, and then he started to write his solutions without hesitation or verbalisation. Andre started with the method signature (Figure 6.18, step1 (A &B)), followed by the FOR-loop statement with the stepper variable *i*. The function of the stepper variable was to iterate for all the elements of the ArrayList called “Student” (Figure 6.18, step2 (A & B)). Then Andre verbalised:

“Student marks this actually a 1D array, and with this array I can I can find the highest”

He then defined a most recent holder variable `highestMark`, and set its initial value to zero (Figure 6.18, step3). Andre continued to enter the rest of his code (as shown in Figure 6.18, step4) in order line by line.

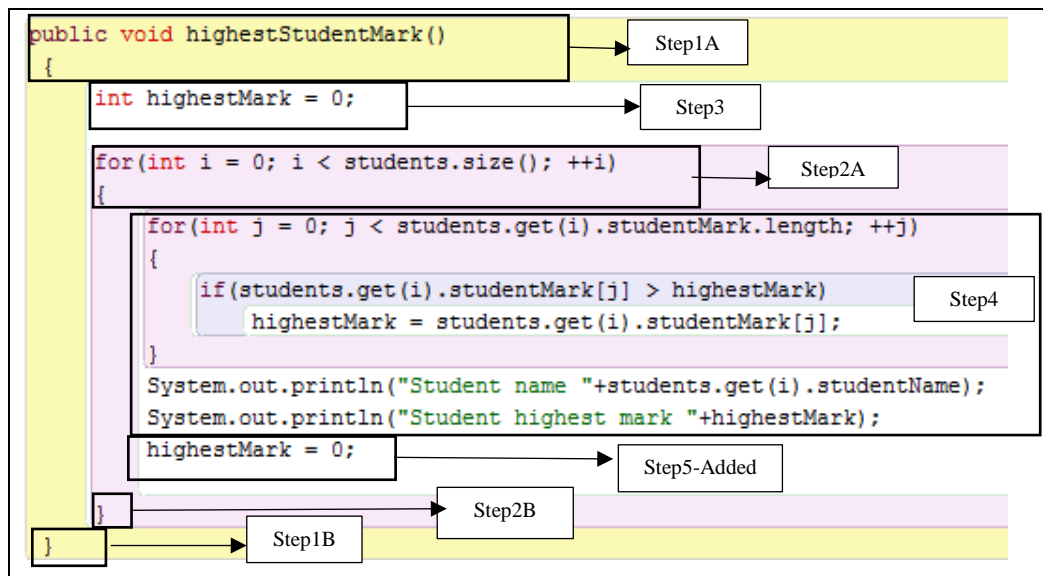


Figure 6.18 Andre’s screen image for highest student mark in a collection of student objects

Finally, Andre ran the supplied unit tests and started to read the output. He was surprised when the output did not match the results supplied to him in the question. He started to read the test code and to compare it with the outputs. While Andre read the test file, he verbalised — “Tom is ah yes, I got it I need to set the highest”

Andre easily fixed his code by adding a line which reset the most recent holder variable to zero before finding the highest mark for the next student in the ArrayList (see Figure 6.18, step5).

6.3. Luke’s Think Aloud Sessions

6.3.1. Summary

At the earliest stages of learning to program, Luke found some difficulties in making a connection between unfamiliar situations in terms of familiar ones (he turned to a stopper twice during the P1 course). But later on, he showed evidence that his level of programming had improved through solving different programming tasks and in following the feedback given to him by the interviewer. During this longitudinal study, Luke changed the meeting sessions approximately two times. Luke showed evidence that his level in programming was high due to his position in the first quartile of P1 and P2 course.

6.3.2. Counting the Number of Beepers in a Single Corridor (Seq2 – Q1)

Encoding

Question	Not Solved	
Behaviours	Stopper	
Emotion	Indiscernible	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging
	<i>Unit test</i>	Read the unit test message – at the final stages of problem solving
	<i>Time on task</i>	9 minutes and 34 seconds
	<i># of compilation</i>	6
	<i># of execution</i>	6
Intervention	Exact solution –interviewer intervention	
Timing	Week four of the P1 course	
Important observations with respect to prior sessions	Luke solved counting the length of corridor (Appendix A) without any difficulty. He struggled to apply what he had practised before (count the length of corridor, far transfer problem) in a new context. This is not unusual for novice programmers. The same result has been reported in similar observations conducted by Ambrose et al.(2010). They stated that learners may fail to transfer relevant knowledge and skills when they do not hold a rich conceptual understanding of underlying principles and structure.	

Data

1. Think aloud:

Luke appears in this case to fall back on a familiar schema which was not the most appropriate schema. He seems to have recalled the algorithm or code to move the robot across only a single corridor and count its length while there are beepers on the ground and print the result to the console. He first wrote the code shown in Figure 6.19 (left). He ran the supplied unit tests and all tests failed. Luke focused on visualising the first Robot World scenario a corridor of length five and verbalised:

“Oh, my program not stop even if the robot hit the wall [pause], I need a way also to check if the robot facing the wall, so to do that I would [pause], I’m not sure how to do that [pause], I will just put IF statement inside the WHILE loop, if space in front of robot clear at that while move the robot forwards”

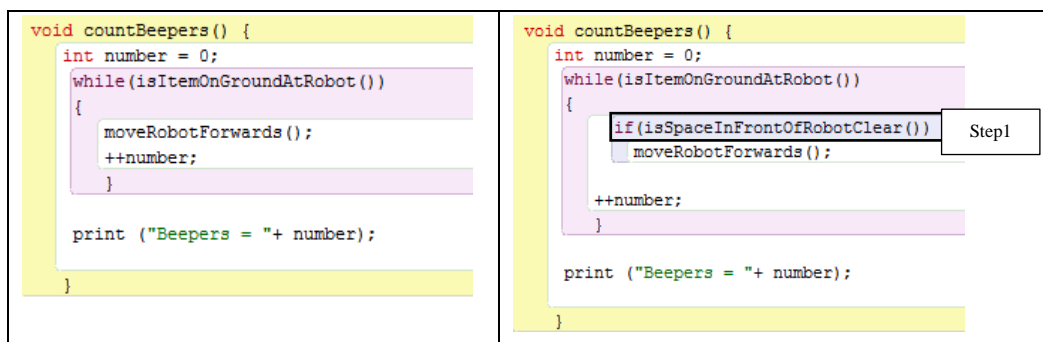


Figure 6.19 Luke’s first and second screen images for counting all beepers

Luke added an IF-block inside the WHILE-block as shown in Figure 6.19 (right) and again ran the first of the unit tests. While, he visualised the robot moving, he verbalised:

“Not working, so it get to the end and then not die this time [pause], just to continue in loop because the WHILE statement, I need a way to, another variable Boolean type and set it to false”

At this point Luke started to adopt a trial and error strategy to programming. After adding each Java command (steps shown in Figure 6.19 (right) and Figure 6.20(left) and (right)).

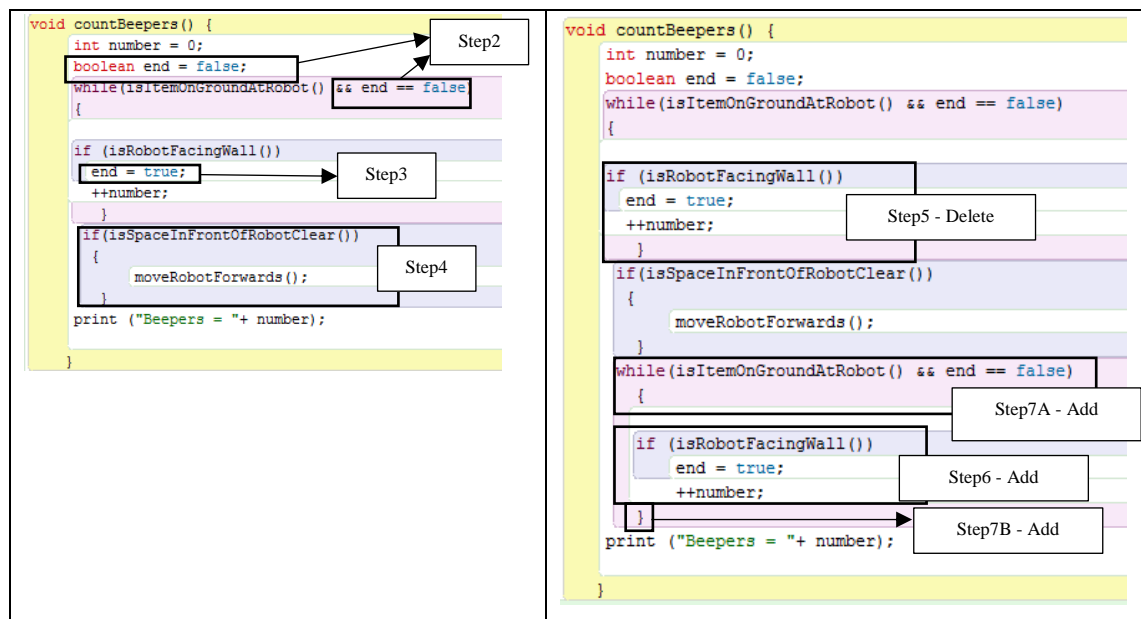


Figure 6.20 Luke’s third and fourth screen images for counting all beepers

Luke ran the supplied unit tests focusing only on the first Robot World scenario. He made no attempt to read or trace his code. After his fifth attempt at running the tests failed for the first Robot World scenario Luke started to read the unit test message for the first time and verbalised:

“Expected seven beepers not four beepers, [long pause] oh there may be more than one beeper so I gonna”

Luke started to update his code as shown in Figure 6.20(right) without any evidence that he tried to mentally trace or read his code. Luke’s code was still well away from a correct solution, therefore the interviewer offered to help him.

2. Scaffolding:

The interviewer asked Luke to write an algorithm that would allow the robot to pick up all beepers at each stack across a single corridor (“General prompt” scaffolding) using the smart-pen and paper. Luke’s attempt is shown in Figure 6.21 – he failed to recall the correct algorithm.

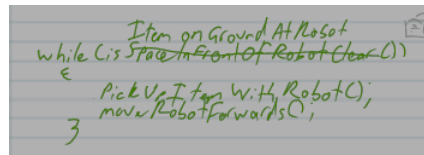


Figure 6.21: Luke's doodle for counting all beepers

The interviewer redirected Luke to trace his online code the final result of the actions provided in Figure 6.20(right):

Luke: *"That caused a problem, when the robot moved. The robot picked up beepers and moved."*

Interviewer: *"Yep, does your program pick up all the beepers in the corridor?"*

Luke: *"No"*

Interviewer: *"Is there anything else you might need?"*

Luke: *"Add WHILE-statement as well"*

Interviewer: *"Yes, please could you show me how?"*

Luke: *"I do not know."*

Luke gave up and asked for help. The interviewer started to use a stepwise refinement technique to explain the code to Luke starting with the algorithm for picking up all the beepers from a single stack, this algorithm was then extended to counting the beepers from each stack, followed by the programming code for pick up all the beepers and finally counting the beepers in a single corridor.

3. Retrospection:

The following is part of the conversation between the interviewer and Luke:

Interviewer: *"Have you seen this question before?"*

Luke: *"No"*

Interviewer: *"How many beepers has each stack got?"*

Luke: *"At the beginning I thought it is one, then, the test shows the expected value was seven, so it is more than one"*

Interviewer: *"Did the test help you to check the number of beepers?"*

Luke: *"Yes"*

Interviewer: *"Do you have an idea about how to implement this code?"*

Luke: *"No"*

Interviewer: *"You do not have any idea about how to write this program, but do you have an idea that you need a counter to count the number of beepers?"*

Luke: *"Yes, I just practised with you"*

Interviewer: *"For this question, you do not have any plan?"*

Luke: *"Yes"*

Interviewer: *"So if I give you one hour, will you try with it until the time is finished?"*

Luke: “Yes”

Interviewer: “For one hour, do you think you can solve it?”

Luke: “May be not sure”

Interviewer: “You mean by trial and error?”

Luke: “Yes”

Interviewer: “If this happened in a test, would you continue with this question?”

Luke: “I will solve the other question then I will return to this”

The interviewer reviewed the video tape with Luke. The interviewer focused on how Luke could avoid problems resulting from the lack of focus on all possible robot scenarios as well as reading and understanding all the unit tests messages before updating the code.

6.3.3. Longest Corridor (Seq1 – Q3)

Encoding

Question	Not solved	
Behaviours	Stopper	
Emotion	Confused then Surprised	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging
	<i>Unit test</i>	Read the unit test message
	<i>Time on task</i>	45 minutes and 5 seconds
	<i># of compilation</i>	8
	<i># of execution</i>	7
Intervention	Exact solution - provided on request	
Timing	At the end of the sixth week of the P1 course in the intra-semester break.	
Important observations with respect to prior sessions	Like Andre, Luke solved the comparing the length of two corridors task (see Appendix A) without any difficulty. He had practiced in isolation counting the length of a corridor and comparing the value of two integers, and he just needed to concatenate these two programming plan. In the case of this question the same two programming plans were used but they had to be combined by merging and nesting rather than concatenating and he had significant difficulty doing this. There were additional robot navigation tasks as the robot had to be moved and orientated in order to get to the second corridor before the second corridor’s length could be counted. Unlike Andre, Luke also had trouble writing the method header.	

Data

1. Think aloud:

Luke began by reading the problem. Luke then attempted to write the method signature – he took two minutes and 39 seconds to figure out how to write the method header. He assumed that he needed to define three input parameters, one for each corridor shown in the example provided with the questions description:

“I’m still working with my homework assignment on methods, let me remember [long pause]. I need to define three variables, one for each corridor. Int [integer] I gonna use int length of a

[integer variable name], *int length of b* [integer variable name], *int length of c* [integer variable name] as [pause]”

At this point Luke realised that there were an unknown number of interconnected corridors, therefore he decided to update the method signature definition and verbalised:

“*It is gonna be different because there is a different number of corridors each time, so I’m going to stick with one* [one variable]”

Luke made his decision to define the method `findLongestCorr()` with no parameter and no return a value (he did not discover until he compiled his code that there was a problem with his method header). He did not appear to retrieve a fully formed schema for counting the length of corridor but instead appears to have retrieved sub-plans and joined those plans. Firstly he recognized the need to iterate in order to move the robot forward then he realised a gatherer variable was required. He hesitated as to what the initial value of the gatherer variable should be, zero or one, and finally after a short pause he made a decision and set the gather variable to one.

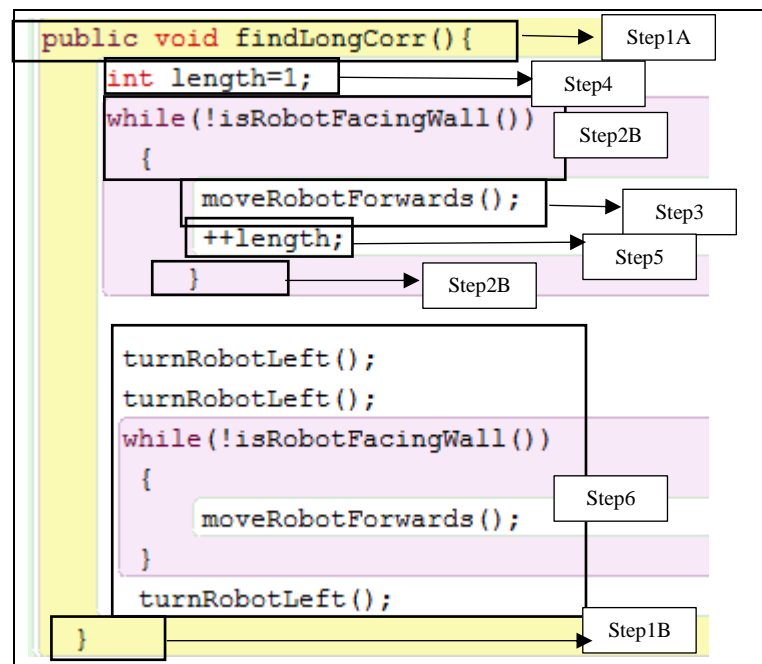


Figure 6.22 Luke’s first screen image for the longest corridor

Luke continued writing code which allowed the robot turn and return back along the corridor. He then added code in an attempt to turn the robot so that it could move on to the next corridor but did not succeed in correctly orienting the robot so that it faced north. He failed to discover this issue despite numerous attempts at running the unit tests his code (Figure 6.22, Luke’s code up to this stage).

From this point on Luke started to experience significant difficulty. He began to generate a solution using a trial and error approach. The most interesting samples of Luke’s

programming are listed in Figure 6.23 and Figure 6.24. Despite the various changes to his code Luke found that the tests failed. His focus was solely on the ultimate test result (pass or fail). He did not follow the visualisation of his code executing in the RobotWorld and therefore missed seeing that his code resulted in the robot returning to the start of the first corridor each time.

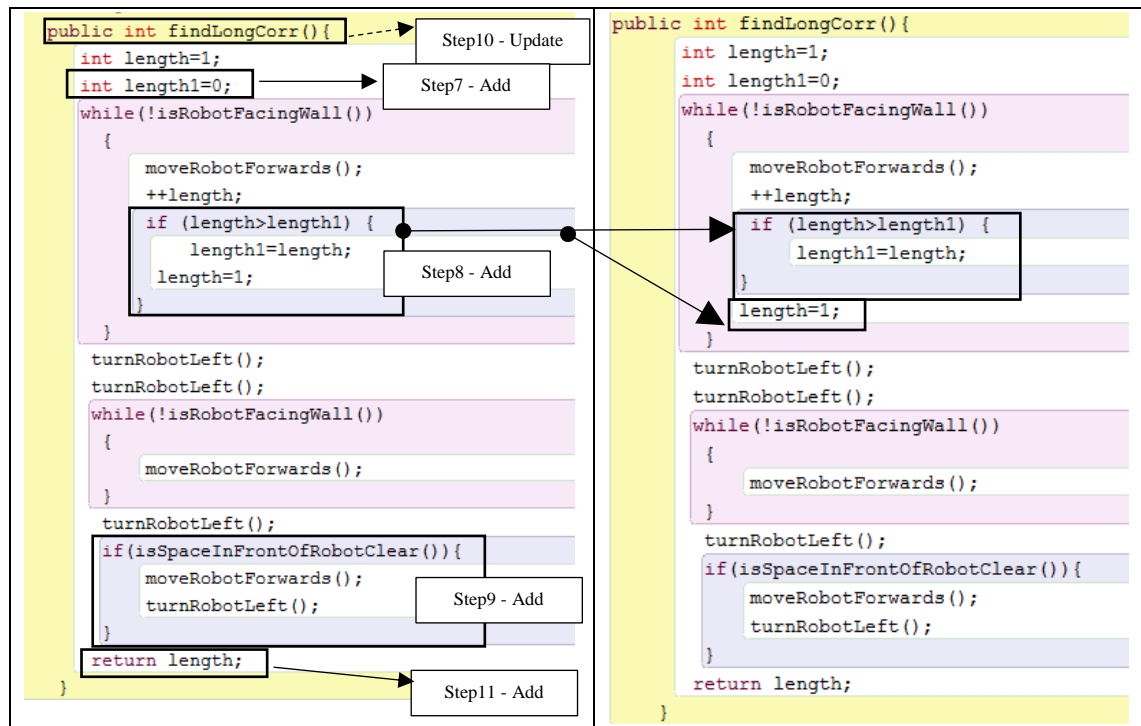


Figure 6.23 Luke's second and third screen images for the longest corridor

After a short pause, Luke updated the RETURN statement to return the length1 variable rather than the length variable (Figure 6.25(left)). He again ran the unit tests and for the first time focused on visualising the robot moving across the corridors. After a short pause, Luke verbalised:

“One of the test failed [the scenario that allowed the robot to move across three corridors] and the other two passed. Expected nine but was six, so here because, um [pause], I'm not sure why the test failed because in this it should change to length of next corridor because in this case it should be um [pause]”

He then moved the first IF-block (Figure 6.25). Luke ran the supplied unit tests and he was surprised that he still got same unit test results. He started to verbalise:

“So okay what is happening, in the last test the robot started at the bottom, while the top for the other two, so the code can work if starting from the top, just from the third test it is started from the bottom [Luke did not realise that in the three scenarios that the robot starts at the same position (0, 0)], so what I'm going to do is [pause]. I'm not sure how to solve this question [long pause] so if I just make it um, I need to make started from the top, how can I do it [long pause].”

```

public int findLongCorr(){
    int length=1;
    int length1=0;
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
        ++length;
    }
    if (length>length1) {
        length1=length;
    }
    length=1;
    // }
    turnRobotLeft();
    turnRobotLeft();
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
    }
    turnRobotLeft();
    if(isSpaceInFrontOfRobotClear()){
        moveRobotForwards();
        turnRobotLeft();
    }
    return length;
}

```

```

public int findLongCorr(){
    int length=1;
    int length1=0;
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
        ++length;
        if (length>length1) {
            length1=length;
            length=1;
        }
    }
    turnRobotLeft();
    turnRobotLeft();
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
    }
    turnRobotLeft();
    if(isSpaceInFrontOfRobotClear()){
        moveRobotForwards();
        turnRobotLeft();
    }
    return length;
}

```

Figure 6.24 Luke's fourth and fifth screen images for the longest corridor

```

public int findLongCorr(){
    int length=1;
    int length1=0;
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
        ++length;
        if (length>length1) {
            length1=length;
        }
    }
    length=1;
    turnRobotLeft();
    turnRobotLeft();
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
    }
    turnRobotLeft();
    if(isSpaceInFrontOfRobotClear()){
        moveRobotForwards();
        turnRobotLeft();
    }
    return length1;
}

```

```

public int findLongCorr(){
    int length=1;
    int length1=0;
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
        ++length;
    }
    if (length>length1) {
        length1=length;
    }
    length=1;
    turnRobotLeft();
    turnRobotLeft();
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
    }
    turnRobotLeft();
    if(isSpaceInFrontOfRobotClear()){
        moveRobotForwards();
        turnRobotLeft();
    }
    return length1;
}

```

Figure 6.25 Luke's sixth and seventh screen images for the longest corridor

After a long pause, Luke continue try and get a working solution as illustrated in Figure 6.26 with no success. Finally he asked for help — “I need your help”.

```

public int findLongCorr(){
    int length=1;
    int length1=0;
    turnRobotLeft();
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
    }
    turnRobotLeft();
    turnRobotLeft();
    turnRobotLeft();
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
        ++length;
    }
    if (length>length1) {
        length1=length;
    }
    length=1;
    turnRobotLeft();
    turnRobotLeft();
    while(!isRobotFacingWall())
    {
        moveRobotForwards();
    }
    turnRobotLeft();
    if(isSpaceInFrontOfRobotClear()){
        moveRobotForwards();
        turnRobotLeft();
    }
    return length1;
}

```

Figure 6.26 Luke’s final screen image for the longest corridor

2. Scaffolding:

When the interviewer attempted to redirect Luke and provide assistance, he gave up on the task and was not receptive to assistance. The interviewer started to use a stepwise refinement technique to explain the code to Luke. The interviewer started with counting the length of corridor program, followed by comparing two integer numbers, then programming plans that allowed the robot to move to the next corridor, and finally repeating the process of moving, counting, and comparing n-1 times.

3. Retrospection:

Luke found it hard to recall the sequences he had used to try solve the question, even though the interviewer reviewed the video tape of the programming session with him. In a way this is not surprising as he fell back on a trial and error approach to programming as soon as he encountered a problem. During the retrospection interview; the interviewer focused on the way that Luke ignored many times visualising the robot moving across his world and focused on unit tests only.

Luke confirmed that he had not solved questions similar to this question even in the homework assignment. And he said that the most difficult part was working out how to repeat the process for moving and comparing the result many times.

6.3.4. Smallest Stack of Beepers (Seq2 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing, Pen and paper, and visual debugging
	<i>Unit test</i>	
	<i>Time on task</i>	12 minutes and 8 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	None	
Timing	After week six in the intra-semester break	
Important observations with respect to prior sessions	<ul style="list-style-type: none"> • When solving Seq2 – Q1, Luke was clearly outside out of his depth and was unable to solve the problem and he was supplied the model answer for this question. Luke was able to solve different programming tasks in the same sequence which suggests that he was able to learn from the model answer and was able to apply that learning to new situations. • In previous sessions, Luke focused on only one of the several example robot scenarios supplied when trying to fix bugs which often led to an incorrect solution. The interviewer suggested to him that he should check all the scenarios during two separate retrospective interviews for Seq2 – Q1 and Seq1 – Q3. When solving this task (Seq2 – Q3) he seems to have taken notice of the advice and he examined all of the scenarios in order to reach an answer. 	

Data

1. Think aloud:

Luke initially wrote the code shown in Figure 6.27 (left) sequentially and without hesitation. Before compiling his code Luke said:

“I’m going to set um ... [pause] ... to um because right now the smallest have no value to compare with the first square, so [pause] I should set this to a hundred [pause], this should compare with the first square, then I need to compile and test.”

Luke set the most wanted holder variable to hundred before compiling and running his code (see Figure 6.27 (right)).

```

int findSmallestStack()
{
  int currentCount = 0;
  int smallest;

  while(!isRobotFacingWall())
  {
    while(isItemOnGroundAtRobot())
    {
      ++currentCount;
      pickUpItemWithRobot();
    }
    if(currentCount < smallest)
    {
      smallest = currentCount;
    }

    currentCount = 0;
    moveRobotForwards();
  }

  return smallest;
}

```

```

int findSmallestStack()
{
  int currentCount = 0;
  int smallest=100;

  while(!isRobotFacingWall())
  {
    while(isItemOnGroundAtRobot())
    {
      ++currentCount;
      pickUpItemWithRobot();
    }
    if(currentCount < smallest)
    {
      smallest = currentCount;
    }

    currentCount = 0;
    moveRobotForwards();
  }

  return smallest;
}

```

Figure 6.27 Luke's first and second screen images for the smallest stack of beepers

On testing his code, Luke discovered that one test failed (in which the smallest stack of beepers was located in the last location in the corridor). He lined up the Robot World windows so he could examine all the test results at the same time. He then started to read his code while checking against the test results.

“Looking at the tests, not testing the last square in each corridor, so I need to look at why. While loop [while not facing wall], moving forwards and not testing the last one, so outside this while loop, another while, I need to copy that”

Because of the limited programming constructs the students have at this stage of the course and the Robot World functionality constraints it not possible to iterate one more time in the current loop to count the final stack of beepers — an extra statement is required after the while loop. Luke copied the code which counted beepers and compared the number of beepers with the most wanted holder variable and pasted at the end of his existing method body (Figure 6.28). It should be noted that the first nested WHILE-loop only runs until the robot is in front of the wall and stops before the final stack beepers is counted. Finally, Luke ran his code and all tests passed.


```

int findSmallestStack()
{
    int currentCount = 0;
    int smallest=100;

    while(!isRobotFacingWall())
    {
        while(isItemOnGroundAtRobot())
        {
            ++currentCount;
            pickUpItemWithRobot();
        }
        if(currentCount < smallest)
        {
            smallest = currentCount;
        }

        currentCount = 0;
        moveRobotForwards();
    }

    while(isItemOnGroundAtRobot())
    {
        ++currentCount;
        pickUpItemWithRobot();
    }
    if(currentCount < smallest)
    {
        smallest = currentCount;
    }

    return smallest;
}

```

Figure 6.28: Luke's third screen image for the smallest stack of beepers

2. Retrospection:

The following is part of the conversation between the interviewer and Luke:

Interviewer: "From the beginning you defined the variable *smallest* but you did not assign a value to that variable, is that right?"

Luke: "When I wrote this line, I did not realise that I needed to set up a value, but later on I skim read my code and realised that I needed to set the variable."

Interviewer: "Have you seen this question before?"

Luke: "No, not this question."

Interviewer: "Have you seen something similar to this?"

Luke: "Yes, in our meeting."

Interviewer: "Do you think that your program will work if the number of beepers is more than 100?"

Luke: "[Pause] ah [pause]"

Interviewer: "Why did you select the number 100?"

Luke: "Because I saw that will be the higher."

Interviewer: "May be you saw that the question was about maximum and minimum students' marks? And students' marks are between zero and hundred?"

Luke: "Oh, yes"

Interviewer: “*May be in the homework, they asked you to find the minimum students’ mark?*”

Luke: “*I think that*”

Interviewer: “*So you remembered that plan, is the right?*”

Luke: “*Yep, and counting beepers*”

Interviewer: “*That means you also started to think how to transfer your knowledge from counting the minimum mark to minimum beepers.*”

Luke: “*Yep*”

The interviewer asked Luke to trace his code using the values 101,102,105,110,104. Once Luke traced his code with these specific values, he realised that his code was not generalisable solution although it worked for the scenarios provided for this task. At the end of the session with Luke, the interviewer discussed the quality of his code and how he could further develop.

6.3.5. Shortest Corridor (Seq1 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Surprised	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing, Visual debugging, Doodles – desk check
	<i>Unit test</i>	Read the unit test message
	<i>Time on task</i>	7 minutes and 41 seconds
	<i># of compilation</i>	6
	<i># of execution</i>	4
Intervention	“General prompt” scaffolding – provided on request	
Timing	After week six in the intra-semester break.	
Important observations with respect to prior sessions	Earlier Luke had been unable to write code to find the longest corridor this problem is isomorphic to that problem. However he had recently been able to write code, albeit not with a fully generalised solution, to solve the smallest stack of beepers Seq2 – Q3 task and was now using the unit tests more effectively. As a result of his discussion with the interviewer after solving Seq2 – Q3 he was starting to appreciate that while a solution might appear to be correct it may not always be able to cope with a new scenario and that he should try to build a general solution.	

Data

1. Think aloud:

As Luke had been advised to do in previous meetings, he started solving this question by writing utility methods to perform the basic robot operations such as turning a robot right and turning a robot around (Figure 6.29(left), steps1→ 2). Then he started to write the method to find the shortest corridor. Luke did not retrieve a fully formed schema for counting the length of the first corridor but instead appears to have retrieved smaller sub-

plans and joined those plans. Firstly he recognized the need to iterate in order to move the robot forward then he verbalised:

“Before while loop, we need to do int [integer], current equal zero, then we need to increment current corridor, then when that is done, then sets.”

As a result, Luke realised that a gatherer variable was required and that that variable `currentCorr` should initially be set to zero and increased each time as the robot moves (Figure 6.29(left), steps3 → 5). And then he defined a most wanted holder variable `smallest` and assigned that to be the length of the first corridor (see Figure 6.29(right), steps6 →7). He then without thinking aloud or hesitating wrote a sequence of commands to reset the gatherer variable to zero ready to count the length of the remaining corridors and return the robot back to the start of the first corridor and finally orient the robot to face north (see Figure 6.29(right), step8). He did this without any evidence that he had read or traced any part(s) of his code.

After a pause, Luke added some more code (see Figure 6.29(right), step9 (A &B)). And then he started to read his code and verbalised:

“I need to check if it right , So it gonna test the first corridor and set it to the smallest, then turn around , then turn right , after move forwards , it gonna turn right, it gonna forwards forwards, turn right, test next corridor, after done that I need to compare. ”

Then, he continued to add another set of Java commands as shown in see Figure 6.29(right), step10. Luke compiled his code twice and he easily fixed the two errors in his code, using the compiler feedback, by adding brackets to the end of the call to the `turnRobotAround` method and adding a `RETEUN` statement to the method (Figure 6.29 (right), steps11 and12 respectively). Luke ran the supplied unit tests and watched the robot moving across the corridor, he quickly realised that he had forgotten to add the method call which would put the robot in the right direction to move up to the start of the next corridor (see Figure 6.29(right), step13). He re-ran the tests and was surprised when all the tests failed for the second time. He started to read the unit test messages and part of his code, Luke verbalised:

“For the first test [scenario] expected five but was four. For the second one [scenario] expected seven but was six. Test the first one[scenario] is set to the smallest and after that test the next one, current less that smallest and smallest equal current and I forget to add the equal statement set the current corridor to zero.”

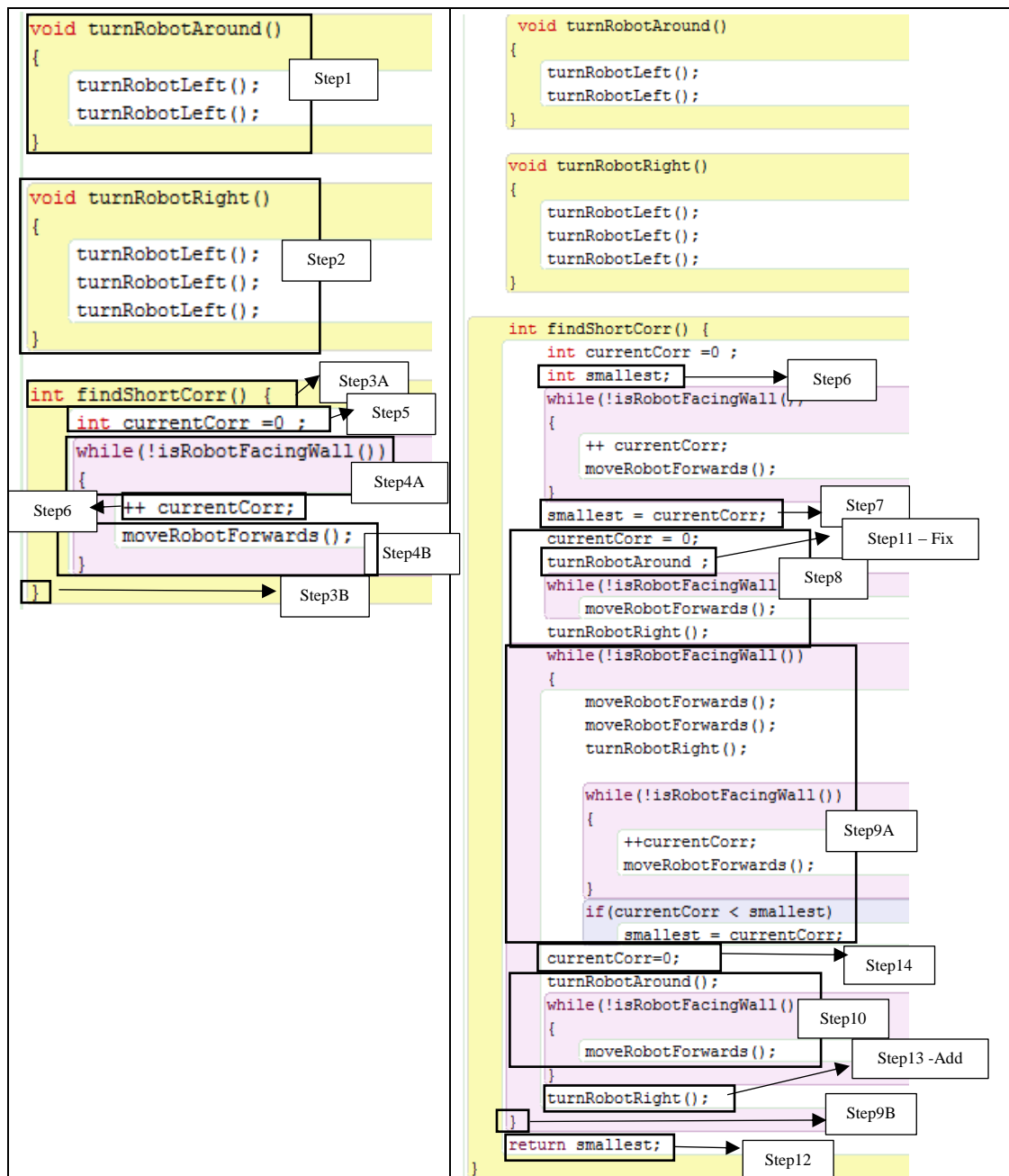


Figure 6.29 Luke's first and second screen images for the shortest corridor

Luke managed to fix one of his mistakes on his own by adding the line of code shown in Figure 6.29(right) and step14. Luke ran the supplied unit tests to verify the correctness of his solution and he was again surprised when all the tests failed. He got the same test results. He again read the test output and his code:

"In this case [scenario] the same problem. Expected four but was five [this scenario], for the second one [scenario] seven but was six, same before so um, so now, set it to zero, and a while loop at the current corridor, if current corridor is less than smallest, smallest equal current corridor and current corridor is zero. The difference is one, the code it should be that is because [pause]. Current corridor started from zero, current plus plus as the robot keep moving and counting".

After a long pause, Luke felt that he reached a dead-end and asked for help — “*I need your help*”.

2. Scaffolding:

Interviewer: “*If you think about the two tests, what is expected for the first test and what is expected for the second?*”

Luke: “*Expected for first is five and I have got four, and then for the second is seven and I have got six, always the difference is one*”

Interviewer: “*Yes the difference is always one, why it is always one?*”

Luke: “*Um*”

Then the interviewer redirected Luke to trace through his code. Figure 6.30 shows what Luke’s trace. Desk checking his code helped Luke identify the problem and he was able to update his code by setting the gatherer variable to one instead of zero. The interviewer used one of the unit test scenario as an example for tracing his code.

Current Corridor	Smallest
a	11
b	3
c	4
d	4
e	4
f	4

Figure 6.30 Trace-table for Luke’s code for the shortest corridor

3. Retrospection:

The following is part of the conversation between the interviewer and Luke:

Interviewer: “*Have you seen this question before?*”

Luke: “*No, not this question. Um, I just solved the one with smallest stack of beepers [pause], ah I think also with corridor. I just remembered. Is that right?*”

Interviewer: “*What was the most difficult part for solving this question?*”

Luke: “*Um, [pause] I’m still confused between counting the beepers and length of corridor*”

6.3.6. Smallest Element in a 1D Array (Seq3 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Unsurprised	
Strategies	Sequential	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	7 minutes and 2 seconds
	<i># of compilation</i>	3
	<i># of execution</i>	1
Intervention	None	
Timing	Week eleven of the P1	
Important observations with respect to prior sessions	Luke did not encounter any significant problems when solving the smallest stack of beepers. It is important to note that although his code passed the tests was not generalised, connected or integrated.	

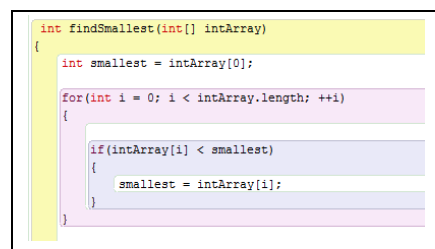
Data

1. Think aloud:

Luke began by reading the problem and immediately started to write the code line by line. Luke started with the method header with an array of type integer as the parameter and which returned an integer. He then defined the most wanted holder variable `smallest`. Luke hesitated as to what the initial value of `smallest` should be and verbalised:

“Integer smallest equals ... [long pause] ... um zero ... [long pause] ... smallest equal int array of zero [intArray[0]]”

After the above utterance, Luke made his decision and set the value of the most wanted holder variable to the first element of the array, followed by the FOR-loop statement that consisted of the stepper variable `i`. Inside the FOR-loop block, Luke added an IF-block as shown in Figure 6.31. There was a long pause before he added the less than operator (`<`) to the IF-block suggesting he was having to think carefully about which operator was appropriate less than or greater than.



```

int findSmallest(int[] intArray)
{
    int smallest = intArray[0];
    for(int i = 0; i < intArray.length; ++i)
    {
        if(intArray[i] < smallest)
        {
            smallest = intArray[i];
        }
    }
}

```

Figure 6.31 Luke’s screen image for the smallest element in a 1D array

Luke pointed to the first assignment statement and then verbalised:

“I need to change this value, let me try it”

He updated that assignment statement from `int smallest = intArray[0];` to `int smallest;`. Luke compiled his code. He did not show any surprise when he received a syntax error. Luke verbalised:

“So I will change it to equals first index array, let me try it again”

Luke directly updated that line of code and wrote `int smallest=intArray[1];`, after that, without hesitation, he changed 1 to 0 and verbalised:

“Smallest equal to the first element of the array, the first element come with index one, no no with index zero”

Luke compiled his code for the second time and he easily fixed the next syntax error by adding the RETURN statement to the end of the method body. Luke compiled his code for a third time and then ran his program — all the tests passed.

2. Retrospection:

The interviewer questioned Luke about the two long pauses while he was writing his code. The first pause was related to selecting the correct initial value for the most wanted holder variable. The second pause was when he was deciding which relational operator to use in the IF-block. Luke responded that he was thinking about how he solved the smallest stack of beepers problem. Clearly he was using his knowledge gained from solving the smallest stack of beepers to try and solve this problem and he saw similarities between the two tasks. During the second pause he said that he was thinking about the direction of the relational operator.

6.3.7. Index of the Largest Element in a 1D Array (Seq3 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	Read messages and test code
	<i>Time on task</i>	9 minutes and 20 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	None	
Timing	Luke solved this question in his eleventh week eleven of P1. Directly after solving the smallest element in a one-dimensional array task.	
Important observations with respect to prior sessions	During solving Seq3 – Q2, two long pauses was recorded. Firstly, before Luke initialised the value of the most wanted holder variable. Secondly, before Luke added the relational operator (<).	

Data

1. Think aloud:

Luke began by reading the problem and immediately started to write the code. Luke first wrote the method header and then defined two most wanted holder variables. The first most wanted holder variable he called `largestIndex` and set it to zero. The second most wanted holder variable, he called `largest`. The function of the second most wanted holder variable was to store the value of the first element of the array. Then, he continued writing a line by line Java commands as shown in Figure 6.32 (left).

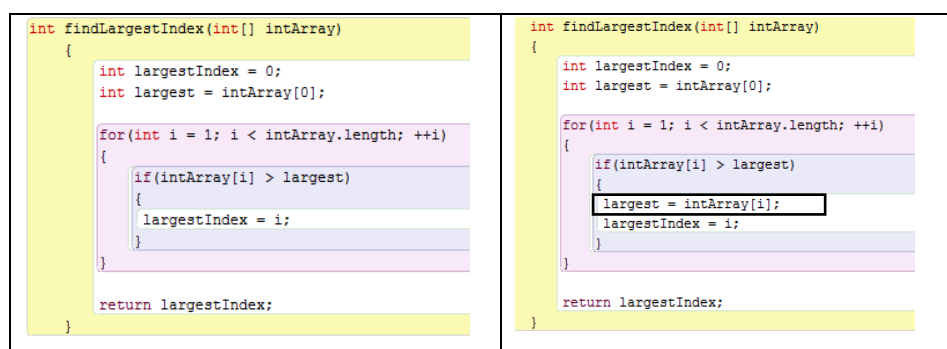
Finally, Luke ran the supplied unit tests. He discovered that one test failed and the other passed. Therefore, he started to read the unit test message for the failed test and then he verbalised:

“Expected two but was five”

After that he viewed the unit test file – `findLargestIndex(new int[] {0, 1, 2, 3, -4})`. Then he verbalised:

“For the first test result three correct [pause], the largest number in index three, ah I’m checking against the initial element let me see.”

As a result, Luke updated his code by adding the following variable assignment: `largest = intArray[i];` immediately after the IF-statement (see Figure 6.32 (right)). Finally Luke compiled and ran the test units to verify the correctness of his solution.



```
int findLargestIndex(int[] intArray)
{
    int largestIndex = 0;
    int largest = intArray[0];

    for(int i = 1; i < intArray.length; ++i)
    {
        if(intArray[i] > largest)
        {
            largestIndex = i;
        }
    }

    return largestIndex;
}
```

```
int findLargestIndex(int[] intArray)
{
    int largestIndex = 0;
    int largest = intArray[0];

    for(int i = 1; i < intArray.length; ++i)
    {
        if(intArray[i] > largest)
        {
            largest = intArray[i];
            largestIndex = i;
        }
    }

    return largestIndex;
}
```

Figure 6.32 Luke’s first and second screen images for find the largest index

2. Retrospection:

Luke focused on talking about two kinds of scaffolding. Firstly on how solving the previous question had helped him to solve this question. Secondly, how the unit tests helped him to correctly fix the mistakes he had in his code.

6.3.8. Checking if Beeper Stacks are sorted in Ascending Order by Size of the Stack (Seq2 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Happy	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	10 minutes and 15 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	After Luke had finished the P1 course.	
Important observations with respect to prior sessions	For Seq2 – Q1, Luke was clearly outside of his ZPD and he was supplied the model answer for this question. Luke was able to solve different programming tasks in the same sequence which suggests that he learnt from the model answer and was able to apply that learning to new situations. Luke solved (Seq3 – Q1, Appendix A) with interviewer assistance.	

Data

Think aloud:

Luke started to verbalise and write his solution shown in Figure 6.33 (left) line by line.

“Integer x equal zero, [pause] I think it is fine to use WHILE loop, [pause], the next step [pause], I need to move, and count all, as usual to count all, pick up, and add. After finish counting, I gonna set x to zero”

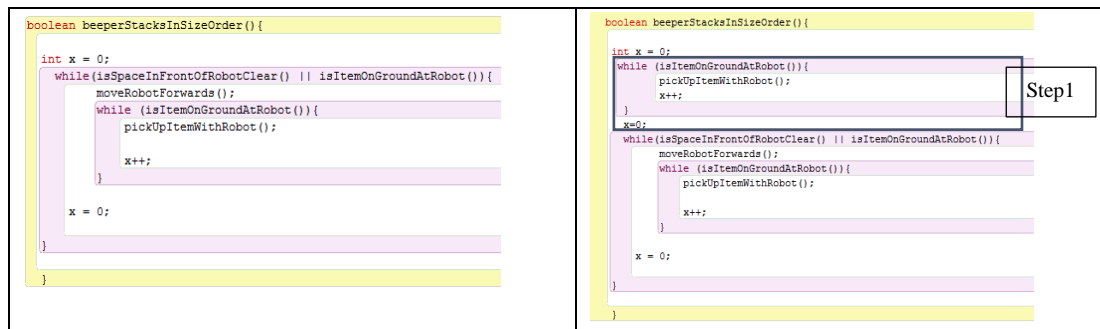


Figure 6.33 Luke’s first and second screen images for checking if beepers stacks are sorted

After a pause, Luke verbalised:

“Ah, to pick up the beeper in the first location”

After the above utterance, Luke started to update his code so that it allowed the robot to also pick up and count beepers at the first location Figure 6.33 (right). Then Luke verbalised:

“I’m not sure the length of corridor [pause], let use make it seven. Let me count the length. I know how to do it”

After the above utterance, Luke continued coding a block of code to move the robot down the first corridor and back and count the length of that corridor (see Figure 6.34, step2). Then, Luke decided he needed to store the counted beepers for each stack count in a one-dimensional array (see Figure 6.34, steps3→5).

Then Luke verbalised:

“I need a loop to compare each of these values”

After the above utterance, Luke added code to check if the elements of the array is sorted ascending (Figure 6.34, step6). Finally Luke compiled and ran his code to verify the correctness of his solution. He encountered no difficulties in solving this problem.

```

boolean beeperStacksInSizeOrder() {
    int x = 0;
    int y=1;
    while (isSpaceInFrontOfRobotClear()) {
        moveRobotForwards();
        y++;
    }
    turnRobotLeft();
    turnRobotLeft();
    while (isSpaceInFrontOfRobotClear()) {
        moveRobotForwards();
        turnRobotLeft();
        turnRobotLeft();
    }
    int [] array = new int [y];
    int z=0;
    while (isItemOnGroundAtRobot()) {
        pickUpItemWithRobot();
        x++;
    }
    array [z] = x;
    z++;
    z=0;
    while (isSpaceInFrontOfRobotClear() || isItemOnGroundAtRobot()) {
        moveRobotForwards();
        while (isItemOnGroundAtRobot()) {
            pickUpItemWithRobot();
            x++;
        }
        array [z] = x;
        z++;
        x = 0;
    }
    for (x = 1; x < array.length; x++) {
        if (array[x - 1] > array [x])
            break;
    }
    if (x == array.length)
        return true;
    else
        return false;
}
    
```

The image shows a screenshot of code with several sections highlighted and labeled with boxes and arrows:

- Step 2:** Points to the first while loop that moves the robot forward and turns left.
- Step 3:** Points to the array declaration: `int [] array = new int [y];`
- Step 4:** Points to the first assignment: `array [z] = x;`
- Step 5:** Points to the second assignment: `array [z] = x;`
- Step 6:** Points to the for loop that checks if the array is sorted: `for (x = 1; x < array.length; x++) { ... }`

Figure 6.34 Luke’s third screen image for checking if beepers stacks are sorted

6.3.9. Largest Element in a 2D Array Task (Seq4 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Happy	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	Read the test message
	<i>Time on task</i>	10 minutes and 3 seconds
	<i># of compilation</i>	3
	<i># of execution</i>	2
Intervention	None	
Timing	Week six of the P2 course	
Important observations with respect to prior sessions	Luke had no problem solving a related and far transfer question the smallest element in a one-dimensional array (Seq3 – Q2).	

Data

1. Think aloud:

Luke began by writing the method header with an array of type integer as a parameter. He defined a nested FOR-loop block, followed by an IF-block, and finally he added a RETURN Java command as shown in Figure 6.35 (left).

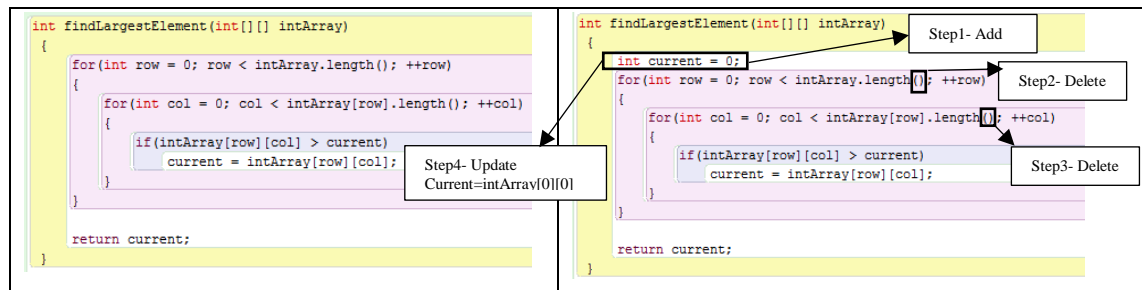


Figure 6.35 Luke's first and second screen images for the largest element in a 2D array

Before Luke compiled his code he verbalised:

"I just remembered that I'm checking against something, I need that thing to check against [pause]"

After the above utterance, he defined the most wanted holder variable `current` after the method signature and set its value to zero as shown in Figure 6.35, step1 (right).

When Luke compiled his code, he got a syntax error. Luke verbalised:

"I've still got a problem with nested loops, I need to practise more and more"

Luke easily fixed the error related to checking the length of the row and the column of the two-dimensional array (Figure 6.35, step2→3) (right). Luke ran the unit tests for the first time One out of three supplied unit tests failed. Luke read the test message and verbalised:

"Expected -1 but was 0, and that's because the initialisation of the current [variable], but I did not think about the negative number so I should set it to ah [long pause] I want to set it to value of [pause], minimum is [long pause], the value of the first one [the first number in the array]."

At this stage, Luke expressed doubt about the initial value of the most wanted holder variable and finally he decided to set its value to the first element of the two-dimensional array and ran the unit tests to make sure that he made the right decision – which he had.

2. Retrospection:

The following is part of the conversation between the interviewer and Luke:

Interviewer: *"Before you compiled your code, you decided to define current variable as the last Java command in your code, is that right?"*

Luke: “Yes, I’m checking against something, and I thought it is zero, but I did not think about the negative number until I read the test message”

Interviewer: “Have you seen this question before?”

Luke: “Yep, I think in a one-dimensional array, I think the question was either the smallest or largest element”

Interviewer: “Did you realise that this question and the smallest element in a one-dimensional array had identical sub goals in common when you started solving the program?”

Luke: “Nope, but when I ran my code, my test failed I did.”

6.3.10. Column in a 2D Which Contains a Smallest Number (Seq4 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Happy	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	3 minutes
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week Six of the P2 course	
Important observations with respect to prior sessions	Successfully solved the previous isomorphic question which required writing code to find the largest element in a 2D array (Seq4 – Q2).	

Data

Think aloud:

Luke read the problem and immediately started to verbalise writing the completed solution code with minimal effort (Figure 6.36):

“Int [integer] find smallest index that takes two-dimensional array. I know I need int [integer] smallest, smallest equal array of zero zero [array[0][0]], and FOR- loop int row equal zero, row less than array length, row plus plus. I need another FOR- loop, column equal zero less row length column plus plus [pause] another int [integer] variable because I need [pause] the index of smallest column [pause] this question is different than the first one [pause]. int [integer] smallest column [smallestCol] equal zero [pause]. If array row column [array[row][col]] [pause] less than smallest smallest equal array row column [array[row][col]]. Close that. Close that. Close that. Return, I need to run the test”

Finally, Luke compiled and ran the unit tests to verify the correctness of his solution – all the tests passed.

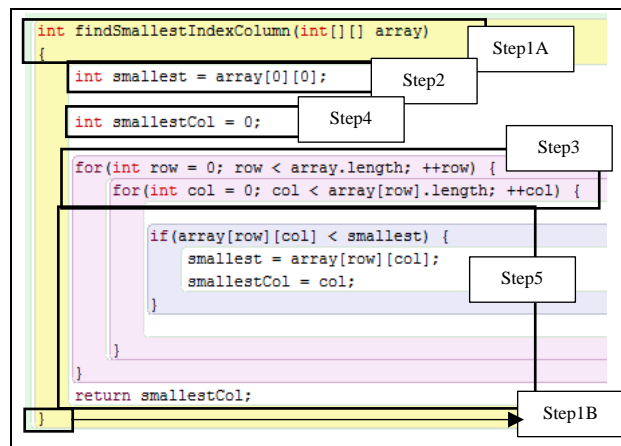


Figure 6.36 Luke’s screen image for column in a 2D array which contains a smallest number

6.3.11. Print the Highest Mark and Name of Every Student in a Collection of Student Objects (Seq5 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Happy	
Strategies	Sequential	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	7 minutes and 17 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week seven of the P2 course	
Important observations with respect to prior sessions	Luke had not encountered any significant issues when solving problems which required iteration and searching of 1D and 2D arrays.	

Data

Think aloud:

Luke began by reading the problem and immediately started to verbalise while writing his code (see Figure 6.37, steps1 and 2):

“So method void, highest student’s details [HighestStudentMark()], I’m going to add integer highest equals, no this not 1D [Luke deleted the line of code he was writing and added [highestMark =]]. For integer i equal zero, i less than student size [student.size()], this FOR- loop end with i plus plus. Close that. Int [integer] highest equals should be zero [pause] no should be equals students dot get dot student mark i [students.get(i).studentMark[i]] [pause] no equal to mark zero [students.get(i).studentMark[0]]. For int [integer] x equals 0, x less than [pause], so x should be less than students dot student mark and dot length [students.get(i).studentMark.length], plus plus x”.

Then Luke updated the stepper variable x from zero to one as shown in Figure 6.37, step3. After that he continued to verbalise while writing a line by line Java commands (see Figure 6.37, step4):

“If students [pause] students dot get i dot student mark x [students.get(i).studentMark[x]] less than [pause] no greater than highest mark [highestMark]. Highest mark equal to students dot get i dot student mark x [students.get(i).studentMark[x]]. Close that. Close that. Print student name. Print student mark. I need to compile and run the test”

Finally, Luke compiled and ran the unit tests to verify the correctness of his solution.

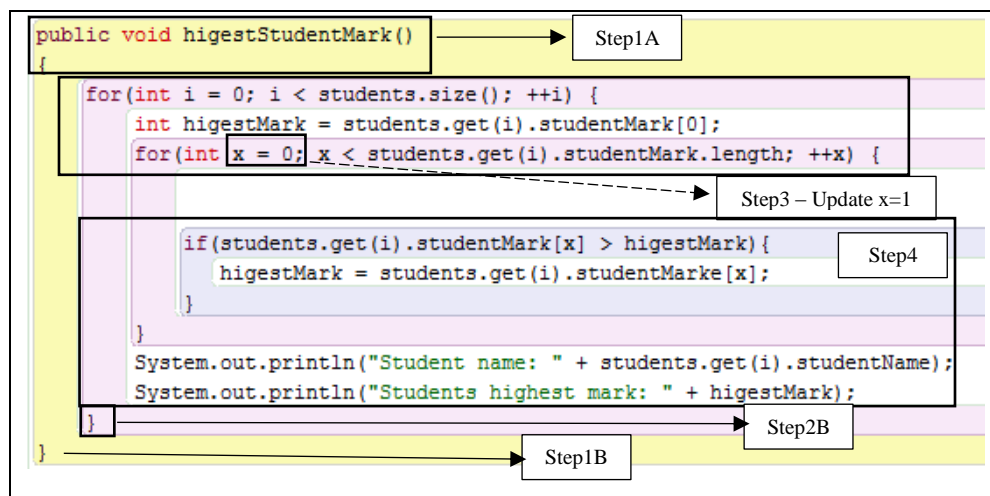


Figure 6.37 Luke’s screen image for highest student mark in a collection of student objects

6.4. Kasper’s Think Aloud Sessions

6.4.1. Summary

Kasper was able to solve 11 out of 12 questions during the think aloud sessions when studying P1 but only three out of seven questions during the P2 sessions. He was in the second quartile of students in P1 and in the third quartile for P2. While, he was able to solve many of the tasks in this study, Kasper was consistently observed to have trouble mastering the Java commands he learned, especially during the P2 course. Kasper showed all the signs of being a tinkerer when it came to writing code and as soon as he faced any difficulty he resorted to trial and error programming. He also seemed to lack motivation and engagement. He regularly postponed sessions and his lack of application is reflected by his lack of progress. Kasper frequently demonstrated during the meeting sessions that he did not consider tracing to be an important skill.

6.4.2. Counting the Length of One Corridor (Seq1 – Q1)

Encoding

Question	Solved	
Behaviours	Tinkerer	
Emotion	Confused	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging
	<i>Unit test</i>	Read message from one Robot World scenario only
	<i>Time on task</i>	8 minutes and 8 seconds
	<i># of compilation</i>	6
	<i># of execution</i>	6
Intervention	None	
Timing	Week four of P1	
Important observations with respect to prior sessions		

Data

1. Think aloud:

Kasper started by writing a WHILE-loop statement, followed by a Robot World command that allowed the robot to move across the corridor (Figure 6.38 (top), step1). Then, he defined a gatherer variable `lengthOfCorridor` at the beginning of the method and set its value to zero (Figure 6.38 (top), step2). After that he added a line which increased the gatherer's value by one inside the WHILE-loop block (Figure 6.38 (top), step3). Kasper hesitated when deciding to print the number of the squares in a single corridor. He had doubt as to whether he should multiply the gatherer value by two or multiply the gatherer variable by itself. It became clear later in the think aloud that he had confused counting the cells or squares in a corridor with the squaring a number — this seems to be an issue with English comprehension rather than with writing code. Finally, he the decision to multiply the gatherer variable by itself and then added the PRINT statement (see Figure 6.38 (top), steps4→5). After a pause, Kasper decided to define another variable to store the result of the multiplication and then he updated the PRINT-statement according (see Figure 6.38 (bottom), steps6→7). Kasper ran the supplied unit tests and all tests failed. He verbalised — “fail”

Kasper then focused on one of the robot scenarios (a corridor of length 5), ignored the test's messages, and verbalised:

“Start from zero, one, two, three, four [pause], I think after that”

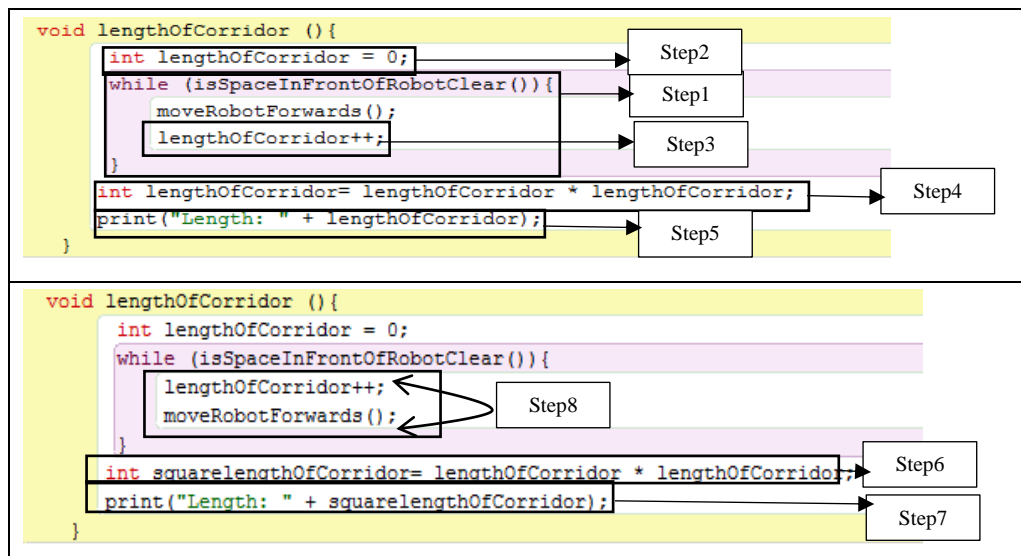


Figure 6.38 Kasper's first and second screen images for counting the length of one corridor

Kasper started directly to swap the position of the two Java commands `moveRobotForwards()` and `lengthOfCorridor++` (Figure 6.38 (bottom), step8 c.f. Figure 6.38 (top)). Kasper was surprised when the supplied unit tests failed for a second time. He started for the first time to read one of the unit test messages (the test message for a corridor of length 10) and verbalised:

"Expected ten but was eighty one"

Kasper started to re-read the question again and verbalise:

"It is not square, it just the square of length. This line is wrong"

He updated his code by deleting the Java command that squared the gatherer variable and then updated the PRINT-statement accordingly. For the second time he swapped the position of the two Java commands `moveRobotForwards()` and `lengthOfCorridor++` back to their original position as shown in Figure 6.38 (top). Kasper ran his code for a third time and again all the tests failed. Kasper started to read the unit test messages for the corridor of length 10 again and verbalised:

"Expected ten but was nine"

Kasper made two more attempts swapping around the position of the two Java commands `moveRobotForwards()` and `lengthOfCorridor++` and running his code focusing on the same test's output ignoring the other unit tests. Finally he increased the value of the gatherer variable before the PRINT statement (Figure 6.39). Kasper compiled his code for the sixth time and ran his program. This time all the tests passed.


```

void lengthOfCorridor (){
  int lengthOfCorridor = 0;
  while (isSpaceInFrontOfRobotClear()){
    moveRobotForwards();
    lengthOfCorridor++;
  }
  lengthOfCorridor++;
  print("Length: " + lengthOfCorridor);
}

```

Figure 6.39 Kasper’s final screen image for counting the length of one corridor

2. Retrospection:

The following is part of the conversation between the interviewer and Kasper:

Interviewer: “Did you plan before you started?”

Kasper: “The first thing, I was thinking about moving the robot, then I was thinking about counting”

Kasper confirmed that he had solved similar question before

Kasper: “Because I have the problem before, I go back and through about the previous problem, the similar code from that maybe one less [Kasper is referring to counting the beepers at a single pile]”

The interviewer reviewed the video tape with Kasper focusing on how Kasper could avoid problems resulting from the lack of focus on all possible robot scenarios and discussed the importance of reading and understanding all the unit tests.

6.4.3. Comparing the Length of Two Corridors (Seq1 – Q2)

Encoding

Question	Solved
Behaviours	Mover
Emotion	Indiscernible
Strategies	Stepwise design
Activities	<i>Planning</i> <i>Tracing</i> <i>Unit test</i> <i>Time on task</i> <i># of compilation</i> <i># of execution</i>
	 Visual debugging, and Hand gestures Read the unit test message for one Robot World scenario. 17 minutes 52 seconds 5 4
Intervention	None
Timing	After the sixth week of P1.
Important observations with respect to prior sessions	In the previous meeting Kasper was not confident about counting the length of the corridor – he was confused and this confusion led to a trial and error approach to programming (Seq1 – Q1). This confusion was evident again in for this task.

Data

1. Think aloud:

Kasper began by reading the problem. For this question a method header had been provided so it is possible to run the unit tests before any code has been written. Kasper ran the supplied unit tests to see the initial robot scenarios. After examining the starting Robot Worlds he read the question again. Kasper first declared and initialised two gatherer variables, one for each corridor length, to one. He then wrote a WHILE-loop block to move the robot and count the length of the first corridor. After closing the WHILE-loop block bracket, Kasper added a command to increment by one the gatherer variable for counting the length of the first corridor and he verbalised:

“Just to calculate then go back to the upper corridor, I just count the last square, and then go back to the upper corridor, so turn left turn left”

Kasper continue write his solution line by line adding code to turn the robot and return it back along the corridor, reorient the robot and move it to the start of the next corridor (see Figure 6.40 (left)). He then tested his code by running one of the supplied unit tests to ensure that the robot moved correctly across the world:

“I just need to test this [he then ran the code and paused to watch the robot moving] ... then left, left, left then up [as he watched the robot move he articulated the movement he was seeing and also waved his hand in the air in the direction the robot was turning]”

Based on visualising the robot moving across the world, Kasper realised that before counting the length of the second corridor, he needed to add Java commands that allowed the robot to face east. He then copied and pasted the Java commands for counting the length of the first corridor and edited the gatherer variable’s name so that when the loop ran it would store the length of the second corridor in the correct variable. Finally, Kasper used three separate IF-blocks to compare the lengths of the two corridors (see Figure 6.40 (right)). Kasper compiled his code which gave a `reached end file of while parsing` syntax error. Kasper fixed his code by adding the missing method close brace.

Kasper ran the supplied unit tests and all the tests failed. He started to read one of the test messages:

“Corridor zero is the longest expected nine but was ten, it supposes to start from zero”

Kasper fixed his code by initialising the gatherer variables to zero instead of one.

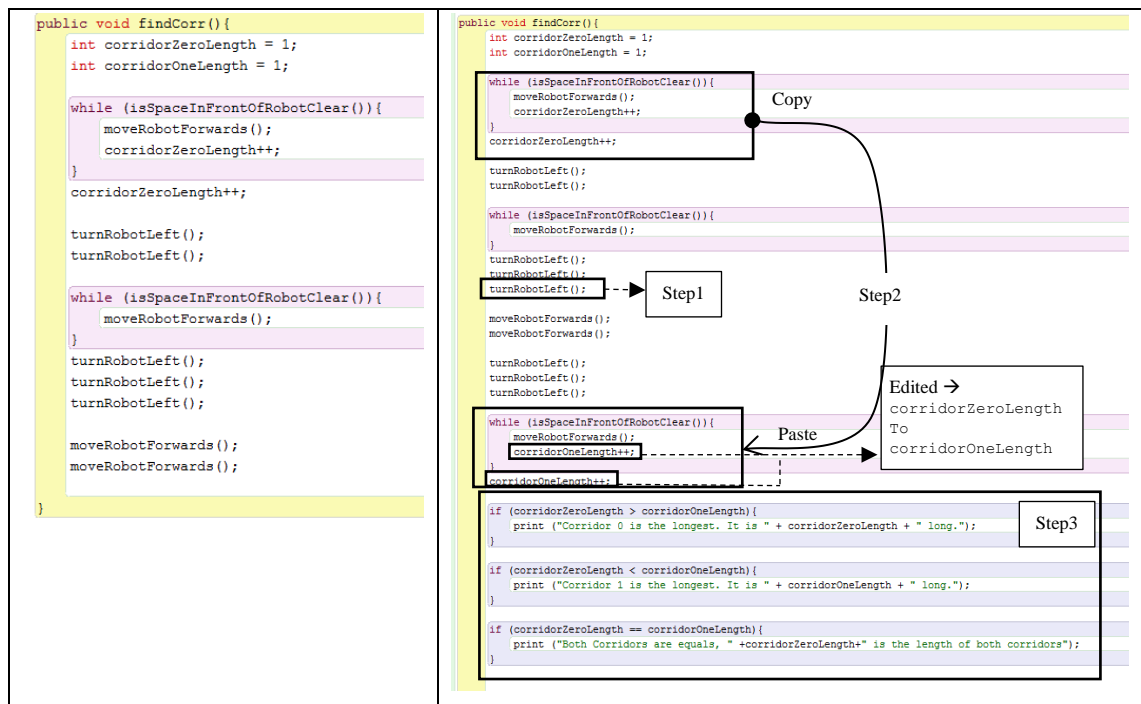


Figure 6.40 Kasper's first and second screen images for comparing the length of two corridors

2. Retrospection:

The following is part of the conversation between the interviewer and Kasper:

Interviewer: "Had you seen this question before?"

Kasper: "Nope"

Interviewer: "Why didn't you use an IF-ELSE block instead of three IF statements?"

Kasper: "Because this is much simpler"

Interviewer: "Have you solved a question that required you to use IF-ELSE statements before?"

Kasper: "Yes in the test"

Interviewer: "In the test or homework?"

Kasper: "Yes in the test because as I remember there was a condition to use IF-ELSE"

At the end of the session, the interviewer discussed with Kasper about the quality of his code and gave examples of when using an IF-ELSE IF block is a good idea.

6.4.4. Longest Corridor (Seq1 – Q3)

Encoding

Question	Solved	
Behaviours	Tinkerer	
Emotion	Indiscernible	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging and PRINT debugging
	<i>Unit test</i>	
	<i>Time on task</i>	15 minutes 25 seconds
	<i># of compilation</i>	4
	<i># of execution</i>	3
Intervention	Hint scaffolding – provided on request	
Timing	The second week of the P1 intra-semester break	
Important observations with respect to prior sessions	When solving earlier tasks (see Seq1 – Q1 and Seq1 – Q2), Kasper had expressed and had been observed having some doubt as to how to count the length a corridor. It also emerged during the retrospective interview for this task that Kasper had used IF-ELSE statements but still found it easier to fall back on several independent IF statements which were easier for him to understand.	

Data

1. Think aloud:

Kasper began by reading the problem and then he verbalised:

“So we make a method that returns an integer and we have to calculate the length of corridors so one corridor, the world is randomly changing, so we have to calculate the first one first.”

Kasper started by writing the method header. He followed this by initialising a gatherer variable `countLength` for counting the length of the first corridor and set its value to zero. Then he added a WHILE-loop block for counting and moving the robot across the first corridor. After closing the WHILE-loop block bracket, Kasper added a debugging PRINT statement to enable him to verify the correctness of the counting of the length of the first corridor (Figure 6.41(left), step1).

Kasper compiled his code which gave him one error. He easily fixed the syntax error by adding the missing RETURN statement (Figure 6.41(left), step2). He re-compiled and ran the supplied unit tests; all the tests failed. He proceeded his debug PRINT statement:

“One, two ... ten. Print nine but should be ten. One, two ... twelve. Print eleven but should twelve. One, two... six. Print five but should six, I miss to count one of them.”

Kasper easily updated his code increasing the gatherer variable by the one at the end of the WHILE-block command and then deleted the debug PRINT (Figure 6.41(left), steps3→4).

Kasper continued programming and verbalised (Figure 6.41(left), step5):

“These for corridor one, now I need to check if there is another corridor. Once it is get back it is facing west, make it turn left, left, left. We check if it block or locked. Move, move to next corridor. Once it is finish this corridor make it face east, so left, left, left. And then count this one [count the second corridor].”

After a pause, Kasper concluded that he needed to define three gatherer variables, for the three possible corridors, instead of one — *“I think we need three integers”*.

It did not occur to him that there might be more than three corridors depending on the height of the Robot World.

Kasper started to update the name of the first gatherer variable from `countLength` to `countLengthCorr0` and then he defined `countLengthCorr1` and `countLengthCorr2` and set them to zero too. He then copied and pasted the Java commands for counting the length of the first corridor and renamed the gatherer variable to count the length of the second corridor. He repeated the same process for counting the length of the third corridor (Figure 6.41 (right)). This exactly the same process as he had used in the previous task when trying to find the longest of two corridors (Seq1 – Q2). After a long pause, Kasper verbalised:

“We need to check to store the value of the largest corridor but I have no idea about how to, so if [pause].”

As he completed the above utterance, he defined the most wanted holder variable `longest` after three gatherer variables and set its value to zero. After a long pause, Kasper changed his mind and he decided to delete the most holder variable definition. After another pause, he decided to define three IF-blocks and deleted the method’s RETURN statement. Then he compiled his code and verbalised:

“Missing return statement I need to use the IF-ELSE statement but I have no idea, ah, could you help me?”

2. Scaffolding:

The interviewer started to explain to Kasper the structure of an IF-ELSE block with examples (hint - soft scaffolding). As a result of this intervention Kasper was able to fix his error.

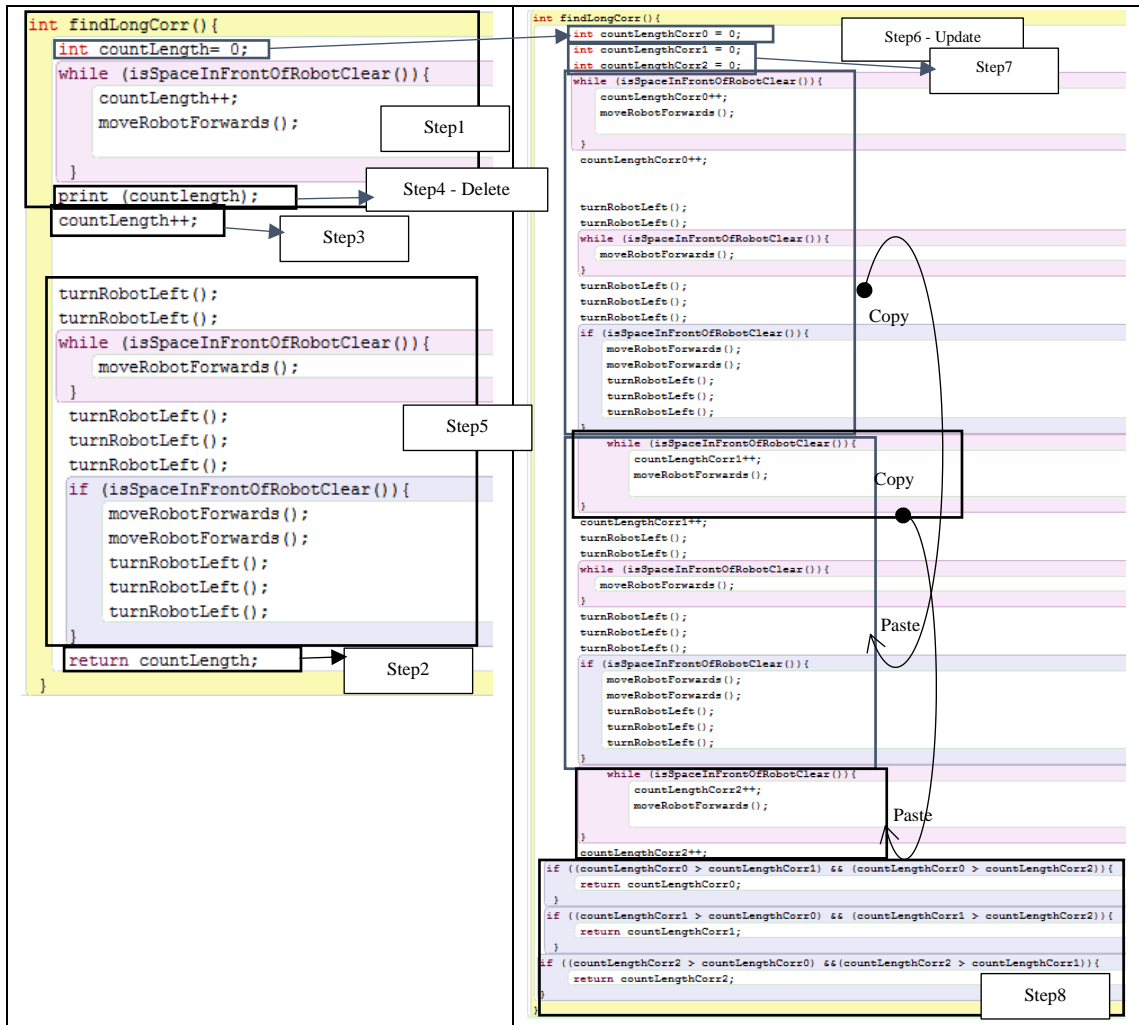


Figure 6.41 Kasper's first and second screen images for the longest corridor

3. Retrospection:

The following is part of the conversation between the interviewer and Kasper:

Interviewer: "When you started solving this question, you didn't think that there may be four, five or six corridors?"

Kasper: "Yes"

Interviewer: "What was your problem for solving this question? Did it begin when you started to compare the length of the three corridors?"

Kasper: "This was the difficult part, but I get it now."

Interviewer: "Have you seen similar questions using an IF-ELSE statement?"

Kasper: "Yes, last meeting. Now I remembered."

6.4.5. Smallest Stack of Beepers (Seq2 – Q3)

Encoding

Question	Solved	
Behaviours	Tinkerer	
Emotion	Confused	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging, Mental tracing , and PRINT debugging
	<i>Unit test</i>	
	<i>Time on task</i>	16 minutes and 46 seconds
	<i># of compilation</i>	6
	<i># of execution</i>	6
Intervention	“General prompt” scaffolding – provided on request	
Timing	The second week of the P1 intra-semester break	
Important observations with respect to prior sessions	Kasper easily recognised the link between this question and the longest corridor problem. However, he found it difficult to transfer his knowledge may be because of the low the quality of the answer code he wrote for the longest corridor question. In the longest corridor solution code, Kasper defined three gatherer variables, one variable for each corridor and then used three separate loops to count and finally compare the three values of the gatherer variables in order to find the longest corridor.	

Data

1. Think aloud:

Kasper started by writing the method signature and then he verbalised:

“I need to pick up the first [beepers at the first location]”

Instead of recalling the schema for picking up beepers at the first location, Kasper recalled the schema for moving the robot across the corridor (Figure 6.42 (left)). Kasper ran the supplied unit tests, and visualised the robot moving. He then started to verbalise while updating his code (see Figure 6.42 (right), steps 1→2):

“The robot should pick up and count beepers on the way. So we put another while statement to pick up the items. Then I need variable to count, and initial that to zero.”

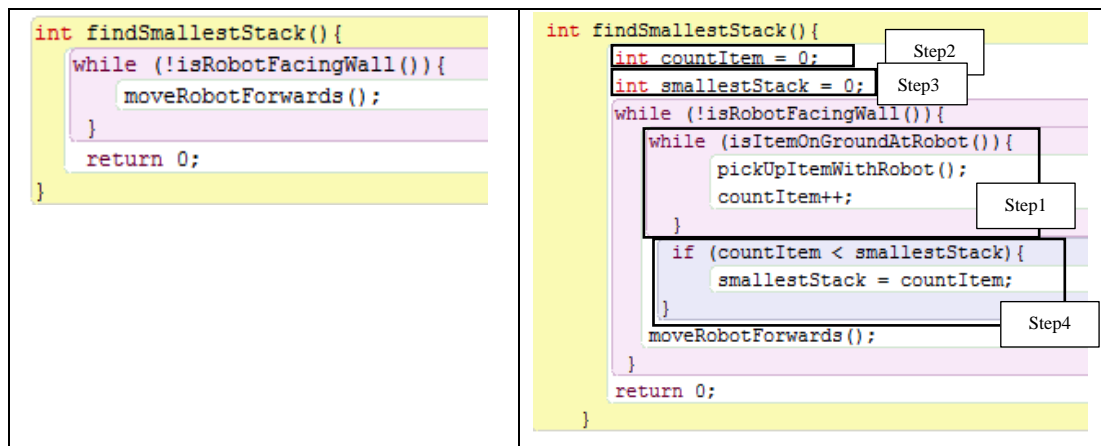


Figure 6.42 Kasper’s first and second screen images for the smallest stack of beepers

After that he started to read his code and verbalised his plan for the next steps:

“Once we count that, move forwards to count the another one, so ah, while is robot not facing wall, while item on ground at robot pick up the first one move forwards, and start again, repeated the loop, after doing that, after picking the second one, should be compare it with the first one, so I need to define another one, ah [pause].”

After a pause, Kasper defined the most wanted holder variable `smallestStack` and set its value to zero. After a long pause, Kasper started verbalising while writing an IF-block followed by reading and planning the next steps:

“So I will compare it, so if ah [long pause], if smaller stack is smaller than [after writing `smallestStack <`, he deleted the line]. [Pause] If count item greater than smallest stack [after writing `>`, he change it to `<`]. If count item smallest than smallest stack, then we make smallest stack equal to count item and then move forward and do again but we need to set the counter to zero [pause], if count item is smallest than smallest stack , smallest stack equal count item, smallest stack will be zero, so, [pause], I think [long pause], so we just assume, so count item is smaller than smallest stack, smallest stack equal count item, smallest stack is zero, um [pause].”

As shown from the above utterance, Kasper expressed doubt about the correct way for comparing the gatherer and most holder variables but finally he made his decision as shown in (see Figure 6.42 (right), step4). As shown from the above utterance and what he wrote in Figure 6.42 (right), step4 he sometimes verbalised information, and thought about possible options, that were not subsequently implemented in his code. For example: *“smallest stack will be zero”*

At this stage, Kasper began multitasking – updating and reading his writing (Figure 6.43(left)):

“I will make this smallest stack [student updated the variable name used for counting the beepers from `countItem` to `smallestStack`], so if the first one is the smallest, and we get the second one, count item less than smallest stack, smallest stack equal count item, I suppose the first one is the smallest stack, ah, then we need something to compare, I think, we have to find out what the first counter will be like. First, this could be the first count [highlighted the WHILE-block for counting the beepers using the mouse] and we need to show how to compare with the first one, and then make that back to zero , and store it somewhere.”


```

int findSmallestStack(){
int countItem = 0;
int smallestStack = 0;
while (!isRobotFacingWall()){
while (isItemOnGroundAtRobot()){
pickUpItemWithRobo; // Step5 - Update
smallestStack++;
}
smallestStack = countItem; // Step6 - Add
if (countItem < smallestStack){
smallestStack = countItem;
}
moveRobotForwards(); // Step10 - Move
}
return 0;
}

int findSmallestStack(){
int countItem = 0;
int smallestStack = 0;
while (!isRobotFacingWall()){
while (isItemOnGroundAtRobot()){
pickUpItemWithRobo; // Step7 - Update
countItem++;
}
print(countItem + "-" + smallestStack);
smallestStack = countItem; // Step8 - Add
if (countItem < smallestStack){
smallestStack = countItem;
}
moveRobotForwards(); // Step10 - Move
print(countItem + "-" + smallestStack);
countItem=0; // Step9 - Add
}
return 0;
}

```

Figure 6.43 Kasper's third and fourth screen images for the smallest stack of beepers

For the second time during the problem solving, Kasper verbalised information that was not implemented in his code. Kasper's fragile knowledge of the role of variables and of the comparison of variables is exemplified here (Figure 6.43(left)). After a pause (60 seconds), he re-read the assignment statement and IF-block he had written, and reasoned about their correctness:

"Smallest stack equal count item, if count item smaller than smallest stack, smallest stack equal count item. Smallest stack equal count item, if count item smaller than smallest stack, smallest stack equal count item. So the number of beepers should store in count item not small stack."

As a result of the above utterance, Kasper again updated the variable name for counting the beepers (using `countItem` instead of `smallestStack`). Then he ran the supplied unit tests. Kasper focused on watching the robot moving on the screen and ignored the unit test messages – *"Not picked up the last"*

After he completed the above utterance, he added the PRINT statement (see Figure 6.43 (right), step8) and re-ran the unit tests. He reasoned about his code's correctness focusing only on the result of the PRINT statement and ignoring the unit test messages and robot scenarios:

"Sixteen, sixteen, [print message for the first scenario]. Fifteen, fifteen [printed message for the second scenario]. Thirty two, thirty two [printed message for the third scenario]. I forget to set the counter to zero."

He then initialised the gatherer variable to zero (Figure 6.43(right), step9) and re-ran the supplied unit tests. Again Kasper focused only on the output of the print statement:

"Six, six [print message for the first scenario]. Two, two [printed message for the second scenario]. One, one [print message for the third scenario]. I think the problem with print statement."

Kasper changed the position of the PRINT statement (Figure 6.43(right), step10). And re-ran the unit tests verbalising:

“Now, print two different numbers, I’m confused, I need your help.”

2. Scaffolding:

The interviewer gave Kasper a robot image scenario, and a trace table with three columns headed `countItem`, `smallestStack`, and PRINT statement. The number of beepers at each location was recorded by the interviewer as [2, 4, 1, 3]. The interviewer used the data in the unit test scenario as an example for tracing (the first four piles from the second robot scenario). Kasper was asked to complete the trace table. Figure 6.44 shows what he wrote in the trace table.

CountItem	smallestStack	Print-statement
0	0	2-0
2	0	4-2
0	2	
4	0	
0	0	

Figure 6.44 Trace-table for Kasper’s fourth screen image for the smallest stack of beepers

The following is the conversation between Kasper and the interviewer once the trace table had been completed:

Kasper: *“Ah, same numbers, I’m confused”*

Interviewer: *“What do you think the problem is?”*

Kasper: *“I do not know”*

Then the interviewer started using the stepwise refinement technique using algorithms which Kasper had seen and implemented in previous think aloud sessions. The interviewer firstly asked him to write code to count the number of beepers at the first location in the corridor and store the result in the most wanted holder variable. Then the interviewer asked Kasper to extend that code so that it counted the beepers at each of the remaining stacks across a single corridor and compare the value of gatherer variable with the most wanted holder variable. Kasper was able to write his solution using a computer, but this solution contains redundant and unnecessary duplication of commands (see Figure 6.45).

```

int findSmallestStack(){
int countItem = 0;
int smallestStack = 0;

while (isItemOnGroundAtRobot()){
pickUpItemWithRobot();
countItem++;
}
smallestStack = countItem;
moveRobotForwards();
countItem = 0;
while (!isRobotFacingWall()){
while (isItemOnGroundAtRobot()){
pickUpItemWithRobot();
countItem++;
}
println(countItem + "-" + smallestStack)
if (countItem < smallestStack){
smallestStack = countItem;
}

moveRobotForwards();

countItem = 0;
}

while (isItemOnGroundAtRobot()){
pickUpItemWithRobot();
countItem++;
}
if (countItem < smallestStack){
smallestStack = countItem;
}

return smallestStack;
}

```

Figure 6.45 Kasper's final screen image for the smallest stack of beepers

3. Retrospection:

The following is part of the conversation between the interviewer and Kasper:

Interviewer: "Have you seen this question before?"

Kasper: "Yes"

Interviewer: "What was that question that you solved before?"

Kasper: "The biggest corridor"

Interviewer: "Yes that is right"

Interviewer: "Did you try to compare and contrast between what you had seen before and this question?"

Kasper: "I just remembered I need to count the first one, store the result. Then count and compare the new value with the old one, then I realised ah there is a problem to set the value of the smallest to zero."

At the end of the session, the interviewer gave Kasper feedback about the quality of his code and how he could further develop it.

6.4.6. Shortest Corridor (Seq1 – Q4)

Encoding

Question	Not solved	
Behaviours	Stopper	
Emotion	Confused	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging and Hand gestures
	<i>Unit test</i>	
	<i>Time on task</i>	16 minutes and 57 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	1
Intervention	Exact solution – provided on request	
Timing	Week nine of the P1 course	
Important observations with respect to prior sessions	Kasper had solved a similar isomorphic problem previously (the longest corridor). In solving the longest corridor task he had found it difficult to recall relevant prior knowledge. After nine weeks, Kasper struggled to recall the schema for counting the length of the corridor and returning back something that he should have had plenty of experience with – in fact it was difficult to do any programming without a solid understanding of the robot methods for navigating the Robot World. His answer code for the longest corridor contained redundancies and did not provide a generalised solution. This fragile understanding may have hindered Kasper’s ability to transfer his knowledge from longest corridor to shortest corridor.	

Data

1. Think aloud:

Kasper first read the problem and then verbalised while writing his code:

“I solved this question before, this question looked familiar to me. So return an integer, let us call it find shortest corridor, so I will make the robot move first. Reach end of the wall, once we do that we count, get the initial for the first corridor [pause] so [pause], so we will make this um say integer count equal 1. After move count just first the corridor count, um, I will do the rest”

Kasper did not retrieve a fully formed schema for counting the length of corridor but instead appears to have retrieved sub-plans and joined those plans. Firstly, he recognized the need to iterate in order to move the robot forward, then he realised a gatherer variable was required (see Figure 6.46(left)).

Kasper decided to add another WHILE-block after the first WHILE-block (see Figure 6.46(right), step1). Then he decided to add Java commands inside the first WHILE-block (see Figure 6.46 (right), step2). Kasper verbalised while continue writing his code and using his hand waving to the robot direction:

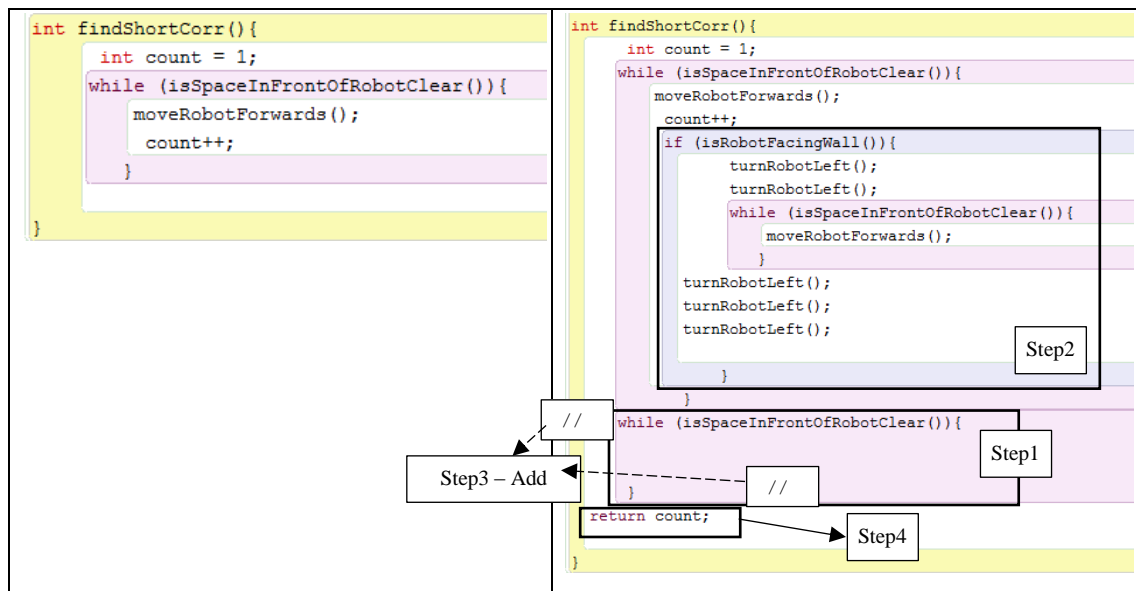


Figure 6.46 Kasper's first and second screen images for the shortest corridor

"If robot facing wall. Turn robot left twice. Then make it move to the other side, while is space in front of robot clear move back to the initial position [pause] um, that should not counted [long pause]. Comes there back to the other side, facing the wall, so should end in the loop lets me finish this. Turn left, left, left."

Kasper's fragile knowledge was evident due to the difficulty he experienced when trying to add java commands that returned the robot back along the corridor. Kasper commented the last WHILE-block (adding // and //, Figure 6.46 (right), step3) and compiled the code. He fixed a syntax error by adding a RETURN statement. He recompiled and ran the supplied unit tests. All the tests failed. Kasper started to visualise the robot moving across the world focusing on one scenario and ignoring the others. After a pause (62 seconds), he started to update his code (see Figure 6.47 (left)).

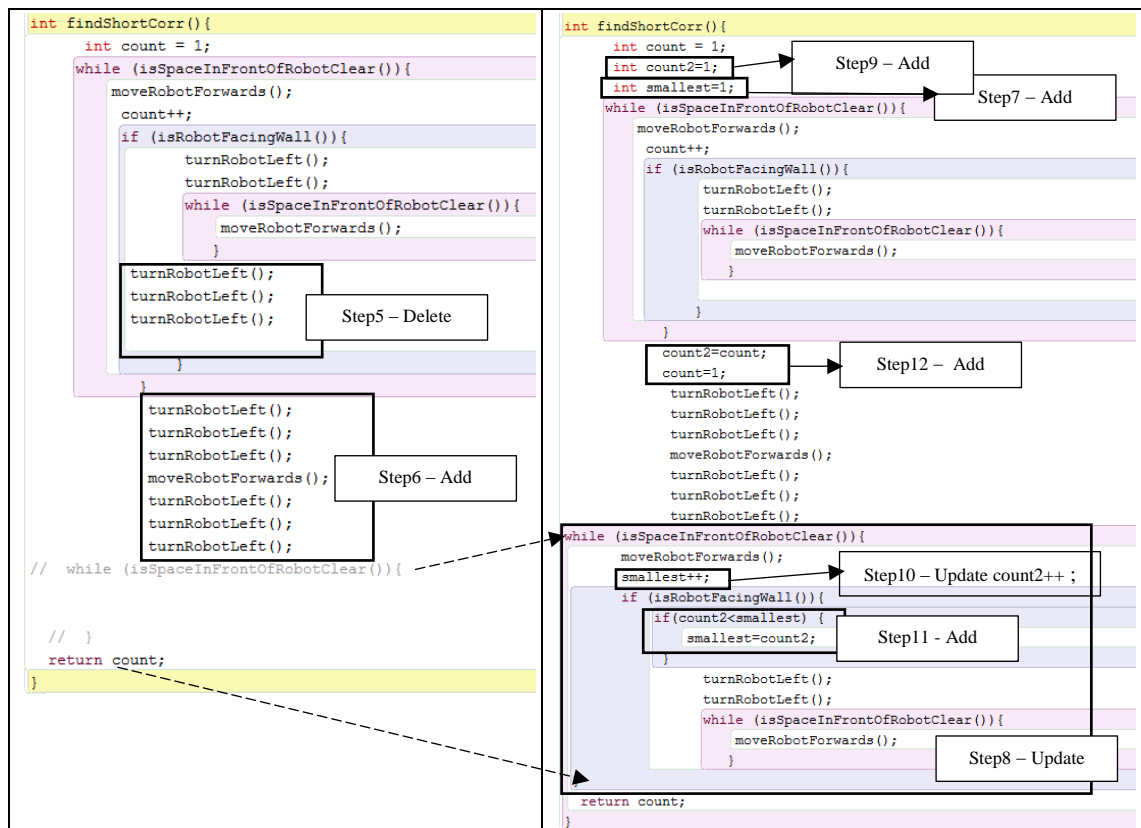


Figure 6.47 Kasper's third and fourth screen images for the shortest corridor

After another pause (25 seconds), he verbalised:

"I think, I need two integers. The count one to count the first one, and compare the first one with the second one, and then store the smaller one and the smallest."

He proceeded to define the gatherer variable `smallest` and set its value to one, uncommented the WHILE-block and continued writing code inside the WHILE-block in order to count the length of the second corridor (see Figure 6.47 (right), steps7→8). After a pause (61seconds), he verbalised while updating and reading his code (see Figure 6.47(right), steps9→12):

"I think, I need another variable, count two equal one, and this should be count two plus plus not smallest. Then, if count two smallest than count one, smallest equal count two, turn left, left, return back, left, left, left, move to the next corridor. [Pause] um, so count equal count two, and set count to zero, no, count should equal one not zero"

After another pause (59 seconds), he verbalised:

"I'm confused, I have got now three variables. I need your help."

At this point Kasper gave up on the task and did not wish to continue.

2. Retrospection:

The following is part of the conversation between the interviewer and Kasper:

Interviewer: *“At the beginning of the problem solving, you said, you had solved this question before, is that right?”*

Kasper: *“Yes, it looked familiar with me”*

Interviewer: *“Did you remember, what the requirement was for the question that you solved before?”*

Kasper: *“Um, using different worlds.”*

Interviewer: *“What else?”*

Kasper: *“The number of corridors changes in this one and the other one as I remember was one, two and three”*

Interviewer: *“In both questions the worlds were changing. In the last meeting, I asked you to find the largest corridor, but today I asked you to find the shortest corridor. By the way, did you remember the question that you solved before to find the smallest stack of beepers? Did you remember the algorithm for the smallest?”*

Kasper: *“Ah, It is at the back off up of my head, but I cannot, I knew what to do.”*

Interviewer: *“Show me, how to find the smallest stack of beepers. Did you remember the algorithm or did you forget it?”*

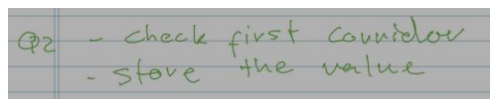
Kasper: *“I think I forget, but I remember about the corridor, I need to check the first corridor, store the value and then make a while statement that while check the other corridors and compare it with first”*

Interviewer: *“Is this plan that was in your mind?”*

Kasper: *“Yes”*

Interviewer: *“Could you write the algorithm for longest corridor?”*

He wrote two lines of the algorithm (see Figure 6.48), then he asked for help.



The image shows a handwritten note on a piece of lined paper. The text is written in green ink and consists of two lines: '- check first corridor' and '- store the value'. There is a small symbol at the beginning of the first line that looks like a circled 'Q' or '2'.

Figure 6.48 Kasper’s doodle for the longest corridor algorithm

3. Scaffolding:

When the interviewer attempted to redirect Kasper and provide assistance, he gave up on the task and was not receptive to assistance. Before Kasper left the interviewer started to use a stepwise refinement technique to help explain the code to him. Starting off with familiar pattern count the length of corridor and storing the result in the gatherer variable followed by Java commands which allowed the robot’s orientation to face west and then return back, followed by the program plan for counting the length of n-1-corridor and comparing integer numbers.

6.4.7. Largest Element in a 2D Array (Seq4 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Sequential	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Pen and paper tracing and mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	7 minutes and 20 seconds
	<i># of compilation</i>	3
	<i># of execution</i>	3
Intervention	<ol style="list-style-type: none"> 1. Clarify scaffolding – provided on request 2. Hint scaffolding – provided on request 	
Timing	Week nine of the P2 course	
Important observations with respect to prior sessions	<ul style="list-style-type: none"> • Kasper did not encounter any significant difficulties solving the smallest element in a one-dimensional array, far transfer problem (Appendix A). • Kasper, for solving the first question in this sequence, requested he use a paper that contains the syntax of a two-dimensional array (Appendix A). 	

Data

1. Think aloud:

Kasper started by declaring a method header, then after a pause (34 second), he asked for help – he needed the task requirements clarified.

2. Scaffolding:

Kasper: *“The question asked for the index or the value?”*

Interviewer: *“Your program should return the largest element in a two-dimensional array.”*

3. Think aloud:

After a pause (25 seconds), Kasper started to verbalise while writing his solution line by line in sequential order (Figure 6.49(left)):

“We take the first and store it. Int [integer] first equal array two d zero zero [arr2d [0][0]] and we compare it with the rest. We need two FOR-loops. Ah, then compare array two d x y [array2d[x][y]] greater than first. First equal array two d x y [array2d[x][y]]”

Kasper read the loop structure he had written, and reasoned about its correction:

“Fix the row and change the column, then each time compare each element with the first. If yes store the new one into the first else do nothing. Ah, I forget to return first.”


```

int findLargestElement(int[][] arr2d){
    int first = arr2d[0][0];

    for (int x = 0; x < arr2d.length; x++){
        for (int y = 0; y < arr2d.length; y++){
            if (arr2d[x][y] > first){
                first = arr2d[x][y];
            }
        }
    }
}

int findLargestElement(int[][] arr2d){
    int first = arr2d[0][0];

    for (int x = 0; x < arr2d.length; x++){
        for (int y = 0; y < arr2d.length; y++){
            System.out.println(arr2d[x][y]);
            System.out.print(" ");
            if (arr2d[x][y] > first){
                first = arr2d[x][y];
            }
        }
        System.out.println();
    }
    return first;
}

```

Figure 6.49 Kasper’s first and second screen images for the largest element in a 2D array

After the above utterance, he added a RETURN Java command. He then ran the supplied unit tests and one out of three tests failed (the test failed – `int[][] {{0, 1, 2, 3, -4}, {-2, -5, -100, 8, 9}}`):

“The problem with the last test. Um, I need your help.”

Kasper asked directly for help without even trying to read or understand the unit test messages.

4. Scaffolding:

Interviewer: “How many tests you have got?”

Kasper: “Three”

Interviewer: “How many tests passed?”

Kasper: “Two”

Interviewer: “Let us trace you code using the supplied unit tests”

The interviewer gave Kasper three two-dimensional arrays similar to the arrays in the supplied unit tests, and a trace table with three columns headed `x`, `y` and `first`. The interviewer asked Kasper to trace his code. Figure 6.50 shows what Kasper wrote in the three trace tables. Kasper could not discover his mistake.

x	y	first
0	0	0
0	1	0
0	2	0
0	3	0
0	4	0
1	0	0
1	1	0
1	2	0
1	3	0
1	4	0

x	y	first
0	0	100
0	1	100
0	2	100
0	3	100
0	4	100
1	0	100
1	1	100
1	2	100
1	3	100
1	4	100

x	y	first
0	0	-19
0	1	-19
0	2	-19
0	3	-19
0	4	-19
1	0	-19
1	1	-19
1	2	-19
1	3	-19
1	4	-19

Figure 6.50 Trace-tables for Kasper’s code for the largest element in a 2D array

The interviewer followed the same procedures used by Kasper in previous think aloud sessions (i.e. using the PRINT command) to test the correctness of his code. Therefore, Kasper was asked to add a PRINT Java command. Kasper updated his code as shown in (Figure 6.49 (right)) and verbalised:

“Not print all the element [long pause], why?”

The interviewer redirected Kasper to open his lecture notes to check the syntax of the nested FOR-loop and two-dimensional arrays. However, he was unable to fix his code. Finally, he solved the question after the interviewer intervened with syntactic help (Hint scaffold — using `for (int y = 0; y <arr2d[x].length; y++)` instead of `for (int y = 0; y <arr2d.length; y++)`).

5. Retrospection:

The following is part of the conversation between the interviewer and Kasper:

Interviewer: *“Had you seen this question before?”*

Kasper: *“Yes, in programming one. Using one-dimensional array”*

The interviewer reviewed the video tape with Kasper. The interviewer focused on how Kasper could avoid problems resulting from the lack of focus on all the unit tests messages before updating the code.

6.4.8. Column in a 2D Which Contains a Smallest Number (Seq4 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Familiar first	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	7 minutes and 31 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week nine of the P2 course	
Important observations with respect to prior sessions	Kasper needed two types of scaffolding during solving the largest element in a two-dimensional array.	

Data

1. Think aloud:

Kasper started to write his solution shown in 6.51(left). Then he verbalised:

“What I should return? [Pause] the column”

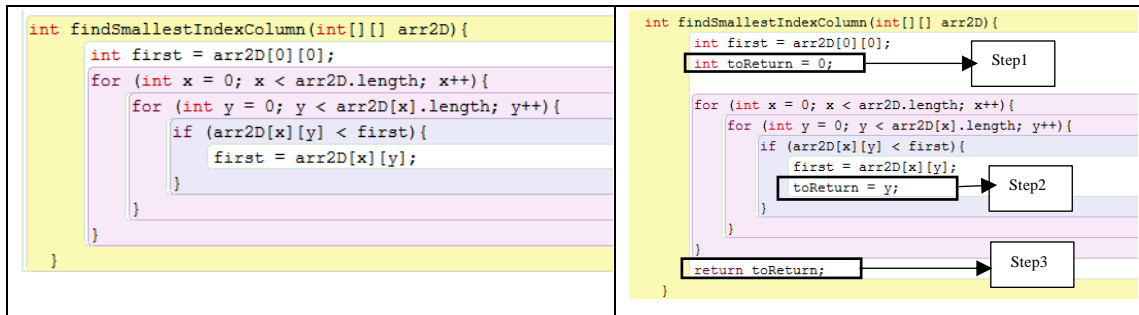


Figure 6.51 Kasper's first and second screen images for column in a 2D array which contains a smallest number

He commenced updating his code (see 6.51(right)). Finally he ran the unit tests to check his solution.

2. Retrospection:

The following is part of the conversation between the interviewer and Kasper:

Interviewer: "Had you seen this question before?"

Kasper: "No"

Interviewer: "Have you seen a question similar to it?"

Kasper: "Yes, the last one [Seq4 – Q2]"

Interviewer: "Did you remember that in the one-dimensional array task, I asked you to find the index of largest element in a one-dimensional array or had you forgotten it?"

Kasper: "I was not thinking about it. Because this question is using two- dimensional array"

Interviewer: "Did you think this question is easy or difficult?"

Kasper: "It is an easy question because of the previous question, if you give me this question first it could be difficult for me to solve it."

6.5. Matthew's Think Aloud Sessions

6.5.1. Summary

Matthew's programming ability was not great and his performance placed him in the third quartile of P1 and in the fourth quartile in P2. The think aloud data revealed that many times during the think aloud sessions for P1 and P2 he became a stopper and this seemed linked to of his fragile knowledge of basic programming commands and syntax – he was not at a level to truly understand the semantics. Matthew had difficulty mastering even the most basic aspects of programming and therefore his knowledge did not develop significantly during his time on this study. He lacked motivation and application. It was the norm for him to postpone meetings and he did not practice his programming. He failed to plan his solutions and did not make use of tools which might have scaffolded his code writing. When faced with a bug or issue in his code he almost immediately asked for help rather than try and solve the problem himself. When the researcher tried to encourage him

to make use of skills and techniques, such as tracing and debugging that were taught to him during his study, to solve an issue independently he resisted and often refused to continue.

6.5.2. Counting the Number of Beepers in a Single Corridor (Seq2 – Q1)

Encoding

Question	Not solved	
Behaviours	Stopper	
Emotion	Indiscernible	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	Read one of the supplied unit test message
	<i>Time on task</i>	14 minutes and 35 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	Exact solution – interviewer intervention	
Timing	Week six of the P1 course	
Important observations with respect to prior sessions	Matthew solved this question straight after solving Seq1 – Q1. Matthew easily solved Seq1 – Q1 despite being one of the poorer performing students.	

Data

1. Think aloud:

Matthew began by reading the problem and immediately started to verbalise and write his solution step by step as shown in Figure 6.52:

“I will apply the variable beeper, because we do not count any beeper yet, so the beeper ah will be zero. Beeper equal zero and [pause]. I will write WHILE loop, while is item on the ground. I will count the beeper at that location, ah, beeper plus plus then I will move the robot forwards”

```

void countBeeper() {
    int Beeper = 0;
    while (isItemOnGroundAtRobot()) {
        ++Beeper;
        moveRobotForwards();
    }
}

```

Figure 6.52 Matthew’s first screen image for counting for counting all beepers

Matthew recalled a plan for counting how many cells in a corridor existed with beepers and to stop when the robot encountered the first location with no beepers (this plan that was less relevant than others he had been exposed to – see Figure 6.52). After a short pause, Matthew verbalised:

“Ah but this one, ah, robot move forwards, so, ah, this one only, move robot one location, and this will stop when there is no beeper at location, ah, I will need to update my code [pause] because the robot will stop. I think, I need to change the WHILE to IF-statement.”

He then updated the WHILE-statement to an IF-statement as shown in Figure 6.53(left), step1. After a long pause, he decided to add an ELSE-block as shown in Figure 6.53(left), step2. After a short pause, he decided to add a WHILE-block before the IF-ELSE block as shown in Figure 6.53(left), step3.

```

void countBeeper() {
  int Beeper = 0;
  while (isSpaceInFrontOfRobotClear()) {
    if (isItemOnGroundAtRobot()) {
      ++Beeper;
      moveRobotForwards();
    }
    else
      moveRobotForwards();
  }
}

```

Figure 6.53 shows two stages of code development. The left panel shows the initial code with annotations: 'Step1- Update' points to the 'if' block, 'Step2 - Add' points to the 'else' block, 'Step3A - Add' points to the 'while' loop, and 'Step3B - Add' points to the closing brace. The right panel shows the code after 'Step4' (Copy/Paste) and 'Step5' (adding a print statement). The code now includes an 'if' block, an 'else' block, a 'while' loop, and a 'print' statement at the end.

Figure 6.53 Matthew's second and third screen images for counting all beepers

Matthew continued writing his solution copying and pasting the IF-ELSE block after the gatherer variable declaration as shown in Figure 6.53(right), step4. Matthew appears to have very fragile knowledge of the basic programming constructs and this is reflected in the chaotic layout of his code and lack of adherence to coding standards. He also is not familiar enough with the algorithm for counting beepers along a single corridor and this seems to impede his ability to construct a solution here.

Matthew ran the supplied unit tests but all the tests failed. Therefore, he read one of the supplied unit test message and verbalised:

“Output should be seven but was no beepers”

Matthew added a debugging PRINT statement to the end of the method (Figure 6.53(right), step5). He re-ran the supplied unit tests, one test failed, he read the test message and verbalised:

“Output should be seven but was four”

From this point that Matthew started to adopt a trial and error strategy to update his code. The interviewer offered to help Matthew after he had spent about seven minutes engaged in randomly changing his code and it seemed that he had little hope of solving the question (Figure 6.54, Matthew's final code).

```

void countBeepers() {
    int Beepers = 0;
    while (isItemOnGroundAtRobot()) {
        ++Beepers;
        moveRobotForwards();
    }
    if (isSpaceInFrontOfRobotClear())
        moveRobotForwards();
    while (isSpaceInFrontOfRobotClear()) {
        while (isItemOnGroundAtRobot()) {
            ++Beepers;
            moveRobotForwards();
        }
        if (isSpaceInFrontOfRobotClear())
            moveRobotForwards();
    }
    print("Beepers: "+Beepers);
}

```

Figure 6.54 Matthew's fourth screen image for counting all beepers

2. Scaffolding:

The interviewer redirected Matthew to write an algorithm, using smart-pen and paper, that allowed the robot to pick up all the beepers in a corridor ("General prompt" scaffolding). Matthew composed the algorithm shown in Figure 6.55. When the interviewer attempted to redirect Matthew to trace his code he gave up and asked for help. Hence, the interviewer started to use a stepwise refinement technique to explain the code to Matthew. The interviewer started by explaining the algorithm and code for picking up all the beepers from a single stack, then counting the beepers from each stack, followed by the programming code for pick up all the beepers and counting the beepers in a single corridor. The session was drawn to a conclusion

```

(while (isItemOnGroundAtRobot()) {
    pick the item with robot();
    move robot forwards();
}
while (isSpaceInFrontOfRobotClear()) {
    pick the item();
    if (isSpaceInFrontOfRobotClear())
        move forward();
}

```

Figure 6.55 Matthew's doodle for counting all beepers

6.5.3. Comparing the Length of Two Corridors (Seq1 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise refinement	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	Read the unit test message
	<i>Time on task</i>	He took 7 minutes and 17 seconds to solve the task.
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week seven of the P1 course	
Important observations with respect to prior sessions	Matthew did not encounter any significant difficulties in solving (Seq1 – Q1) in spite of the fact that he was among the bottom participants.	

Data

Think aloud:

Matthew began the problem solving by writing a Java command that defined the gatherer variable. He hesitated about what the initial value of the gatherer variable should be:

“Robot started at location zero, zero, I need to write a code to measure corridor one, the first I have to apply variable for the corridor. Because true, are are, but the first corridor not computed, yes, and the robot started from the first position”.

He then added a line by line Java commands that allows the robot to move, count the length of a single corridor, and return back without any evidence that he read or traced his code, as shown in Figure 6.56 (step1 and step2) (left). He then read the last WHILE-loop, and reasoned about the suitable Java commands to recall:

“Now the robot in location zero, zero, it is facing west, so I have to turn the robot left three times for facing the west no facing the south”.

After the above utterance, He continued to add a line by line Java commands that allowed the robot to turn north Figure 6.56 (step3) (left). He then verbalised:

“I think I need to use the same code to measure the corridor one”.

After the above utterance, he defined a second gatherer variable and copied the programming plan that counted the length of the first corridor, and then he renamed the gatherer variable to count the length of the second corridor as shown in Figure 6.56 (step1 and step2) (right). Finally, Matthew used a nested IF-ELSE block to compare the lengths of two corridors as shown in Figure 6.56 (step3) (right). Matthew ran his code and all tests passed from the first trial.

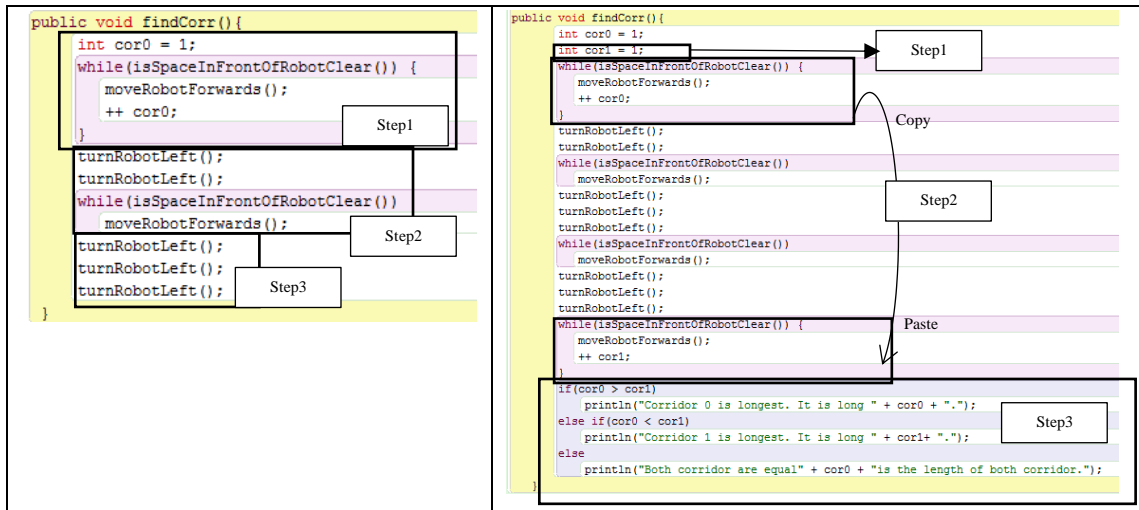


Figure 6.56 Matthew's first and second screen images for comparing the length of two corridors

6.5.4. Longest Corridor (Seq1 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing and visual debugging
	<i>Unit test</i>	Read the unit test message
	<i>Time on task</i>	15 minutes and 17 seconds
	<i># of compilation</i>	7
	<i># of execution</i>	4
Intervention	1. <i>Hint scaffolding</i> – provided on request 2. <i>“General prompt” scaffolding</i> – provided on request	
Timing	During the intra-semester break of the P1 course after week six.	
Important observations with respect to prior sessions	Matthew had solved the comparing the length of two corridors task (see Appendix A) without any difficulty.	

Data

1. Think aloud:

Matthew started by adding a method header followed by an incorrect variable declaration `int length()=0`. After writing this line of code, he verbalised while writing the subsequent WHILE-loop block:

“Because the robot start at location (0, 0) facing east, I have not to change the robot direction. First, ah, I will write a code to move robot forwards. I need to count the first also, so this should be [pause] ah one.”

After the above utterance, Matthew updated the value of the gatherer variable from zero to one (see Figure 6.57 (left), step3). Then, he wrote a closed bracket followed by a line by line block of commands (Figure 6.57 (left), step4) intended to enable the robot to face west. He did not discover that this code was incorrect until he tested his code and

visualised the robot moving. After a short pause, Matthew added a WHILE-block to return the robot to location (0, 0) as shown in Figure 6.57 (left), step5.

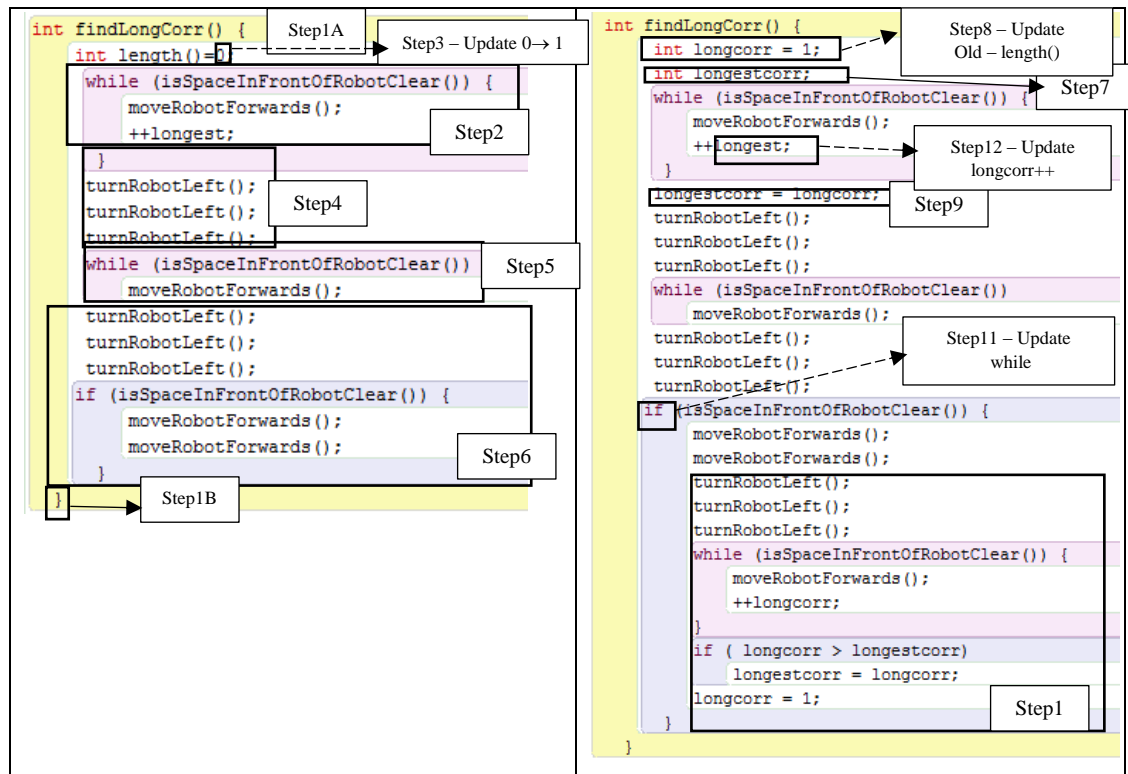


Figure 6.57 Matthew's first and second screen images for the longest corridor

Matthew re-read the last WHILE-block:

"Now the robot back to location (0, 0)"

After a long pause he verbalised while writing his code (see Figure 6.57 (left), step6):

"So I should turn the robot because it facing west now, it should be turned three times to allow the robot facing north. For ah this one, I will move the robot forwards two times"

After yet another pause, he verbalised:

"Now, um, I need to start another variable so to start measure one and to compare with the second one, so for his one"

He then defined the most wanted holder variable `longestCorr` immediately after the gather variable definition. After a short pause, he finally released he had made a mistake and that the gatherer variable name should not ended with () and he corrected the gatherer variable definition and changed its name. He then assigned the value of the gatherer variable to the most wanted holder variable as shown in see Figure 6.57 (right), steps7→9.

After a short pause, Matthew continued adding line by line Java commands as shown in Figure 6.57 (right), step10. After a long pause, Mathew verbalised:

“So which one try to use the while loop to the second corridor, is the less corridor it is possible so I think I will start the while loop at the face the robot to the ah north and the south one, I think I will need while”

After the above utterance, he changed the IF-statement to the WHILE-statement as shown in Figure 6.57 (right), step11.

Matthew started to compile his code and fix the error related to using the wrong name for the gatherer variable `longest` rather than `longcorr` (see Figure 6.57 (right), step12). His code would not compile and he failed to identify the problem – the method needed a return statement. He asked for help.

2. Scaffolding:

The interviewer redirected Matthew to read the question again

Interviewer: *“Read the question again. What should the method return?”*

Matthew: *“Return the longest corridor”*

Interviewer: *“Yep, the method should return the length of the longest corridor.”*

After a long pause, Matthew decided to add the PRINT statement as the last Java command in his code (see Figure 6.58, step13). The function of that PRINT statement was to print the value of the holder variable of course this did not fix the problem:

Interviewer: *“This method should return the longest corridor, so that means it should return an integer, do you know how write a return?”*

Matthew: *“No”*

Interviewer: *“This method should return an integer value, is that right?”*

Matthew: *“Yep”*

Interviewer: *“How did you define your method at the beginning?”*

Matthew: *“int [integer]”*

Interviewer: *“So you need to return the length of longest corridor which is an integer, can you do that?”*

Matthew: *“I do not know”*

The interviewer explained to Matthew method signatures and return types and how to write a RETURN statement using a different example (Hint scaffolding).

3. Think aloud:

Matthew ran the supplied unit tests. He discovered that there was a mistake in the robot's orientation. After updating his code as shown in Figure 6.58, step15, Matthew ran the supplied unit tests for the second time. But all tests failed, therefore he started directly updating his code by adding a line by line Java commands without any evidence that he had reread or traced his code as shown in Figure 6.58, step16.

```
int findLongCorr() {
    int longcorr = 1;
    int longestcorr;
    while (isSpaceInFrontOfRobotClear()) {
        moveRobotForwards();
        ++longcorr;
    }
    longestcorr = longcorr;
    longcorr = 1;
    turnRobotLeft();
    turnRobotLeft();
    turnRobotLeft();
    while (isSpaceInFrontOfRobotClear())
        moveRobotForwards();
    turnRobotLeft();
    turnRobotLeft();
    turnRobotLeft();
    while (isSpaceInFrontOfRobotClear()) {
        moveRobotForwards();
        moveRobotForwards();
        turnRobotLeft();
        turnRobotLeft();
        turnRobotLeft();
        while (isSpaceInFrontOfRobotClear()) {
            moveRobotForwards();
            ++longcorr;
        }
    }
    if ( longcorr > longestcorr)
        longestcorr = longcorr;
    longcorr = 1;
    turnRobotLeft();
    turnRobotLeft();
    while (isSpaceInFrontOfRobotClear())
        moveRobotForwards();
    turnRobotLeft();
    turnRobotLeft();
    turnRobotLeft();
    println("Longest corridor: "+ longestcorr);
    return (longestcorr);
}
```

The code is annotated with several boxes and arrows:

- Step 17:** Points to the line `longestcorr = longcorr;`.
- Step 14 - Delete:** Points to the line `turnRobotLeft();` (the third one in the first loop).
- Step 16:** Points to the line `turnRobotLeft();` (the first one in the second loop).
- Step 13:** Points to the line `println("Longest corridor: "+ longestcorr);`.
- Step 14:** Points to the line `return (longestcorr);`.

Figure 6.58 Matthew's third screen image for the longest corridor

When Matthew ran the supplied unit tests for the third time, two out of three tests failed, and therefore he started to read the unit test messages and reasoned about correction:

"Expected ten but was sixteen. Expected nine but was fourteen."

After the above utterance, Matthew asked directly for help.

4. Scaffolding:

The interviewer redirected Matthew to count the length of the first and second corridor for the second and third scenarios ("General prompt" scaffolding):

"Then length of the first corridor is ten while the second is seven. The length for the first is six and the length for nine. Oh, I need to set the counter after counting the length of the first corridor"

Matthew then updated his code as shown in Figure 6.58, step17.

5. Retrospection:

In the retrospective interview, Matthew confirmed that he had neither solved similar questions, nor had he solved homework assignments related to using methods and writing with a return value. During the retrospective interview; the interviewer focused on the importance of reading and interpreting unit test messages.

6.5.5. Smallest Stack of Beepers (Seq2 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Pen and paper tracing
	<i>Unit test</i>	
	<i>Time on task</i>	16 minutes and 15 seconds
	<i># of compilation</i>	
	<i># of execution</i>	
Intervention	1. “General prompt” scaffolding – provided on request 2. “General prompt” scaffolding – provided on request	
Timing	During the intra-semester break of the P1 course after week six.	
Important observations with respect to prior sessions	When solving the longest corridor task Matthew could not write code to return a value in a method.	

Data

1. Think aloud:

Matthew began by reading the problem and immediately started to verbalise and write his solution as shown in Figure 6.59 (left), step1(A &B):

“Int [integer] find smallest, first I will count, ah. I will initialise variable, so integer count equal zero, integer smallest count [smallestCount] equal to zero [pause]. I think I just need smallest count [smallestCount]. First I need to count the first stack of beeper. Now I will use WHILE. While pick up beeper and increment. Ah, I need to count the other stacks. [Pause] then I will store the variable count to the smaller count. Ah, [pause] now I will use WHILE. Move robot forwards.”

He then copy-pasted the upper While-block as shown in Figure 6.59 (left), step2. After that Matthew verbalised continued writing his code line by line Figure 6.59 (left), step3:

“Then I need to use if for this one, if ah [pause] count [pause] smaller than smallest Count [smallestCount]. So at the end we will return the smaller count.”

Matthew ran the supplied unit tests, two of three tests failed. Matthew asked directly for help without attempting to solve the problem himself he actually didn’t even go as far as reading the unit tests output.

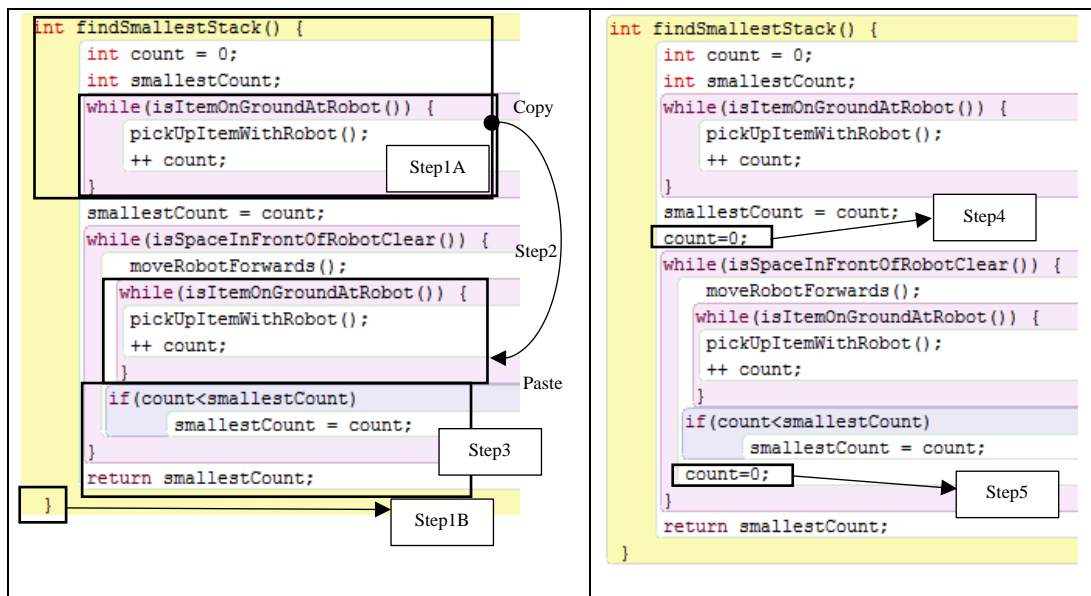


Figure 6.59 Matthew's first and second screen images for the smallest stack of beepers

2. Scaffolding:

The interviewer asked Matthew to desk check his code, giving him a corridor of length three, and each corridor had different stack of beepers. The interviewer used the data in the unit test scenario as an example for tracing (the first three piles from the first robot scenario). Mathew traced to work out how his code should work (see Figure 6.60).

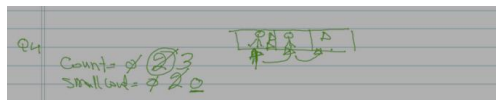


Figure 6.60 Matthew's first doodle to trace the small stack of beepers algorithm

3. Think aloud:

"I think, I need to set count to zero"

Matthew updated his code as shown in Figure 6.59 (right), step4. Mathew re-ran the supplied unit tests and two out of three supplied unit tests failed for the second time. Matthew asked directly for help for the second time even without trying to read the supplied unit test messages.

4. Scaffolding:

The interviewer redirected Matthew for the second time to trace his code using the example shown in Figure 6.60, Matthew's trace is shown in Figure 6.61.

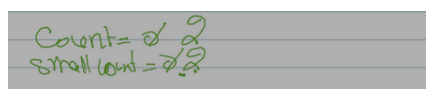


Figure 6.61 Matthew's second doodle to trace the small stack of beepers algorithm

5. Think aloud:

Matthew started directly to update his code as shown in Figure 6.59 (right), step5. Then Matthew ran the supplied unit tests and all the tests passed.

6.5.6. Shortest Corridor (Seq1 – Q4)

Encoding

Question	Not solved	
Behaviours	Stopper	
Emotion	Indiscernible	
Strategies	Trial and error	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	18 minutes and 14 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	0
Intervention	Exact solution - provided on request	
Timing	Week nine of the P1 course	
Important observations with respect to prior sessions	Matthew could not solve the longest corridor problem and required intervention and teaching related to returning values from a methods. Consequently, in the same meeting session Matthew was then able to apply what he had learnt to solve the smallest stack of beepers task.	

Data

1. Think aloud:

Matthew’s first plan involved using methods. He began by writing a method signature for counting the length of the corridor with a void return type. Matthew did not retrieve a fully formed schema for counting the length of corridor but instead appears to have retrieved sub-plans and joined those plans as shown in Figure 6.62 (left), steps1→4. The way Matthew wrote his code that indicates his fragile knowledge of methods that return values, in every case his methods had no return value.

He added commands to return the robot to its starting position and changing the robot’s orientation to face north ready to move to a second corridor, as shown in Figure 6.62 (left), step 5. After a short pause, Matthew verbalised:

“I need to use another method, change corridor”

After the above utterance, he wrote another method which moved the robot to the next corridor (Figure 6.62 (left), step6).

After a short pause, Matthew verbalised:

“So after that I will us a method to find shortest corridor, so I will use void”

He then started to write a third method which should have returned the length of the shortest corridor (Figure 6.62 (left), step7) but had a void return type and no RETURN-statement.



Figure 6.62 Matthew's first and second screen images for the shortest corridor

After a long pause, Matthew started to edit the method name `count ()` to `countSquare ()` throughout his code.

After yet another long pause, he continue updating and writing the `findShortCorr ()` method as shown in Figure 6.62 (right). As indicated by the code produced Matthew not only has difficulty with writing methods but also finds it hard to understand the differences between local and global variables.

Matthew compiled his code for the first time. He struggled to fix the errors in his code and resorted to a trial and error approach to generating his code. After 15 minutes, randomly changing his code without thinking aloud, he asked for help.

2. Scaffolding:

When the interviewer attempted to redirect Matthew and provide assistance, he gave up on the task and was not receptive to assistance. The interviewer tried to use a stepwise refinement technique to explain the code to him. The interviewer started with counting the length of corridor program, then followed with examining code for comparing two integer numbers. Programming plans that allowed the robot to move to the next corridor, and finally repeating the process of moving, counting, and comparing n-1 times were discussed.

3. Retrospection:

In the retrospective interview, Matthew confirmed that he had neither tried to practise solving the questions given to him in the think aloud sessions, nor had he undertaken the homework assignment related to using and writing methods with RETURN statements.

6.5.7. Smallest Element in a 1D Array (Seq3 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Pen and paper tracing
	<i>Unit test</i>	
	<i>Time on task</i>	5 minutes and 14 seconds
	<i># of compilation</i>	4
	<i># of execution</i>	3
Intervention	1. “General prompt” scaffolding – provided on request 2. “General prompt” scaffolding – provided on request	
Timing	Week eleven of the P1 course	
Important observations with respect to prior sessions	Matthew had difficulty solving the previous problems and had developed a pattern of depending solely on the interviewer’s assistance.	

Data

1. Scaffolding:

Matthew expressed doubt about being able to solve any question that involved one-dimensional arrays even after the interviewer had revisited the one-dimensional array lecture given to him in week eight. In response, the interviewer asked Matthew to make up any question using a one-dimensional that he thought he could answer successfully and then write the solution using smart-pen and paper. Matthew elected to solve the question that required him to print all the integers stored in a one-dimensional array (Figure 6.63).


```
int [] num = new int[] {1, 3, 4, 70}
for (int i=0; i<num.length; i++)
    println (num[i]);
```

Figure 6.63 Matthew's code to print all the elements of a 1D array

Based on his code, the interviewer then asked Matthew to try and start with this code in order to build a solution for the smallest element question.

2. Think aloud:

Matthew started by writing the method header and then the FOR-loop statement which he had written in Figure 6.63 using the stepper variable `i`. After a short pause, he decided to define a most wanted holder variable `smallestNum` before the FOR-statement and set its value to zero. Inside the FOR-block, Matthew added a line which set `smallestNum` to be the current index (the value of the loop's stepper variable `i`). After a long pause, he wrote an IF-statement which incorrectly checked if the stepper variable was less that the most wanted holder variable's value. Finally he added a RETURN statement to return the smallest number (Figure 6.64 (left)). Matthew ran the supplied unit tests, all the tests failed. He directly asked for help without trying to read any of the supplied unit test messages.

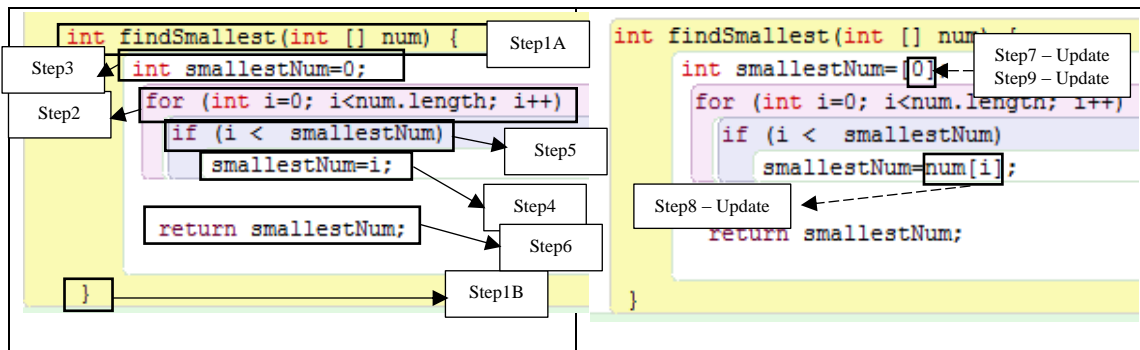


Figure 6.64 Matthew's first and second screen images for the smallest element in a 1D array

3. Scaffolding:

The interviewer gave Matthew a one-dimensional array similar to the array in the supplied unit tests that consisted of positive numbers {2, 9, 1} and a trace table with three columns headed `smallestNum`, `i`, and `num[i]`. The interviewer asked Matthew to trace through his code using this test data. Figure 6.65 shows what Matthew wrote in the trace table. Matthew was able to discover independently what his mistakes were and updated the code as shown in Figure 6.64 (right). Matthew compiled his code which generated a syntax error – illegal start of expression. Matthew corrected his code updating `int smallestNum=[0]` to `int smallestNum=num[0]`.

SmallestNum	i	run[]
0	0	2
0	1	9
0	2	1

Figure 6.65 Trace-table for Matthew's code for the smallest element in a 1D array

4. Retrospection:

The following is part of the conversation between the interviewer and Matthew:

Interviewer: "How would you normally fix the errors in your code if the unit tests have failed? Do you try to read the unit test messages or trace your code?"

Matthew: "I will call my friend"

6.6. Summary

This chapter focused on the think aloud transcriptions, the encoding and a preliminary analysis of the code writing of four participants (Andre, Luke, Kasper, and Matthew). The next chapter discusses various themes arising from the participants' verbal protocols in light of the literature on the cognitive theories and examines factors that influence novice programmers learning to program within the structure of these theories.

Chapter 7. Theory of Learning and Learning to program

7.1. Introduction

In this chapter common patterns of learning, which have been extracted from the observations of participants' programming which are detailed in Chapter 6, are explored. The aim of this chapter is to link these observations to the cognitive theories (which were discussed in Chapter 2) in a way that provides a reasonable explanation about learning to program, and about the extent to which these theories fall short as an explanation of cognitive development in the programming domain and in identifying why novice programmers are having difficulties in learning to program.

7.2. Piaget and Neo-Piagetian Theories

Piagetian and neo-Piagetian theories offer two components that could have potential for providing insight into the reasoning used by the novice computer programmers studied during this research: a stage theory and proposal regarding the way in which concepts are formed and modified.

The stage theory contains two stages that are of particular relevance to learning to program by the young adults in this study: concrete operations and formal operations. Like the knowledge domains of mathematics and the sciences, computer programming requires hypothetical reasoning which is a feature of formal operational thought. It is possible for students to engage successfully in solving arithmetic or geometric problems, that can be represented by physical objects or diagrams, by using concrete operational reasoning but formal operational thinking must be used to solve algebraic or geometric problems that demand hypothetical reasoning. Reasoning at a preoperational level cannot be used to develop even the most basic concepts in these fields such as conservation of number or classification of objects into sets.

It would be possible for computer programming students to use concrete operational reasoning to develop some of the basic concepts used to write a program, such as concepts of number, order and classification but most computer programming tasks are complex and also require formal operational thinking. Students not able to think in a formal operational way would not be able to write a program to perform these tasks except by resorting to a trial and error approach. Several students exhibited trial and error approaches to solving the problems, among these was Kasper. When Kasper tried to solve the first question in sequence one the first time he ran the program the unit tests failed. Kasper identified where he thought the problem might be and narrowed the issue down

to a couple of lines of code. He proceeded to switch the lines of code around in a random trial and error process, running the program each time he reorganised the lines of code. Eventually he managed to solve the problem but reaching the solution was a matter of luck rather than understanding.

Formal operational logic is necessary for solving the programming tasks set in courses for novice programmers. However, research has shown that people use more than one stage and move backwards and forwards through stages as the knowledge domain and the particular problems they face change. The Piagetian notions of horizontal and vertical *décalage* describe this feature but do not explain why or under what circumstances it occurs. This means that although it is highly probable that the students studied can apply formal operational logic to solve some problems in some domains they are also likely to revert to more primitive forms of reasoning at other times. The use of Piagetian stages to label the level of reasoning used by a student trying to solve a particular programming problem would provide little insight into the way in which students learn to program and so has not been attempted here.

The Piagetian notions of schema development (equilibration, assimilation and accommodation) appear to have more potential for providing insight into how students learn to program. The concepts of equilibrium and disequilibrium, and the process of equilibration describe possible mental states that cause the learning process of accommodation to occur. Thus when disequilibrium is present (i.e. when new information cannot be assimilated into existing schemas) it is assumed that there is a mental tension caused by a lack of fit between existing schemas and observed events or problems to be solved and that this is the motivation for the person to modify schemas or develop new schemas and thereby achieve equilibrium. Neither the states of disequilibrium and equilibrium nor the process of adaptation can be measured directly. However, it is possible that evidence of these could be found within the programming behaviour and think aloud responses of the participants.

There was certainly evidence that some of the participants had a great deal of difficulty when faced with some of the programming tasks provided. They could not simply expand existing schemas which had been learned through previously encountering very similar tasks and thereby reach a correct solution, i.e. a working program that correctly fulfilled the requirements of the task. In Piagetian terms they could be said to have experienced disequilibrium. Unfortunately, for the less able participants this state of disequilibrium often generated a sense of frustration and defeat. They were often overwhelmed by the

task or had no expectation that they would be able to find a resolution to the programming problem and simply appealed for help or gave up. They were unable to modify an existing schema or develop a new schema that would enable them to complete the task. They were unable to use accommodation to restore equilibrium. For example, when Luke tried to solve the longest corridor task (Seq1 – Q3) he first assumed that he needed to define three input parameters - one for each corridor — “...*I need to define three variables, one for each corridor*”. Then he identified that there was an unknown number of interconnected corridors — “*It is gonna be different because there is a different number of corridors each time, so I’m going to stick with one [one variable]*”. Therefore, he altered the way that he proposed to solve the question in response to the new information, he wrote pieces of code which solved sub-problems of moving the robot across a single corridor, counting the length of the corridor, and comparing two numbers. He had used these pieces of code to solve earlier tasks in the same sequence. But he did not combine them correctly in order to build a program that allowed a robot to move around its world and count the length of the longest corridor. Therefore, he began to generate his solution through a trial and error process. Finally, he asked for help. As another example, when Luke solved the shortest corridor (Seq1 – Q4), he wrote two methods to perform the basic robot operations, such as turning the robot right and turning the robot around. Then he wrote the main method. Firstly he merged the code for two different schemas or sub-problems, namely the programming plans for moving the robot across a single corridor, and counting the length of the corridor and storing its value into the most wanted holder variable. Directly following this code he then added code that counted the length of each remaining corridor checking using a loop. Each iteration he checked if the length of the corridor was shorter than that which was stored in the most wanted holder variable. If it was shorter, then the most wanted holder variable was set to be the current corridor’s length. To solve this question, Luke retrieved his existing schema for counting the length of the corridor but it was flawed. He had missed the subtlety of having to start the counter at 1 (to allow for the robot to sit in the first square) rather than 0. Luke was unable to fix the error in his code even when he read the unit test messages. After a long pause, Luke felt that he had reached a dead-end, and was unable to progress independently. He eventually asked for help. In the retrospection interview Luke said “*I’m still confused between counting the beepers and length of corridor*”, this suggests that he was in a state of disequilibrium.

Although disequilibrium may be a useful concept to describe the state of mind of a student prior to the adaptation process of accommodation it would seem that accommodation does not necessarily occur. These participants must find some other way of managing the state

of disequilibrium. It is quite possible that some participants have experienced failure so often that they learned to minimize the sense of disequilibrium and perhaps do not have a strong sense of expectation that they are capable of completing programming tasks or solving programming problems. For example, Andre's lack of prior knowledge led to a pattern of continual error (i.e. faulty schema for comparing integer numbers to store the largest or smallest element — making a pairwise comparison). Andre's think aloud sessions for the longest corridor (Seq1 – Q3), the shortest corridor (Seq1 – Q4), the smallest element in a one-dimensional array (Seq3 – Q2), and the largest element in a two-dimensional array (Seq4 – Q2) all contain examples of the application of this faulty schema. During these meeting sessions, the interviewer gave Andre information on the correct algorithm many times but he was unable to take advantage of this. Disequilibrium may be a necessary condition for accommodation of schemas but is not a sufficient condition.

Some participants showed evidence of accommodation to develop new schemas. Typically, the process of accommodation occurred when they were faced with larger problem which consisted of recognisable subcomponents for which they had suitable existing schema. The accommodation process consisted of restructuring and combining these sub-schemas to form a new schema. For example, Luke showed evidence of accommodation of schema when writing code to check whether or not beeper stacks were sorted in ascending order by size of the stack (Seq2 – Q4). He started by writing code that he had used to solve an earlier task for counting the number of beepers in each stack along a corridor (Seq2 – Q2). Luke then realised that he needed to know the length of the corridor in order to be able to create a one-dimensional array. Doing this would enable him to store a count of the number of beepers in each stack for each square in the corridor — *“Let me count the length. I know how to do it”* — in turn this allowed him to check whether or not the stacks were sorted. Once he had the array of stack sizes, he used his existing schema for checking if the elements in an array were sorted. Luke had already encountered the task of checking for descending (Seq3 – Q1) and ascending order in the P1 course. He added the code to check the order of the elements in the array to the end of the method. It was clear that he recalled all three schemas one by one as he worked on a solution and that he was reorganising and combining these existing cognitive schemas to solve this new problem. Also, Luke showed evidence of accommodation of schema when writing code for the counting smallest stack of beepers (Seq2 – Q3). He started by merging two pieces of code which he had used before to solve earlier problems; these code schemas were for counting the number of beepers in each stack along a single

corridor (Seq2 – Q2) and for finding the lowest number from a sequence of numbers inputted from the command line (a completed homework assignment). However, he failed to notice that during writing he had forgotten to pick up and compare the beepers at the last location. On running his code, he recognised his mistake and from there started to update his code accordingly. We conclude from this that Luke could not only retrieve the two schemas required to solve the problem but was also able to merge and tailor his schemas. In order to do this he is likely to have restructured, i.e. accommodated, his existing cognitive structures. Moreover, because he had not previously seen a problem like this he could not have already formed such a knowledge structure.

As another example of accommodation, Andre attempted to write a program to calculate the highest student mark in a collection of Student objects (Seq5 – Q1). To solve this problem, Andre started by writing a FOR-loop. The function of the FOR-loop was to iterate for all the elements (Student objects) stored in an ArrayList called “Student”. Then he verbalized “... *ID array, and with this array I can ...find the highest*”. Andre appears to have related finding the largest item in a one-dimensional array to finding the highest student mark from a list of student objects. However, he has at this stage written code to find the smallest element in a one-dimensional array, iterate over an ArrayList of objects using a FOR-loop, and add, get, set and remove elements in an ArrayList. When he started writing the code he first wrote a FOR-loop statement to iterate through elements in an ArrayList. Then he set the gather variable which was part of his schema for finding the smallest item. Then he wrote the code for checking for student with the highest mark and updating the gather variable. Based on Andres think aloud and sequence when writing his code it seems that he restructured his existing schema for finding the smallest element in an array to write this “checking step”. Andres existing mental schema for a one-dimensional array had to be modified in order to accommodate the concept of an ArrayList.

Some of the concepts used in the process of learning programming and in constructing a computer program may well be gained through assimilation to schemas previously learned in other contexts. For example, an existing schema for objects may be expanded to accommodate the idea of objects within the concept of object oriented programming. Some instances of assimilation were found during the analysis of the data collected from the participants in this study. For example, Andre showed evidence of assimilation of schemas when writing code to check whether or not the one-dimensional array elements were sorted in descending order (Seq3 – Q1). He started by verbalizing that this question

was similar to checking whether or not the elements were sorted in ascending order (homework assignment) — “ ... *if it sorted ascending, ah, if first place smaller than second one, second one is smaller than third one, ah, this time descending not ascending, the numbers are arranged from the largest to the smallest*”. After this, he started to write code that he had used to solve an earlier task (homework assignment – whether or not the one-dimensional array elements were sorted ascending) he adapted his existing schema by simply replacing the less than relational operator with the greater than operator. It was clear that he was able to match the description of the target program with the source program (existing mental schema) and make one minor change to the source structure to produce a correct solution. Kasper showed evidence of assimilation of schema when writing code for finding the column in a two-dimensional array which contained the smallest number task (Seq4 – Q3). For solving this question, Kasper first focused on writing the code that he had used to solve an earlier task for finding the largest element in a two-dimensional array (Seq4 – Q2), making only one minor change to the source structure to count the smallest element instead of the largest element. Then Kasper refined his solution by adding new Java commands to solve the programming task.

The focus of both Piagetian and neo-Piagetian theorists on defining and gathering evidence about cognitive stages seems to have resulted in a lack of research into the circumstances within domains of learning, such as computer programming, that trigger disequilibrium and into processes that could be used to achieve an optimal degree of disequilibrium for the purpose of bringing about successful schema adaptation. It has been possible to find evidence of learning through the use of the adaptation process by participants trying to solve new programming tasks but Piagetian theory has not provided a framework that could be used to design learning opportunities that would generate a level of disequilibrium most likely to generate learning. It has been left to other theorists, such as Vygotsky and Sfard, to develop ideas about how teachers can optimise learning by controlling the gap between existing schemas and those required to solve a new problem and to build bridges between a student’s schemas and those needed for a given task.

7.3. Vygotsky's Theory and the Notion of Scaffolding

7.3.1. Identifying the Zone of Proximal Development (ZPD) of Participants

The ZPD is the zone between what a learner can achieve independently and what the learner could achieve in the near future with guidance from, or collaboration with others who have more expertise in the field. Vygotsky believed that when a learner is at the ZPD for a particular task, provision of appropriate assistance by *more knowledgeable others* will give the learner enough of a boost to achieve the task and make progress. Otherwise, if a task is too difficult for a learner to achieve on their own, they cannot make progress may become frustrated and lose motivation and interest. Vygotsky believed that learners construct new knowledge within their ZPD with the help of guidance from *more knowledgeable others* and by integrating their own understandings with ideas provided by *more knowledgeable others*.

The participants were categorised according to their ability to solve the programming tasks. The three different categories were:

1. What a participant can do independently or with the assistance of software tools.
2. What a participant can do with the assistance of someone else.
3. What is beyond the participant's reach even if assisted by someone else.

In Chapter 6, the main focus was on the two top participants and the two bottom participants to capture their knowledge and the processes they adopted while problem solving. In this chapter, the results of the middle three participants have also been included in the analyses where this information could be useful for understanding the type of assistance that each participant required, and their ability to solve the programming tasks correctly.

The top participants (Andre and Luke) demonstrated their ability to move forward and learn. As a result of practising problem solving supported by scaffolding, they were able to recall and make associations based on their past experience in order to achieve a new, higher level of understanding. This became evident in the the last P2 think aloud sessions where these participants were able to solve different programming tasks with little guidance.

For example, Andre was in his sixth week of the P1 course when he solved the find the longest corridor task (Seq1 – Q3). Andre had not previously solved a question like this and he confirmed this fact in the retrospective interview. Andre solved this question correctly with hint and “general prompt” scaffolding. The scaffolding and feedback

provided enabled him to move forward and solve the smallest stack of beepers task (Seq2 – Q3, a far transfer problem – Appendix A), the shortest corridor task (Seq1 – Q4, an isomorphic problem), the smallest element in a one-dimensional array task (Seq3 – Q2, a far transfer problem), the largest element in a two-dimensional array task (Seq4 – Q2, a far transfer problem), and the highest student mark in a collection of Student object (Seq5 – Q1, a far transfer problem). However during these meeting sessions, Andre showed evidence that his faulty schema(s) still exist but later on as he practised solving multiple questions supported with scaffolding and feedback, he succeeded in applying his knowledge and skills in the different programming concepts and the task contexts.

As another example was observed during Luke's think aloud sessions for counting the number of beepers in a single corridor task (Seq2 – Q1). This question was clearly outside of his ZPD. But the model answer that was given to him helped Luke to move forwards to solve different programming tasks in the same sequence without the interviewer's assistance. For example, he solved the tasks that required him to write programs for counting the number of beepers in each stack along a single corridor task (Seq2 – Q2, Appendix A), the smallest stack of beepers task (Seq2 – Q3), and or checking whether or not the stacks were sorted task (Seq2 – Q4).

The middle participants (Chen, Isaac and Harry) were working within their ZPD and still needed scaffolding provided by a *more knowledgeable others* to solve the tasks. They also needed more practice to develop more comprehensive or more strongly related programming knowledge and skills. These middle participants were able in many cases to retrieve schemas but their retrieval was unreliable and their schemas were often flawed. In most cases they were able to solve the problems with scaffolding (see Appendix G). In contrast, the bottom participants (Matthew and Kasper) had difficulty recalling what they had previously learned. In addition, they did not practice programming outside of classes and think aloud sessions. They did not undertake the homework assignments or work on resolving programming tasks that had been sent to them after the research sessions designed to support their learning. They therefore found it difficult to solve the more advanced problems. Their learning capacity could be improved if they changed their approach to learning by actively engaging with learning tasks such as practising program syntax and solving programming tasks. These participants were not open to receiving scaffolding from the researcher or other *more knowledgeable others*. Matthew was in the third quartile of P1 and just passed the course. He solved 8 of the 19 questions in this research. He dropped to the fourth quartile in P2 and failed the course. On the other hand,

Kasper who was able to solve 11 out of 12 questions during the think aloud sessions for P1 was in the second quartile. Kasper showed evidence many times during later sessions that he had not mastered the Java commands taught during P2 and his performance dropped to the third quartile in P2.

During a think aloud session Kasper solved the largest element in a two-dimensional array task (Seq4 – Q2) with the interviewer’s assistance (clarify and hint scaffolding – syntax support). In the same meeting session, Kasper was able to transfer his knowledge from (Seq4 – Q2) to find the column in a two-dimensional array which contained the smallest number (Seq4 – Q3, an isomorphic problem). Yet, after one week he struggled to transfer his knowledge of the largest element in a two-dimensional array task to the related far transfer problem that required him to write a program for finding the highest student mark in an ArrayList of Student objects. His initial ability to transfer new understandings to another problem seemed to show that he was working within his ZPD but, perhaps because he did not hold a rich understanding of the principles underlying the ArrayList concept, he was later unable to use this knowledge to solve a more distant, far transfer problem and that problem appeared to be outside his ZPD.

Matthew was twice provided with scaffolding by the interviewer during his think aloud session for the longest corridor task (Seq1 – Q3). The first scaffolding was a hint; the interviewer explained how to define and use the method signature and the RETURN statement. The second scaffolding was a “general prompt” scaffold; the interviewer redirected Matthew to count the length of the first and second corridor for the second and third scenarios which contained more than one corridor. With assistance from this scaffolding he was able to produce a working solution. In the same meeting session, Matthew was able to apply his knowledge about the method signature and the return value to solve the smallest stack of beepers task (Seq2 – Q3, a far transfer problem). But in the next meeting session (week nine), Matthew struggled to solve the shortest corridor task (Seq1 – Q4, an isomorphic problem). At the retrospective interview, Matthew confirmed that he had neither tried to practise solving the questions given to him in the think aloud sessions, nor had he tried to solve related homework assignment related. Like Kasper, Matthew appeared to be functioning within his ZPD during the first of the sessions when he was able to transfer new understanding from one problem in order to solve the next problem. However, at a later session he also was unable to transfer the earlier learning to a new problem and in this case was unable to do so even though the new problem was closely related to the ones solved earlier. For the weaker students, time between learning

sessions and a lack of motivation to practice newly acquired programming knowledge between sessions appear to hinder progression of the ZPD.

7.3.2. Scaffolding Influence

Vygotsky believed that scaffolding plays an important role in assisting learners to a higher level of understanding. Scaffolding at the right time and of the right nature can boost learning in the present as well as in the future, which means that the “*student learns both by being taught and by self-instruction*” (Simon, 1979, p. 87).

Figure 7.1 shows the relationship between the ZPD and types of scaffolding. Within the ZPD, participants could solve a problem with the assistance provided by software tools (hard scaffolding) or with the help of *more knowledgeable others* (soft scaffolding). Outside the ZPD, participants were unable to solve the problem even when provided with scaffolding. In these cases when appropriate participants were supplied with a model answer based on stepwise refinement technique, an exact solution to study.

When providing scaffolding in this research, the interviewer used this hierarchy of scaffolding types to support the participants were necessary. In the first instance of a request for help clarification was used if appropriate otherwise a “general prompt” was given which helped participants focus their efforts. Examples of “general prompt” included suggestions to trace/desk check their code or refer to another similar problem. If that was unsuccessful the participants were given a hint as to how to solve the problem as a last resort an exact solution was required – usually at the point where the participants had given up.

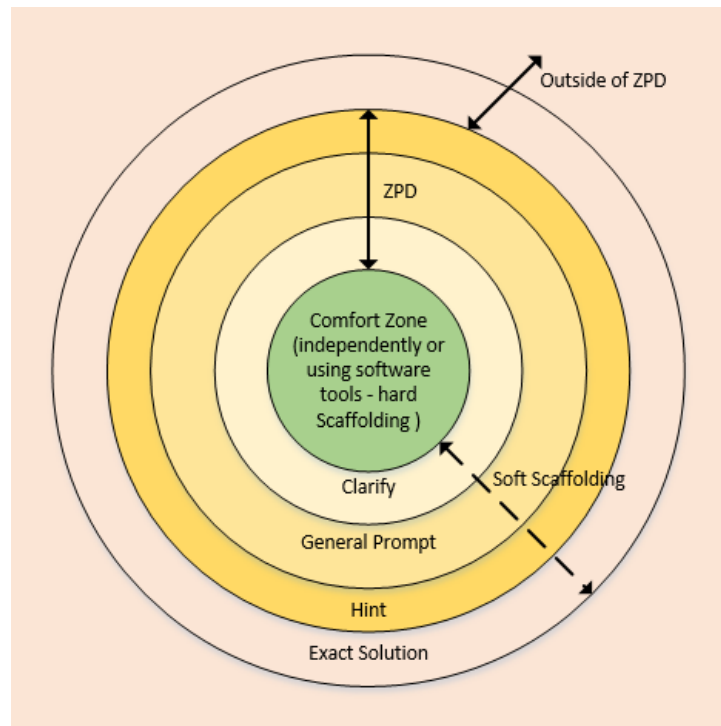


Figure 7.1 The relationship between ZPD and scaffolding

7.3.2.1. Soft Scaffolding – Assistance by the Researcher

The intervention model proposed in Chapter 3 was intended to give adequate opportunities for all the participants to successfully complete the programming task and for learning to take place.

The interviewer from time to time gave a hint without any prior “general prompt”, or provided the exact solution without a prior hint because the participant appeared to need such kind of immediate support. In addition, a participant was sometimes provided with more than one instance of scaffolding when solving a problem. Table 7.1 contains the number of clarify, “general prompt”, and hint scaffolding that led the participants to behave as movers as the tasks were within their ZPD.

Information about the think aloud session included in Chapter 6 provides evidence that Kasper’s think aloud session for the largest element in a two-dimensional array task (Seq4 – Q2) is example of clarifying scaffolding. Further examples are given in Appendix A – Andre’s think aloud session for counting the number of beepers in a single corridor (Seq2 – Q1) and Andre’s think aloud session for the smallest stack of beepers task (Seq2 – Q3).

As shown in Table 7.1, “general prompt” scaffolding was the most frequently provided form of scaffolding. The participants who were assisted were placed into the following three categories: tracing, stepwise refinement, and other. Three examples of “general prompt” that were categorised as other were: the interviewer’s prompt to Andre to think about how to implement and use variables (Seq1 – Q3), the interviewer’s prompt to Andre

to compare the number of open and closed brackets, and the interviewer’s prompt to Matthew to count the length of corridors for different Robot Word scenarios (Seq1 – Q3).

*Table 7.1 The number of soft scaffolding given during the think aloud sessions for the 133 participant solutions*¹⁰

Scaffolding type		Number of scaffolds =62
Clarify		12
General prompt	Tracing	19
	Stepwise refinement	8
	Other	3
Hint		20

Many participants required prompting before they would try to resolve an issue in their code by desk checking or tracing their program (Table 7.1). For example, Luke did not trace his code for solving the shortest corridor task (Seq1 – Q4) without prompting and nor did Matthew when trying to solve the smallest stack of beepers task (Seq2 – Q3). In these think aloud sessions the researcher prompted the participants to discover the answer by using the data in the unit test scenario as an example for tracing their code.

Table 7.1 shows that stepwise refinement scaffolding was not often required or used. The participants had to already have retrieved the appropriate schemas to solve the problem but were unable to adapt those schemas independently. Stepwise refinement scaffolding¹¹ is a type of metacognitive scaffolding which aims to help the participants to become aware of the potential connections between the new tasks and familiar previously solved tasks. This recognition of the similarities between previously solved problems and the new problem enables them to move forward. When Andre had difficulty solving the smallest element in a one-dimensional array task (Seq3 – Q2, a native Java task). He was redirected by the interviewer to solve a previously solved problem the smallest stack of beepers (Seq2 – Q3, a far transfer problem in Robot World) using pen and paper. He was able to successfully solve the Robot World problem again. This activity triggered his memory and he was then able to solve the smallest element task by using his schema for the Robot World problem.

¹⁰ See Appendix G for a summary of the type of scaffolding required by participant and code writing task.

¹¹ Metacognitive (reflective) scaffolding is defined as guiding the learner in such a way that they are encouraged to reflect on the way in which they are learning and to look inward in order to examine what learning strategies are effective for them (as detailed in Chapter 2).

Similarly, during Kasper's think aloud session for the smallest stack of beepers (Seq2 – Q3) he was redirected by the interviewer to solve a previously solved program counting the number of beepers at the first location and store the result in the most wanted holder variable. Then the interviewer asked Kasper to extend that code so that it counted the beepers at each of the remaining stacks across a single corridor (Seq2 – Q2) and compare the value of the gatherer variable with the most wanted holder variable. Kasper was able to write his solution using a computer. During Matthew's think aloud sessions for the smallest element in a one-dimensional array (Seq3 – Q2), he was redirected by the interviewer and as a consequence decided to solve a print all the elements of a one-dimensional array task in order to better understand what would be needed for the one-dimensional array problem. In all cases the participants were able to move forward to a successful solution to their particular programming problem.

Andre's think aloud session for the longest corridor (Seq1 – Q3) contains an example of hint scaffolding. The interviewer suggested that Andre create as many variables as he needed. Another type of hint scaffolding was provided when the interviewer intervened with syntax support during Kasper's think aloud sessions for the largest elements in a two-dimensional array (Seq4 – Q2). Matthew's think aloud session for the longest corridor (Seq1 – Q3) also contains an example of hint scaffolding in which the researcher made him aware of the fact that the method return type needed to be integer not void. Kasper was able to move forward to a successful solution. Andre and Mathew were also able to move forward to a successful solution after additional scaffolding was provided during the session.

Some participants exhibited stopper behavior, there were a total 31 instances recorded during the course of this research. For example, in Kasper's attempt at the shortest corridor (Seq1 – Q4) he simply gave up. This question was clearly outside of his ZPD. But the model answer given to him in the retrospective interview, later helped Kasper to arrive at a correct solution for the smallest element in a one-dimensional array (Seq3 – Q2, a far transfer problem – Appendix A). Similarly, Luke's think aloud session for the longest corridor (Seq1 – Q3) showed that this question was clearly outside of his ZPD. But the model answer given to him helped Luke to solve an isomorphic problem, the shortest corridor (Seq1 – Q4), with minimal intervention.

The researcher focused on engaging the participants in the discovery process but the participants were still receiving assistance from the researcher. The inclusion of the stepwise refinement category to Perkins and Martin's soft scaffolding model (1985) in

this research proved successful as a scaffolding method. It helped some participants to correctly solve the programming task. In the case of some participants they were able to learn from a model answer and to move forward onto a similar but different question. Stepwise refinement scaffolding led to a new understanding with minimal intervention.

Feedback given to participants during the retrospective interview on how to utilise tools or to avoid problems that resulted from not considering all possible robot scenarios, and/or not reading and understanding unit test messages (procedural scaffolding¹²) helped some participants to solve future programming tasks. For example, the feedback given to Luke regarding the importance of considering all the scenarios not just the simplest during a retrospection interview about counting the number of beepers in the single corridor (Seq2 – Q1) and the longest corridor (Seq1 – Q3) helped him to solve other programming tasks. Evidence of this was found during Luke’s think for the shortest corridor (Seq1 – Q4). During the course of solving this task Luke thought aloud about the various scenarios provided in the unit tests when he found a bug in his code — *“For the first test [scenario] expected five but was four. For the second one [scenario] expected seven but was six...”* — *“In this case [scenario] the same problem. Expected four but was five [this scenario], for the second one [scenario] seven but was six...”*.

7.3.2.2. Hard Scaffolding – Software Scaffolding – Robot World

Robot World has been reported to assist students in the debugging process because they can visualise the execution of the code (McIver & Conway, 1996). In this research it was also found that viewing the execution of the code – the robot moving across its world – assisted the participants in the debugging process. This was observed often one example was during Andre’s think aloud session for counting the length of the corridor (Seq1 – Q1) where he actually tapped the computer screen to count the number of squares in the Robot World window to check the output of his executed code. Even the lowest achieving participants appeared to be assisted in this way as demonstrated in Kasper’s think aloud session for comparing the length of two corridors (Seq1 – Q2). In this session, he wrote part of his solution and then wanted to ensure that his robot was facing in the correct direction for the next step in the solution sequence – *“I just need to test this [he then ran the code and paused to watch the robot moving] ... then left, left, left then up [as he watched the robot move he articulated the movement he was seeing and also waved his hand in the air in the direction the robot was turning]”*.

¹² Procedural scaffolding: - Redirecting learners to use resources and tools (as detailed in Chapter 2).

7.3.2.3. Hard Scaffolding – Software Scaffolding – Unit Test

Unit tests have been found to provide students with immediate feedback which allows them to independently test whether or not their program has provided a working solution to a question (Whalley & Philpott, 2011; Cardell-Oliver, 1995).

Some participants were able to correctly interpret the unit test messages. They found the unit test and the generated error messages easy to understand. For example, in Luke's think aloud session for the largest element in a two-dimensional array (Seq4 – Q2), when he ran his code a unit test failed. He quickly interpreted the error message and fixed the issue and verbalised this process — *“Expected -1 but was 0, and that is because of the initialisation of the current [variable], but I did not think about the negative number so I should set it to ah [long pause] ... to value of minimum is [long pause] the value of the first one [the first number in the array]”*. The two top participants, Luke and Andre, were able to learn how to read and understand the unit test cases on their own and update their code accordingly. For example Andre encountered one of three tests failed when answering Seq4 – Q2 therefore he started to read the code for the test failed — *“Expected -1 but was 0, if it minus one let me see, so the largest one should be the largest, so I just need to modify the code because there is a minus number there -19, -1,-2,-9. Let me set the largest number to a1 zero zero so it will be the first”* Similarly, Luke showed this ability during his think aloud session for the index of the largest element in a one-dimensional array (Seq3 – Q3). He was able to fix the error in his code after viewed the unit test file — *“For the first test result three correct [pause], the largest number in index three, ah I'm checking against the initial element...”*

The unit tests helped some participants to realise that their code was not running correctly, but it appears that they had difficulty correcting their solution. Luke's think aloud sessions for counting the number of beepers in a single corridor (Seq2 – Q1), and the shortest corridor (Seq1 – Q4) both contain instances where he ran the unit tests, read and verbalized the message but did not demonstrate that he understood the implications of the message and he was unable to correct the errors. In such cases, clearly the unit tests did not provide a sufficient scaffold for the participants.

It was apparent some participants took little care to read and understand the unit test messages. Mathew's think aloud sessions, for example see (Seq2 – Q3) and (Seq3 – Q2), provide evidence that Matthew learned that it was easier for him to seek feedback from the interviewer than to do his own code testing and correction. When this behavior emerged, the interviewer began by providing the participants with clues using the data in

the unit test in order to move the participants towards a more independent approach to problem solving in the future. The researcher focused on engaging Matthew in the discovery process so he could begin to internalize the new information and learn from his mistakes. While some participants then started using unit test before asking in Matthew's case he rarely used the unit test without prompting he just wanted to be given the answer. When the interviewer asked about why he was reluctant to use unit tests he replied that he never used them even if he was stuck on something at home he contacted a friend for help. This means that rather than spending time trying to learn independently he relied on others for an answer. Matthew failed the course.

7.3.2.4. Metacognitive Scaffolding

The instrument designed in this thesis helped the researcher to assess each participant individually by identifying their ZPD. Secondly, it encouraged participants to think of new ways to apply previously learned concepts (i.e. increasing metacognition). To do so effectively, the tasks needed to take principles from previous tasks and embed them in new scenarios (i.e. promoting schema abstraction). Some participants succeeded in building connections between the new problem and previously solved problems (i.e. increasing metacognition) by developing categories for sorting problems that had an identical schema (i.e. promoting schema abstraction). At the end of this study, participants were questioned about the usefulness of their participation in the study to themselves. Some responses indicated the development of metacognitive reasoning.

Andre: *“Yes, actually it helped me to sort my understanding to some questions and knowledge. I started to realise there are relations between questions”*.

Kasper: *“Yes, it's a good practise for me. It help me to understand my own thinking”*.

7.4. Sfard's Theory

Sfard based her theory on a dichotomy, identified by Piaget: figurative or *structural* conception in which states are viewed as momentary and static, and *operative* thinking which deals with transformations. Sfard believed that in order *“to speak about mathematical objects”* you must deal with products of a process without being concerned about the process itself (Sfard 1991, p.10). This means that when concepts are formed, operational conceptions (*process*) must precede structural conceptions (*object*).

Sfard identified three hierarchical stages (phases) of mathematics concept development: *interiorization*, *condensation* and *reification*. The transition from *process* to *object* understanding, via these stages, takes place on a concept-by-concept basis and once a

concept is reified to an abstract object it can then be used as a primitive in the acquisition of a higher level concept. It has been argued that computer science is different to mathematics and that in mathematics concepts tend to be discrete and have a neat hierarchical progression but this is not necessarily true in computer programming (Colburn and Shute, 2007). Lister et al. (2009, p. 161) argue that, in computing, modular concepts are encapsulated at the next level through information hiding and that this therefore means that *interiorization* is not a distinct step but is blended in some way with the *condensation*. Sfard herself also noted that the theory of concept development and object construction is not as linear and hierarchical as her theory might suggest, “*Even so, in the light of both theoretical arguments and experimental findings, our model does seem to present a prevailing tendency. In fact, this tendency may be so strong, that even if a new concept is introduced structurally, the student would initially interpret the definition in an operational way*” (Sfard, 1991, p.23).

The phases of cognitive development, described by Sfard, cannot be measured directly — “*We must be aware of the methodological difficulty stemming from the fact that we are dealing with a student’s implicit beliefs about the nature of ... objects. Unable to investigate the problem in a direct way, how shall we diagnose the different stages in the conceptual development of a learner? It seems that we have no choice but to describe each phase in the formation of abstract objects in terms of such ‘external’ characteristics as student’s behaviours, attitudes, and skills.*” (Sfard, 1991, p.18). Thus, it is possible that evidence of these stages could be found within the programming behaviour and think aloud responses of the participants in this study.

At the *interiorization* phase learners start to become familiar with the processes which will in time lead to a new concept being developed. *Interiorization* is the stage in which a set of actions leads to the modification of existing *objects*. This phase is considered to be a gradual and quantitative learning process. The majority of examples of *interiorization* found in the literature are of situations in which a learner becomes familiar with applying processes to data or concrete objects. Sfard gives the example of counting as a process which eventually leads to the development of the concept of natural numbers.

At the *condensation* phase learners “squeeze” sequences of operations into a single entity. They are more able to think of the process as a whole rather than a series of steps. At this stage students are able to combine processes, and make comparisons. “*At this [condensation] stage a person becomes more and more capable of thinking about a given process as a whole...*” (Sfard, 1991, p.19) and “*progress in condensation manifests itself*

as a growing ability to alternate between different representations of a concept. ” (Sfard, 1991, p.19). This phase continues while the newly developed entity is still linked closely with a particular process. Sfard suggested that condensation of the concept of negative numbers may be assessed by combining the underlying process of subtraction with other computational operations (e.g. addition).

A concept has been *reified* when that concept is seen as a fully-grown object. Sfard noted that reification, unlike the *interiorization* and condensation phases, tends to represent a leap in understanding rather than a gradual evolution. In reification a new object is separated from the process that created it and it becomes a static abstract structure in an ontological shift. In Sfard’s negative number example *reification* is achieved when a student can treat a negative number as a subset of the ring of integers without fully understanding the formal definition of a ring.

The following quotes from Sfard’s writing help to illuminate the notion of *reification* and were found to be useful by the researcher when attempting to analyse a participant’s code writing approaches:

“being able to recognize the same concept under many different disguises may be regarded as one of the important characteristics of thinking in terms of abstract objects.” (Sfard, 1992, p.76).

“Various representations of the concept become semantically unified by this abstract, purely imaginary construct. The new entity is soon detached from the process which produced it and begins to draw its meaning from the fact of its being a member of a certain category” (Sfard, 1991, p.20).

One paper in computer science education describes the development of the concept of a variable using Sfard’s stages (Wille, 2010). In this case the students were writing programs in LOGO in a Robot World. Each robot needs matchboxes (which are its “memory”) on which letters for the names of variables such as “a” and “b” are written. These matchboxes served as pre-set reifications of the idea of variable. Wille (2010, P.663) defined Sfard’s three phases for the variable concept development as follows:

- *Interiorization*: the student can handle the program: processing the program, filling matchboxes with matches, etc.
- *Condensation*: the student deals with variables as with objects but does not see them as objects, the input and output is more important than the process itself.
- *Reification*: variables are seen as independent objects.

While these categorisations relate to programming rather than mathematics and are useful when examining the development of the concept of variables they were conceived specifically in the context of the research and the LOGO Robot World. It was found that they were not so useful when interpreting data in this study. The descriptions present a simplistic view of Sfard's original phases and some of the useful ideas are lost in the new definitions. For this reason, in this analysis it was attempted to map the think aloud data to the Sfard's original definitions and theory.

In week four of the P1 course Andre attempted to solve counting the length of a single corridor task (Seq1 – Q1). He approached the code writing task line by line and did not compile his code until he had completed the method body. When he ran the program there was a compiler error – he had forgotten the closing bracket of the WHILE- loop. He was unable to make use of the processes necessary to solve the problem and could not interpret the compiler's error message. He appeared to be unable to deal with the schema for counting corridors as a *single object*. He instead resorted to a familiar cognitive schema (counting the number of beepers in a single stack) and attempted to modify that schema. In writing the code, he appears to be starting to *condense* the concepts of iteration with WHILE loops and counting beepers but because he was unable to fix the minor bug independently it seems that he is still dealing with the code as a sequence of steps or operations rather than as a single entity. Thus, Andre's code writing process and think aloud suggests that he is still *interiorising* many of the concepts required to solve the problem. It should be noted that while some concepts are still being interiorised and condensed other concepts have become reified even at this early stage of learning to program. In writing the code, Andre had clearly reified the idea of variables and was able to view and use them as a single *object*. He was able to use variables in various contexts (gatherer, most wanted etc.) without focusing on the specific syntax of writing a variable – such variables were treated as *objects*. While Andre showed further evidence of condensation and reification he consistently had faulty reified objects. It could be argued that although he appeared to be condensing and reifying concepts, because the concepts were faulty he was actually still at the interiorization phase. The following discussion focuses on one participant, Luke, and his progression in learning through Sfard's stages illustrating the cyclical nature of development of concept development. Luke was selected because he was able to move through the cycle of phases without faulty concepts being developed.

Luke was half way through his first semester of programming when he attempted to solve the smallest stack of beepers task (Seq2 – Q3). He wrote the code line by line and did not compile his code until he had finished. Before compiling his code, Luke set the most wanted holder variable to hundred. On running his code, Luke noticed that he had forgotten to pick up and compare the beepers at the last location and he was able update his code accordingly. In the retrospection interview, Luke confirmed that the reason he had set the variable to 100 initially was because he had recalled solving the highest mark problem and recognised the link between *finding the smallest stack* and *finding the lowest (minimum) mark* where the *lowest mark* variable was set to 100 as this was the highest possible mark.

Interviewer: “So you remembered the ... [finding minimum mark] ... plan?”

Luke: “Yep, and counting beepers”

His code writing process and think aloud suggests he had clearly condensed the idea of counting the beepers and minimum students’ mark, and had tried to merge them. Luke did not appear to have had any difficulties in making a connection between the problems he had previously solved and this question. It is clear that Luke had not reified finding the smallest number (minimum mark) because he had not at this stage generated an abstract schema which could be used as input to a new process reliably. In addition, he had not yet formed a generalised single object that he could apply to new situations; it appears that while he was able to recognise the same concept under a different guise he was not yet able to use that object. Even when applying the counting of beepers plan he missed the final location – suggesting that this concept was also not a reified object. Because he was able to recognise each plan in the guise of a new problem it seems that these concepts of counting and finding the lowest number are both at the *condensation* stage. Luke was able to use the concepts of simple iteration¹³ (WHILE-loop), variables and selection as single abstract *reified* entities suggesting that these concepts were now abstract objects independent of process, which could be used as input to a new process to develop a new concept.

Five weeks after solving the smallest stack of beepers problem, Luke attempted to write code to find the smallest element in a one-dimensional array task (Seq3 – Q2, a far transfer problem). He approached the code writing task line by line. Luke hesitated many times

¹³ Simple iteration here is defined as iteration in order, one by one through a sequence of steps or a list of items, without nested selection or iteration.

about what the initial value of the most wanted holder variable should be. He eventually settled on the first element of the array as his starting value — *“Integer smallest equals ... [long pause] ... um zero ... [long pause] ... smallest equal int array of zero [intArray[0]]”*.

He continued writing the FOR-loop, followed by the IF-statement which checked the current array element against the most wanted holder variable and updated the most wanted variable if necessary. A long pause was recorded in the think aloud before Luke added the correct relational operator (<) to the IF-statement which updates the most wanted holder variable. When the interviewer asked him, in the retrospection interview, what he was thinking at that point he said he was thinking about which operator he needed to use.

Luke was observed to go over the line that defined and initialised the most wanted holder variable again — *“I need to change this value, let me try it”*. He updated this line as follows:

1. `int smallest = intArray[0];` became `int smallest; ”`
2. Compile
3. `int smallest= intArray[1];` after that without hesitation, he changed one to zero

At the retrospective interview, Luke confirmed to the interviewer that solving the smallest stack of beepers problem helped him to solve this question. In solving this task Luke again, as in Seq2 – Q3, called on his cognitive schema for finding the smallest number (minimum mark). In this case he was able to solve the problem independently. He only encountered one problem on compiling his code. He found he had missed the RETURN statement for the method and was able to fix this immediately on his own. It was unclear at this stage whether or not Luke had reified the concepts related to a one-dimensional array such as indexing, iteration and searching. However, it became evident in the same session solving the next task (Seq3 – Q3) that he had reified the concept of searching for an element in a one-dimensional array.

The task Seq3 – Q3 is a problem that is isomorphic to the previous question (Seq3 – Q2). The task asks the participant to find the index of the largest element in a one-dimensional array. There were two new aspects to this problem: adapting the logic of existing schemas or plans from the familiar find smallest element (or number) to finding the largest and moving from the familiar find element in an array to finding the index of the element.

During the writing process, Luke seemed to be trying to update his existing schema for smallest element in a one-dimensional array (Seq3 – Q2) by adding new information. As Luke wrote his code, he forgot to update the value of the most wanted holder variable. On running his code and reading the code for the test file, Luke said “*For the first test result three correct [pause], the largest number in index three, ah I’m checking against the initial element let me see*”. The first test has an array which includes positive and negative numbers and stores the largest element (the highest number) at index three. He realised that he was always checking against the first element in the array. Luke updated his code correctly. In order to solve this problem Luke used two gatherer variables one for the current element in the array and one for the index of the current element in the array. The use of this extra variable is redundant but interestingly all participants in the study took this approach. It seems that Luke is at the *interiorization* phase for finding the index of the largest element in the array. On the other hand, Luke’s responses to this question showed that he had become increasingly capable of thinking about finding the smallest or the largest element in a one-dimensional array as a *condensed* entity without needing to distinguish between the largest and the smallest element. Unlike in Seq3 – Q2, Luke did not need to take time to think about which operator to use and wrote the selection statement without hesitation. It was not clear at this stage whether or not the concept of finding elements in a one-dimensional array had been *reified* but Luke was able to use this concept in different guises.

After a further four months (a time which included the three-week inter-semester break), Luke requested an opportunity to solve the largest element in a two-dimensional array task (Seq4 – Q2, a far transfer problem). During the solving of this task it was clear that Luke was still at the *interiorization* stage of concept development for finding elements in a two-dimensional array. He relied on existing concepts, iterating over a two-dimensional array and comparing two values. This posed a problem because while the concept of comparing two values was reified, he had not yet reified the concept of two-dimensional array iteration. Although in principle he should have developed a schema for iteration of two-dimensional arrays as a result of his P2 course work this schema was clearly not well formed and Luke was aware of this — “*I’ve still got a problem with nested loops, I need to practise more and more*”. When Luke ran his code one of three supplied unit tests failed — “*Expected -1 but was 0, and that’s because the initialisation of the current [variable], but I did not think about the negative number*”. He was clearly linking this problem with his understanding of two-dimensional array. It was not until after he had

run the code that he realised there might also be a link with finding elements in a one-dimensional array:

Interviewer: *“Have you seen this question before?”*

Luke: *“Yep, I think in a one-dimensional array, I think the question was either smallest or largest element”*

In the same meeting session, Luke solved an isomorphic problem which asked him to write code to find the column in a two-dimensional array which contained the smallest number (Seq4 – Q3). When solving this question Luke appeared to have no difficulty in making a connection between the previous question and this question. There is no evidence that he tried to read or trace his code and all tests passed on the first attempt. He started with the most holder variable definition and initialisation, followed by the schema for iterating over a two-dimensional array then — *“... another int [integer] variable because I need [pause] the index of smallest column [pause] this question is different than the first one [Seq4 – Q2]”*, followed by the rest of the Java commands. In the other words, Luke’s responses to this question showed that he became more and more capable of thinking about the smallest and the largest element of a two-dimensional array as a *condensed* entity thus the concepts first encountered in Seq4 – Q2 appear are beginning to be condensed. In writing this problem; Luke seems to have *reified* the concept of a two-dimensional array and was able to view and use them as a single object. Also, the concept of smallest/largest element. This was confirmed in the next meeting session one week later when Luke attempted to write a code to calculate the highest mark for each student in an ArrayList of student objects (Seq5 – Q1). The marks for each student are stored as an instance variable which is a one-dimensional array.

Luke first defined the most holder variable — *“I’m going to add integer highest equals, no this not ID”* he deleted the line declaring the most wanted holder variable. He then continued to add his code line by line. Luke was obviously recalling his schema for finding an element in an array which was formed for a one-dimensional array in order to solve this problem. He then went on to code the solution rapidly and on the first compile and run it passed all of the tests. Luke appeared to be able to deal with the schema for finding the smallest or largest element as a single *reified* object, and was able to use this familiar object to develop a new concept. In solving this question he also needed to be able to use ArrayLists of objects so it is highly likely, given his ease of use of ArrayList (iteration and access of objects), that he had *reified* the idea of an ArrayList of objects as

a single object. In a subsequent session Luke again demonstrated his ability to use ArrayLists as a reified object (see data for Seq5 – Q2, and Seq5 – Q3 in Appendix A).

In learning programming all of the participants in this study tried at various points to build reified concept objects – some more successfully than others. Some participants such as Andre were able to reify the very early concepts taught in P1 (e.g., variables and simple selection) but were subsequently unable to move beyond the interiorization phase for more advanced concepts. Andre was able to solve the questions in this study and write code that passed the tests but his schema's were flawed – this might suggest that it is possible to use concepts that are not reified to produce a working solution but that solution is not necessarily “well written”. Other students such as Luke were able to move forward developing concepts. It was shown that a gradual development of concepts occurred from *interiorization* to *condensation* to *reification*. Each object (a generalised abstract concept) was then used to develop more advanced concepts in order to solve more difficult or complex code writing problems. It was clear to the researcher that students in their first year of programming are exposed to many new concepts all of which build on each other in a very short period of time (24 weeks of teaching).

According to Sfard concept development and in particular the phases of interiorization and condensation take time. Reification involves a leap in understanding which, beyond the simple initial concepts which are first taught in an imperative first pedagogy, for most novice programmers seems on the basis of this study to take more than a year. Most of the participants, as the tasks became progressively more difficult, operated and in some cases floundered about at the interiorization phase and sometimes at the condensation phase. The questions as designed in this study should have supported concept development far more than the questions which the students encountered in the courses themselves. This is because the tasks were deliberately designed in order for concepts to build on concepts or expand a concept. It is obvious that being cognisant of Sfard's concept development phases and the cyclical nature of concept development should ensure that educators are able to provide better code examples and code writing tasks which should assist learning. Additionally, being aware of the considerable time that concept development takes might result in a more realistic and achievable curriculum.

7.5. Cognitive Load Theory

CLT provides a model which can be used to examine the load on working memory in three dimensions - intrinsic, extraneous and germane cognitive load.

Intrinsic load depends on the internal difficulty of the learning task, moderated by the level of learner expertise. Extraneous cognitive load is produced by the learning context and the way in which instructional content is presented to the learner. In contrast, germane cognitive load is the degree of mental effort that is applied to schema acquisition, i.e. to schema construction and automation. The learning activities generally consist of comparing and contrasting between existing mental schemas and newly presented information, in conjunction with some form of practice in order to initiate schema development. The working memory available for learning could be overloaded if the combination of intrinsic and extraneous cognitive load is too high and there is therefore insufficient learning memory available for the learner to be able to meet the germane cognitive load required to modify or construct a schema or to solve a problem.

When a novice first has to write a computer program, the intrinsic load is high and that form of load will only get lower when the person has learned the language and the programming constructs and has had practice at programming. However, inefficient instructional designs can add unnecessary extraneous cognitive load and therefore interfere with learning by overloading the working memory. Recommended methods for reducing the load include segmenting the programming task into simpler sub-programming tasks and providing opportunities for practising relevant components of knowledge. This can increase the availability of working memory resources for processing interacting elements and constructing or modifying schema that are necessary for accomplishing learning goals. In this study, the programming tasks used were designed to encourage participants to think of new ways to apply previously learned concepts. To do so effectively, the tasks needed to take principles from previous tasks and embed them in new scenarios using the same or different programming concepts and task contexts. The extraneous component of the cognitive load was reduced by sequencing, ordering and organising the learning required to solve the programming tasks presented to the participants.

During think aloud sessions some participants showed evidence that the use of simple-to-complex sequencing, ordering and organising questions presented was a useful approach to reducing extraneous cognitive load and thereby increasing the working memory available for the germane load (schema acquisition and construction). An example of this was observed during Luke's think aloud session for solving the largest element in a two-dimensional array task (Seq4 – Q2). Luke was questioned during the retrospective interview as to whether or not he had solved the same question before, and he said he had

not done so. For solving this question, it seemed that he was trying to temporarily reduce the load by dividing the unknown complex problem into known sub-problems. The order in which he coded his solution suggested that he probably retrieved the schema for iterating the elements of a two-dimensional array, then the schema for comparing two numbers first. Later he realised that a most wanted holder variable was required. Luke could easily fix the syntax error in his code that related to checking the length of the row and the column of the two-dimensional array. On running his code and reading the unit test message — *“Expected -1 but was 0, and that because the initialisation of the current [the most wanted holder variable], but I did not think about the negative number ...”* — *“... I think in a one-dimensional array, I think the question was either the smallest or largest element”* — at this stage of problem solving, Luke started to become aware of the connection between this code and that of the previously solved question. This allowed him to activate related information in his long term memory and as a result he directed his attention to fixing the error in his code (updating the value of the most wanted holder variable). In the same meeting session, he was able to use his existing schema for solving this question and apply his knowledge and skills to find the column in a two-dimensional array which contained the smallest number (Seq4 – Q3, an isomorphic problem).

Another example was observed during Kasper’s think aloud sessions for the find the column in a two-dimensional array which contained the smallest number (Seq4 – Q3). For solving this question, he firstly focused on the familiar pattern of the problem (i.e. existing schema for solving the largest number in a two-dimensional array (Seq4 – Q2)), and then refined his solution by dealing with the parts not dealt with during the first phase of the solution. Finally, he ran the supplied unit tests. All the tests were passed from the first trial. At retrospective interview, Kasper said — *“It is an easy question because of the previous question, if you give me this question first it could be difficult for me to solve it”*. Kasper’s approach of segmenting the problem appears to have been successful in reducing the extraneous cognitive load and limiting the intrinsic load at any given time thereby increasing the working memory available for use in reasoning about ideas central to the germane cognitive load; he could easily see similarities and he was successful in solving the extension to this question.

During his think aloud session for the index of the largest element in a one-dimensional array (Seq3 – Q3), Andre started problem solving by verbalizing his plan — *“This question is similar to the first question[find the smallest element in a one-dimensional array], the first question find the smallest one, and this question find the largest one, um, should*

return the index of largest element, the point is how to find the index, is the basically the same so, the difference the first one is asked to return the smallest element and now asked as to return the index of it, basically is the same returning, the most difficult part of this question how to find the index of an array.” — “*Let me think about the largest, so first think to compare with, find out the largest, and I need to know the index of it, index of the largest element ...*” — Andre’s plan provided evidence that he was thinking in terms of retrieving his existing schema for solving the smallest element in a one-dimensional array (Seq3 – Q2) and he used it as a template for solving this question. In his planning he focused on comparing and contrasting between the source and the target problem. During the writing process, Andre seemed to be trying to add new information to his existing schema for the source code. Andre was able to write his solution in a linear order. Andre paused twice to question the value of the stepper variable, and what and how to compare it. He was able to come up with the solutions. Andre tried twice to read part of his code, compiled his code and was able to identify and fix the error (he had forgotten to add the word `int` before the stepper variable initialisation). Finally, Andre ran his code and all tests were passed from the first trial. He mastered the language syntax and semantics of the new task (target), and showed that he was aware of similarities between problems. Andre correctly solved the programming task by connecting the new problem to previously solved programs. This may have enabled him to free cognitive resources for germane activities such as reading his code and identifying and fixing errors, and solving the question correctly. Andre’s approach of segmenting the problem appears to have been successful in reducing the extraneous cognitive load and limiting the intrinsic load at any given time thereby increasing the working memory available for use in reasoning about ideas central to the germane cognitive load; he could easily see similarities and he was successful in solving the extension to this question.

During his think aloud session for the highest mark in a collection of Student object (Seq5 – Q1), Luke showed that he had been able to use information about the source programming task (smallest element in a one-dimensional array (Seq3 – Q2, a far transfer problem)) as a template for solving this question. He was observed to go over some lines of his code and update it accordingly. Luke ran his code and all tests were passed from the first trial. For solving this question, Luke’s approach indicated that he had been able to use schema developed during his efforts to solve Seq3 – Q2 and thereby to reduce the cognitive resources required. This is similar to the evidence from Andre’s think aloud session for the highest mark in a collection of Student object (Seq5 – Q1).

Additionally, in this study some participants used simple-to-complex sequencing and ordering and organising questions, in ways that could have resulted in transfer of performance from earlier tasks and of lower demands on working memory and hence a decrease in the time required for the later related questions. For example, Andre took 46 minutes and 13 seconds to solve the smallest element in a one-dimensional array (Seq3 – Q2), while he took 6 minutes and 42 seconds to solve the index of the largest element in a one-dimensional array (Seq3 – Q3, an isomorphic problem), and 9 minutes and 16 seconds to solve the largest element in a two-dimensional array (Seq4 – Q2, a far transfer problem). Also, Luke took 10 minutes and 3 seconds to solve the largest element in a two-dimensional array (Seq4 – Q2), while he took 3 minutes to solve find the column in a two-dimensional array which contained the smallest number (Seq4 – Q3, an isomorphic problem).

Excessive cognitive load automatically influences learning by causing frustration that can hinder learning activities. It is possible to argue that instances of participants spending very little time on task indicate excessive cognitive load because learners quickly stop putting effort into investigating their learning material. However, one might also argue that the time needed to solve a programming task is an indication of the extraneous the load imposed by the way in which the instructional material or problem is presented. The data analysis revealed that some participants are unable to recall the thought processes used when solving problems which took them a long time, as reported in Andre and Luke's retrospective interview for the longest corridor (Seq1 – Q3). This may have been because they floundered many times during the problem solving session, or they spent a lot of time solving the programming task. Andre took 40 minutes and 13 seconds for solving the longest corridor question (Seq1 – Q3). While Luke spent 45 minutes and five seconds trying to solve the same question before he turned into a stopper and ask for help.

CLT has been useful in explaining why some participants cannot progress or have difficulty with certain aspects of learning. Excessive load caused learner frustration even if participants were assisted in the task, and they were unable to take advantage of this experience when faced with a related task. An example of this was Andre's think aloud session for the smallest element in a one-dimensional array (Seq3 – Q2). Andre made a lot of mistakes when trying to solve this question, which may be because his lack of prior knowledge led to a pattern of continuous errors (referring to Andre's think aloud sessions for the longest corridor (Seq1 – Q3), and the shortest corridor (Seq1 – Q4)). Finally, he solved the question with the interviewer's assistance ("general prompt" scaffolding). In

the same meeting session, Andre correctly solved the question that required him to find the index of the largest element (Seq3 – Q3, an isomorphic problem). After fifteen weeks, Andre made the same mistakes when trying to solve the largest element in the two-dimensional array question (Seq4 – Q2, a far transfer) which means that his faulty schema still existed. This may be because of the length of time since he had last practised solving similar problems, but finally he succeeded in solving this question after he started to trace his code and read the unit test message, and unit test file (i.e. he directed his attention to the learning activities). Doing so allowed him to add more information to his knowledge structure to update his faulty schema(s). This became evident in the next think aloud session, when Andre was instructed to solve find the column in a two-dimensional array which contained the smallest number (Seq4 – Q3, an isomorphic problem – Appendix A) and the highest student mark in a collection of Student object question (Seq5 – Q1, a far transfer problem). Another example was Matthew's think aloud session for the longest corridor (Seq1 – Q3). He solved this question with interviewer assistance (one of these assistance was the interviewer intervened with syntax help). On checking his homework assignment on week seven it was found that Matthew had not solved any questions related to the method signature and return value. However, later on, during the same meeting session, he could easily use the method signature and return value (referring to Matthew's think aloud sessions for the smallest stack of beepers (Seq2 – Q3, a far transfer)) even though his behaviour was shown to be fragile when he attempted to solve the shortest corridor (Seq1 – Q4, an isomorphic problem) in next meeting session (after three weeks). This means that Matthew did not learn from his mistakes and this could be the result of cognitive overload. If Matthew was cognitively overloaded, then this could be a cause of the learning deficiencies. This is demonstrated by the following: Firstly, Matthew solved the longest corridor with the interviewer's assistance however, he could not learn from this question because there was insufficient working memory resources left over to develop appropriate schema in long term memory, i.e. to learn. Secondly, Matthew had not practised a homework assignment so had not done the work that could have helped him to establish appropriate schemas. This would have imposed a higher level load on the working memory.

The unavailability of relevant schema may be a hindrance to the adaptation of new knowledge because the intrinsic cognitive load involved in finding a correct solution to a program is likely to be high (i.e. participants had to simultaneously process many new elements of information in working memory). For example, Luke's think aloud sessions for counting the number of beepers in a single corridor (Seq2 – Q1) and the longest

corridor (Seq1 – Q3). Another example is in Matthew’s think aloud session for counting the number of beepers in a single corridor (Seq2 – Q1). Moreover, if the quality of the solution is not generalised, connected or integrated, the intrinsic cognitive load involved in finding a correct solution to a program is likely to be high. This evidence cannot be detected on a single snapshot of time, but by more than one meeting session. Empirical evidence for this was Andre’s think aloud session for the longest corridor (Seq1 – Q3). Andre’s solution was to set a most wanted holder variable to zero to store the current longest corridor found. Each corridor is checked sequentially until there are no more corridors. Andre’s solution was not valid for all situations. This became evident in the next meeting session, when Andre was instructed to solve the shortest corridor (Seq1 – Q4, an isomorphic problem). Another example of evidence for high intrinsic load was Kasper’s think aloud session for the longest corridor (Seq1 – Q3). For solving this question, Kasper defined three gatherer variables, one variable for each corridor, and then used three separate loops to count and finally compare the three values of the gatherer variables in order to find the longest corridor. His solution provided a directed translation of the code scenarios and would only work with the scenarios provided in the question. This became evident in the next meeting session, when Kasper was instructed to solve the shortest corridor (Seq1 – Q4, an isomorphic problem). Kasper failed to provide the required solution for this question and as a result he became a stopper.

According to CLT, when learners are novices in a domain, the cognitive load associated with unguided learning is high because novices lack any sort of guide to aid their knowledge acquisition processes. The general observation from think aloud sessions is that when participants were struggling to solve the programming task, the cognitive load imposed during the writing process was probably high, and this load could be reduced when the interviewer guided them to aid their knowledge acquisition processes. This was shown in the think aloud session for Andre, when he experienced difficulties while trying to solve the smallest element in a one-dimensional array (Seq3 – Q2). He was redirected by the interviewer to solve a previously solved problem, the smallest stack of beepers (Seq2 – Q3). In the same meeting session, Andre succeeded in applying his knowledge and skills for solving the smallest element in a one-dimensional array to solve the index of the largest element (Seq3 – Q3, an isomorphic problem). Also, in Kasper’s think aloud session for the largest element in a two-dimensional array (Seq4 – Q2), the interviewer intervened by providing Kasper with syntax support of the nested FOR-loop and two-dimensional array. It became evident in the same meeting session that Kasper had learnt from this intervention. When he was instructed to solve find the column in a two-

dimensional array which contained the smallest number (Seq4 – Q3, an isomorphic problem), Kasper had no difficulties with the syntax of nested FOR-loop or the two-dimensional array. For solving each row of a two-dimensional array elements are sorted task (Seq4 – Q4, Appendix A). Kasper had no difficulty with the syntax of the nested FOR-loop or the two-dimensional array, but he had difficulty recalling his existing schema for whether or not a one-dimensional array is sorted (Seq3 – Q1).

The issues that are explored in this study underline the relevance of using common patterns (analogies) for practicing problem solving. However, at some stages of learning some participants could not immediately draw on relevant prior knowledge; but later on as they practised solving multiple questions supported with different kinds of scaffolding some participants succeeded in building a connection between the new problem and previously solved problems (i.e. increasing metacognition) by developing categories for sorting problems that had an identical schema (i.e. promoting schema abstraction). As evidenced by the data presented in Chapter 6 often participants mentioned the connection between an earlier task and the current task. Once they had seen the value in thinking about previous tasks in order to develop a solution for the “new” task they continued to use this strategy. In another example, Andre was directed to desk check in order to solve one problem and then continued to use this approach in future when he encountered problems. It seems very likely that he had reflected on that strategy and saw it as an effective approach to debugging and problem solving. This suggests that students were able to develop metacognitive thinking, identify suitable strategies, and thus were able to solve the more difficult questions more easily than expected and that these strategies may reduce cognitive load.

This supports the notion that self-regulation and metacognition affects cognitive load. Prior knowledge affects intrinsic cognitive load, a learner’s metacognitive development and their ability to self-regulate. A participant with a high level of prior domain knowledge will be more likely to experience a lower level of mental effort.

Analogies are very powerful in CLT terms as they can foster schema activation, and therefore help schema construction. Additionally, correspondences between existing schemas and the new instance have been found to influence the way that relevant analogues interact with each other. Merging and nesting are difficult skills in themselves because they require great attention to detail and deep interaction (i.e. intrinsic and extraneous load tends to be high). All four participants could easily solve the question that required them to compare the length of two corridors (Seq1 – Q2) (see Andre and

Luke – Appendix A. Kasper, and Matthew’s think aloud sessions – Chapter 6). The solution for this question required an abutment of more than one plan together. Some of them struggled with questions which required merging and nesting programming plans (referring to Andre and Luke’s think aloud sessions for the longest corridor (Seq1 – Q3), and Kasper’s think aloud sessions for the smallest stack of beepers (Seq2 – Q3) and the shortest corridor (Seq1 – Q4)). Soloway (1986) also found that merging and nesting programming plans are difficult skills especially for novice programmers because they require great attention to detail and deep interaction.

In this study, variances of approaches in transfer strategies were found among the participants. These transfer strategies generally consisted of comparing and contrasting between existing mental schemas and newly presented information in conjunction with some form of practice in order to initiate schema development. The names of these two forms of strategies are forward-reaching transfer and backward-reaching transfer. In forward-reaching transfer, the participants initially focused on generating abstract plans and as they engaged in problem solving they considered where these abstractions might be applied. The general observation about Andre’s behaviour during the think aloud sessions is that he started to plan out his solution prior to coding. As he engaged in solving the programming tasks, he considered other situations. An example of this was observed during Andre’s think aloud session for the longest corridor (Seq1 – Q3). Andre started problem solving by formulating a plan around the difficulties in solving this question in term of sub-programs — *“I need to compare the numbers, the number of corridors changes each time it is created, so I need to find it out, whether there are more than one corridor, I need to compare the length of corridors, the first situation there are one corridor, so move the robot to the end of corridor, and count the numbers, and turn robot back, and to check whether there is wall ... the problem how to compare, ah, the problem how to memorise the long of corridor, this is the longest ah [long pause] ... how to compare, three is not enough is keep changing go to the first, go to the second corridor, [pause] but if there is more, four corridors, how to assign the integers, to assign the variables, what I can do”*. Another example of forward-reaching transfer was observed during Andre’s think aloud session for the shortest corridor (Seq1 – Q4). For solving this question, Andre showed evidence of retrieving his existing schema for the longest corridor that could be used to solve this question — *“So it is similar to last meeting, it was find out the longest, now it shortest, I think it is basically ah the same ... First as I remembered, we need to have ah, we need to have two, set up two integer variables to have comparison”*.

Backward-reaching transfer occurred when the participants were faced with a problem and abstracted key characteristics from the problem and reached back into their existing knowledge for matches. Some participants showed evidence of backward-reaching transfer by verbalizing statements about the next programming plan to apply during the writing process. An example of this was observed during Kasper's think aloud sessions for the smallest stack of beepers (Seq2 – Q3). Kasper started by writing the method signature and then he verbalized — *“I need to pick up the first [beepers at the first location]”*. After writing the code for picking up the beepers at each stack along the corridor except the last stack, he also verbalized — *“... after doing that, after picking the second one, should be compare it with the first one, so I need to define another one [Kasper was thinking back about comparing two numbers]”*. Another example of backward-reaching transfer was observed during Matthew's think aloud session for comparing the length of two corridors (Seq1 – Q2). For solving this question, he started with a gatherer variable definition — *“Robot started at location zero, zero, I need to write a code to measure corridor one, the first I have to apply variable for the corridor. Because true, are are, but the first corridor not computed, yes, and the robot started from the first position”* — after counting the length of the first corridor (corridor zero), Matthew verbalized again — *“I think I need to use the same code to measure the corridor one [the second corridor]”*.

A key objective between CLT and the learning program is to develop a useful method for acquisition of schema(s) and their use during problem solving that enables novice programmers to draw on previous experiences to facilitating learning.

7.6. Summary

The Piagetian explanation of learning through the processes of equilibration, assimilation and accommodation could be used to explain some of the learning behaviours of participants observed during think aloud sessions and their responses during retrospective interviews. The findings reported here indicate that although Piagetian theory may be used to explain learning success it is not useful for predicting whether or not learning is likely to take place, or whether disequilibrium will lead to schema construction or to a form of avoidance behaviour that minimises the impact of disequilibrium. Piaget provided a new way of thinking about learning and was a leader in the development of constructivist learning theory. However, from the perspective of a teacher of computer programming, the principal weakness of Piagetian theory is that it places an emphasis on the independent construction of knowledge by students to the exclusion of the social

aspect of learning and does not provide ideas about what could be done to facilitate schema construction through improvements in course construction or teaching strategies. Vygotsky promoted guided discovery by providing novice programmers with the assistance and feedback they required during problem solving. Vygotsky's theory promotes gradual changes using cultural tools and social contact. In this study, it was found that scaffolding plays an important role in keeping participant's practice moving toward improvement. The scaffolding provided to the participants in this study ranged from low level support (i.e. hard scaffolding – using software tools) to high levels of support (i.e. providing the exact solution using a stepwise refinement). Information about the problem solving behaviours of participants shows that the concepts of ZPD and scaffolding provide a useful way of describing learning within the context of computer programming. They also provide information about how a teacher can prompt learning and what suitable software tools can be used to prompt learning. The instrument designed for this thesis allowed the researcher to identify the ZPD of the participants and to some extent to predict which the participants could take advantage of the scaffolding given to them during the retrospective interview. The participants with a larger ZPD demonstrated their ability to move forward. As a result of practising problem solving supported by scaffolding, they were able to recall and make associations based on their past experience in order to achieve a new, higher level of understanding. Participants with a smaller ZPD found difficulties recalling what they had previously learned and therefore found it difficult to solve the more advanced problems.

One of the main deficiencies of Vygotsky's theory is that it does not explain how the process of cognitive development occurs. We have been able to use the theory to engage participants in successful teaching strategies but not to develop an understanding of the cognitive processes that take place as novice programmers learn to program.

Sfard's theory, like Vygotsky's, focuses on engaging the learner in the discovery process by providing learners with assistance from a more knowledgeable source. Sfard's framework while developed as a theory for explaining concept development in mathematics is also relevant to learning computer programming. Like mathematics, programming involves "tightly integrated concepts". Sfard explored the three stages of mathematical concept development. The empirical evidence derived from this longitudinal study revealed that these same three stages are involved in learning to write a computer program and the development of mental abstraction of the programming plan (pattern). The findings reported in this study indicate that at the interiorization phase, the

cognitive demands on the participants are high. It may be that the participants have difficulty in simultaneously attending to an understanding of the interactions among the sub-problems, mastering the language syntax, reading, tracing, and testing. At the condensation phase, the participants started to become aware of common patterns, and the interactions among the program patterns, and used them as templates that could be manipulated in diverse ways to enable them to correctly transfer the learned solution to a new characteristic. At the reification phase, some participants are more successfully than others showed evidence that they were able to recall relevant schemas and to construct a solution that demonstrated that they were thinking about this solution as a unified entity. The adaptation of Sfard's stages proves to be of particular value when interpreting the process of the novice programmers' development from a cognitive perspective.

These three stages contribute to deeper understanding of novice programmers' way of developing patterns and reusing them in solving another programming task (abilities to view a current problem in terms of old problems). They also provide information that shows educators how to teach programming in away that allow them to organise related topics of the programming course.

When a novice first has to write a computer program, the intrinsic load is high and that form of load will only get lower when the person has learned the language and the programming constructs, and has had practice at programming. In this research, CLT has been used to explain why some participants could not progress or had difficulties with certain aspect of learning. The concepts of intrinsic and extraneous load were used to gain an understanding of the difficulties being faced by participants who were struggling with a problem and whose cognitive resources were probably overloaded and so were unable to build the necessary new schema. CLT assumes that learning results in schema development but it does not provide a theory about how schema are developed.

Information about the problem solving behaviours of participants shows that the use of simple-to-complex sequencing, ordering and organising questions was a useful approach to reducing extraneous cognitive load and thereby increasing the working memory available for the germane load (schema acquisition and construction). In addition, information about the problem solving behaviours of participants shows that when participants were struggling to solve a programming task, the cognitive load imposed during the writing process was probably too high but could be reduced if the interviewer guided their knowledge acquisition processes.

Chapter 8. Conclusion

8.1. Overview of Research

The main aim of this study was to gain a deeper understanding of the ways in which novice programmers learn to program, with an emphasis on their cognitive development processes. This was achieved by following the same participants during their first year learning to program at the AUT. The participants were observed solving code writing tasks from a set of related, progressively more difficult tasks while providing think aloud information about what they were doing and thinking about during their problem solving. They were also interviewed retrospectively.

Information obtained from observation, the think aloud protocols and retrospective interviews was used to analyse the match between the cognitive processes used by the participants and the ideas about learning presented in selected cognitive theories (Piaget, Vygotsky, Sfard and CLT) in a way that provides a reasonable explanation about learning to program and the extent to which these theories fall short as an explanation of cognitive development in the programming domain.

8.2. Research Questions

This research was guided by five main questions.

Research question 1 (Q1): *Can we develop a framework that describes the difficulty of novice code writing tasks?*

A novel task difficulty framework was developed which consisted of a new empirically verified software metric and a SOLO classification for code writing tasks. It was found that these two dimensions – an objective and a subjective measure – were needed to fully explain the difficulty of code writing tasks, and that when combined they were a useful way of distinguishing tasks. The software metric reflected the structural complexity, size and readability of the solution code, and the SOLO taxonomy the level of cognitive complexity of the code writing task. This framework was then used to guide the development of the sequences of code writing tasks used in this research. The initial data transcription and analysis, presented in Chapter 6 and Appendix A, showed that the questions which were more difficult according to the framework were also the questions the students found the most difficult to solve. Indications of increased difficulty for a student included: more time on task, higher frequency of compilation and test runs, and an increased reliance and need for assistance and whether or not a problem was solved.

This difficulty framework should be applicable for most novice programmer code writing tasks in the first year of learning to program and should prove to be a useful tool for educators and researchers when designing such tasks.

Research question 2 (Q2): *How do novice programmers integrate new programming structure or elements into their current understanding of code?*

Through the research methodology adopted it was possible to identify common novice programmer code writing strategies. Four approaches to code writing emerged based on the observations of the novice programmers in this study. The most common strategy was to adopt a stepwise design in which they broke down the problem into manageable subparts and then recomposed these subparts to form a new solution. This is not at all surprising as the tasks were designed so that each task built on the previous task's programming schema or plans in some way. This finding has implications for teaching as it suggests that explicitly designing programming exercises to progressively build on concepts and mental schemas improves learning.

There is also strong evidence to suggest that when novice programmers cannot identify all the subparts of a problem they tend to fall back on the most familiar aspect of a task or problem first and then try to build a solution from that point. It is also clear that whenever possible they rely on building code sequentially (combining different or repeating the same schemas in sequence) and tend to have difficulty when a task requires the combination of cognitive schemas in a non-sequential manner. Lastly when novice programmers cannot reach a solution within a relatively short time they tend to resort to a trial and error approach to programming. Participants unable to retrieve an existing schema tended to program using a trial and error approach which inevitably led to failure. None of these observations are surprising and similar observations have been made in published research studies of novice programmers.

It is clear from this research that novice programmers can be taught to be aware of common patterns and the way in which patterns relate to each other and can be combined. This awareness lends itself to more successful programming strategies. In cases where the participants were able to recognise that a problem was analogous to a previously solved problem they were almost always able to reuse their prior knowledge to solve the new task and were able to identify the variation or differences between these tasks. In such cases they tended to first write a solution fairly quickly without compilation or testing and then used the feedback from the tests and the compiler to refine their solution.

Scaffolding and feedback seemed to play a major and constructive role in learning for the participants of this study. Through the use of a strategic scaffolding approach, inspired by Perkins and Martin's (1985) soft scaffolding model, students were able to integrate new programming knowledge into their existing schemas. The researcher found that using a stepwise refinement process, described in Section 3.7, as one means of scaffolding was a particularly successful approach. Much of the time participants were able to take advantage of scaffolding and feedback given to them to achieve a new, higher level of understanding. However, more often than not the poorer performing participants while able to make use of scaffolding to build a correct program were unable to fully recall that knowledge and apply it to future tasks. It is clear that these students require more time to learn, more guided learning and more practice.

The sequence of programming tasks designed for this research encouraged participants to think of new ways to apply previously learned concepts. During the think aloud sessions some participants were able to develop broader schemas for recognizing familiar program structures. Some participants could not immediately draw on relevant prior knowledge but later, as they practised programming supported with different kinds of scaffolding, they succeeded in applying their knowledge to different programming concepts and contexts (either a Robot World or a native Java task).

In this study, we were able to identify whether or not a participant was within their Comfort Zone and/or ZPD when solving a question. Anecdotally, the researcher was able to identify the ZPD of the participants and to some extent to predict which participants would be able to take advantage of the scaffolding given to them during the retrospective interview. If a participant was within their Comfort Zone they were able to solve the task independently and therefore had reasonably robust cognitive schema which could be applied to solve the problem. In the ZPD, but outside of the Comfort Zone, the participants were able to solve a task with assistance; for the weaker students working too far towards the edge of their ZPD meant that they did not form adequate cognitive schema as a result of solving a problem, and in order to solve the problem they required a high degree of intervention. For participants, where a task was well within their ZPD less intervention was required and the intervention was less directive thus learning more by discovery; schemas formed as a result of solving the task were able to be used reliably to solve future tasks.

Research question 3 (Q3): *Does a student's approach to integrating new knowledge change over time? If it does, what triggers this change?*

There were no obvious trends in change over time. However the participants did tend to move between different strategies depending on the task. Factors such as prior knowledge, student motivation and attitude, strength of existing schemas and difficulty of the task in terms of structure and the level of abstraction of thinking required affected the approach used in learning.

The fact that no change was observed could be because there is not sufficient progress in learning and knowledge in the first year of programming to be able to detect a change or require a change. Moreover, in the programming courses taken by these students reflection on learning and the way in which programming can be learnt effectively are not aspects that are explicitly taught or focused on in instruction.

Faced with real world problems that require moral reasoning, adult learners have been shown to use different levels of argument depending on the problem and to move backwards and forwards between simple child-like reasoning and relatively advanced reasoning in a similar manner to that observed for this research's participants. More advanced reasoning can be taught and learners can be taught how reason or reflect on their own reasoning. Successful students already have a good approach to learning programming and coping with new concepts and are able to progress by integrating these new concepts into their knowledge structure. In the case of weaker students, explicitly teaching metacognitive techniques could possibly be of help.

Research question 4 (Q4): *What specific properties does a programming question or task need to trigger a learning event?*

In order to significantly progress learning a task needs to be designed to cause cognitive dissonance and subsequent reconstruction. To trigger incremental steps in learning existing knowledge is used and transferred to a new situation and retained over time. In order for learning to occur tasks should be within the novice programmers ZPD to ensure that a learning event is triggered. The task needs to be sufficiently challenging and require effort but on the other hand be within the ZPD and require minimal assistance from a significant other. This is a delicate balance.

Ultimately students are at different stages of learning at different times. Determining which tasks should be given to a student at a given time depends on their current stage of development. Because we can determine their ZPD we should be able in theory determine

which tasks are most appropriate at a given time. The ideas included in transfer¹⁴ of learning are useful to determining the properties of a task which trigger different learning events.

As a result of observing the participants solving various types of tasks the following hypotheses have been developed:

1. Tasks which are Isomorphic have a tendency to consolidate existing cognitive schemas. Such tasks require the minor adaption of automated schemas.
2. Tasks which are Glued Isomorphic require the retrieval and adaptation of more than one cognitive schema. Such tasks may require minimal or significant reorganization of the automated schema and become progressively more challenging as the degree of reorganization and recombination of schemas increases.
3. Tasks which are Far Transfer tasks trigger significant leaps in learning where the other two types of tasks tend to trigger more incremental learning. These types of tasks requires a shift from one concept to a new concept. For example transferring a 1D array iteration schema to a 2D array schema.

All of these types of tasks have the potential to trigger learning experiences. The researcher theorises that for most students *Isomorphic* and *Glued Isomorphic* tasks are normally within their ZPD. However *Far Transfer* tasks are often outside of the ZPD for students whose schemas and programming knowledge are fragile. Careful consideration therefore is required as to when such tasks are given to a student – given too early such tasks can stall progress and discourage the learner.

Research question 5 (Q5): *Can we develop a cognitive framework that describes the ways in which novice programmers integrate new programming structure or elements?*

The information gathered for this research was not sufficient for the construction of a fully formed cognitive framework, and additional information gained through future research is required. Nevertheless, the data gathered over a full year points to some important components that could form part of a future complete framework.

The results of this research indicate that both cognitive and sociocultural approaches are important in the development of knowledge of novice programmers. All theories

¹⁴ The types of transfer tasks used in the remaining discussion were defined by the researcher for the purpose of studying novice programmers (Section 5.5).

discussed in this thesis, except Piagetian theory, focused not only on intellectual development but also on social interaction as an important factor in learning and development. Of the theories examined two were found to be the most useful—Vygotsky’s notions of the Zone of Proximal Development, the role of *more knowledgeable others* and more recent ideas about scaffolding, and Sfard’s theory of concept development.

The researcher was able to use Vygotsky’s concepts of ZPD and scaffolding as a successful teaching strategy for engaging participants in learning and to progress participant learning but these concepts were not sufficient to explain the cognitive processes that take place as novice programmers learn to program. Sfard’s ideas appeared to be useful in explaining the process of the novice programmers’ development from a cognitive perspective.

8.3. Reflections on the Think Aloud Method

Think aloud has been used effectively in the areas of psychology and education to investigate cognitive processes but the method does have some recognised limitations. The most critical issue is that the requirement to problem solve and to speak about the reasoning being used simultaneously may be too difficult for some participants (Branch, 2000). This problem was evident early on in this research – the participants were very often unable to both think aloud and to focus on the new concepts and the unfamiliar task of code writing. Most of the students in this study were English as a second-language students which placed an additional burden in terms of thinking aloud. They often found it difficult to think aloud and tended to switch between their native language and English while solving the programming tasks. At times the participants had difficulty finding the appropriate English vocabulary to describing their reasoning.

Unfortunately think aloud is the only tool which researchers have to try and find out what a person is thinking when solving a complex problem (Van Someren et al., 1994). In this research the addition of video and retrospective interviews was found to enhance the data acquired and provide greater insight into the participants’ thought processes. Even with the revised method, for some participants the data acquired was sparse. It is important to recognize that there is a limitation on the information acquired though retrospection because working memory capacity is limited. Thoughts are retained briefly in working memory and may be lost as soon as new thoughts supersede it – only certain information is retained in long term memory, triggered by video playback and therefore recalled in the retrospection. However, as the retrospection was always undertaken immediately after the think aloud this issue was minimized.

The influence of the researcher on the method must also be acknowledged. In order to minimize interviewer influence the think alouds were conducted in so far as it was possible without prompts from the researcher. The interpretation of the think aloud data required the researcher to make her own inferences. With the researcher aware of not introducing her own bias to the transcription process the reconstruction of participant remarks was as much as possible “literal” and closely connected to context by transcribing using video and audio. However, some utterances appeared to consist of more than one thought process and therefore required interpretation by the researcher.

Inadvertently changing a novice’s approach to solving the programming task was always a possibility as a result of the intervention model used in this thesis. However, the intervention model used was a valuable way of ensuring that participants benefited from their participation and were therefore motivated to continue to participate throughout the year. Many participants were able to complete a programming task as a consequence researcher assistance and this process reflects teaching and learning. The interventions were recorded and their effect on the participants’ subsequent thought processes noted and considered in the analysis.

Participating in a research study such as this requires a real time commitment from the students involved. Some participants withdrew from the research on completing the P1 course. Others frequently postponed meeting sessions due to other commitments. The participants who completed the study commented on the usefulness of participation and how it improved their understanding of programming. Interestingly it was not just the highly motivated and engaged students who persisted in the study and the spread of participants in terms of performance in the study, and on the relevant programming course, was maintained until the end of the study.

8.4. Validity, Reliability and Generalisability of the Difficulty Framework

Validity in quantitative research refers to whether the research truly measures that which it was intended to measure and whether the results are accurate (Joppe, 2014).

One obvious threat to the *validity* of the difficulty framework’s WM is that there is no reliable way of identifying how the participants arrived at a given correct or incorrect answer. This threat to *validity* is considered to be minimal due to the triangulation of this method with the think aloud results. The observations of the participants solving the code writing tasks confirmed that the tasks they found easy were the tasks which were measured as easy by the WM. Similarly, tasks measured as hard by the WM were also

difficult for the participants. A link between time spent on the task, the number of compilations required, and degree of success and the difficulty of the task as measured by the WM was noted.

One possible threat to *validity* in using naturally occurring data, such as exam answers or online test responses, is that the students may have “cheated” on the exam/test. This possibility is minimal because of the closed programming environment used in online tests and through the use of invigilators. The quantitative data for this research was based on series of programming tests held throughout the P1 course. The programming tests were computer-based and open book. These tests were about three hours long and were conducted under formal examination conditions; therefore, the opportunity for plagiarism was low.

Another threat to the *validity* of the results of this research concerns the mode of marking and the rubrics for marking. In order to avoid issues related to potential inconsistencies between markers and in marking schemas which award marks for incomplete and often non-functional answer code, the researcher re-marked the assessments explicitly for the purpose of this research.

The *reliability* of results of a quantitative research method refers to the extent to which results are consistent over time and are an accurate representation of the total population under study is a measure of the data’s reliability (Joppe, 2014).

In order to ensure the *reliability* of the difficulty framework a purposeful non-random sampling strategy was used (see details in section 3.5.2.1). Because AUT ethical consent is based on the principle of informed and voluntary consent it is possible that the sample is not representative of the entire cohort. The overall grade distribution of the entire cohort was compared with that of the sample – they were the same. Thus the sample is considered to be representative and reliable.

Another issue worthy of note is that of the *generalisability* of the framework. This framework is limited to code writing tasks that are typically given to novice programmers in their first year of learning to program. Moreover, it is further limited to a small degree due to the nature of the course in which this research is situated. The course was largely procedural, even though it used a micro-world. For an objects first approach to teaching and learning programming the WM would probably need to be extended to include object oriented metrics. Thus, in the context of this thesis, the notion of an *objective* measure of difficulty is considered in the positivist sense of developing an objective measure in research involving specific students, situated in a specific course, and at a specific time.

However, the tasks presented to the students involve programming schemas and concepts that are typical for current first programming courses globally. This means that it is likely that the WM metric will be found to be applicable to other courses, in different programming languages, and taken by other students.

8.5. Trustworthiness of the Think Aloud Data

The four criteria for trustworthiness in quantitative research are:

1. Credibility

Activities that increase the likelihood of *credibility* in a qualitative study include prolonged engagement, persistent observation, and triangulation (Lincoln & Guba, 1985). Each of these activities has been part of this research project.

As part of the observational method used in this research retrospection interviews were included to ensure the *credibility* of the data gathered. As part of the retrospection process, any data that was in doubt was checked with the participant and participant feedback sort. This process of asking the participant, albeit indirectly in this research, is known as a “*member check*”. Guba and Lincoln (1985) consider member checks an important provision that can be made to reinforce a study’s credibility.

Triangulation further contributes towards the *credibility* of this study. Triangulation is “*checking information that has been collected from different sources or methods for consistency of evidence across sources of data*” (Mertens, 2005, p. 225) and was core to the research method employed in this research. *Credibility* of data was ensured, therefore, by:

- data being collected from multiple participants with diverse code writing abilities,
- the code writing tasks being designed using a verified difficulty framework,
- using more than one theoretical scheme in the interpretation of the observed phenomena.

Further *credibility* is given to the result and data in the study by:

- the researcher having 20 years of experience teaching computer science,
- a pilot study was conducted to ensure that the methods were appropriate for the purposes of the research.

2. Transferability

Guba and Lincoln (1989) define *transferability* as the extent to which results of a study can be generalized to other situations (settings, contexts, or populations). There are some concerns associated with transferability such as precisely how the findings of a particular

study could be replicated or applied in different settings. One way to overcome this concern is to use many cases (participants) in studies to enhance replicability (Marshall & Rossman, 2006). Also, providing a detailed description report (“*thick description*”) on the interpretation of a particular research sample (such as choosing of data collection methods and participants, quotations from interviews, analysis methods, process of analysis, and the inferences the researcher has come) helps other researchers to decide for themselves the extent to which the findings of a particular study can be transferred to their own research (Davis, 1995). A detailed description increases the chances of transferability in qualitative research. However, it must be noted that it is up to the reader to judge if the data is rich enough to make any comparisons.

Transferability has been ensured in this research through a thick description in the following ways. An in-depth detailed description was made to ensure that the group of students investigated, the choice of participants for the think aloud sessions, the course content of the programming course they studied, the programming tasks they solved, the research approach taken in collecting the data, and the analyses performed were described in as much detail as possible. Moreover, an in-depth discussion and interpretation of the relevant aspects of the cognitive theories considered in the research was presented along with a detailed interpretation of the observations and how these related to these cognitive theories.

3. Dependability

A third criterion of major concern in qualitative research is its *dependability*. *Dependability* refers to the degree of consistency and reliability of the data and the interpretation of the results of a study. Lincoln and Guba write it is argued that if *credibility* is achieved, *dependability* is also achieved: “*Since there can be no validity without reliability (and thus no credibility without dependability), a demonstration of the former is sufficient to establish the latter*” (p. 317). Techniques related to *credibility*, thus ensure that *dependability* is achieved.

4. Confirmability

The fourth criterion of concern in qualitative research is *confirmability*. *Confirmability* includes the confirmation that the results of a particular data are related to the study conducted. This issue can be overcome if a researcher is able to provide a detailed description of the data collected - “*tracked to its source*” (Mertens, 2005, p.257). This would enable other researchers to modify, confirm, or reject the interpretations and inferences that are made (Mertens, 2005).

In the present study the difficulty framework was used to inform the design of the think aloud tasks, and therefore, an interpretative qualitative approach was adopted to extract patterns of behaviour arising from the relationship between verbalization (cognitive processes) and the task solution (final product quality).

Analysis in studies of cognitive processes is always subjective because a researcher influences both the collection and the interpretation of the data. The level of *subjectivity* was mitigated by adopting the following practices:

- A fixed coding schema was used to categorise the transcribed.
- A verbal protocol was specified and used in order to understand and interpret the meanings of the actions of participants under study.
- The researcher provided a detailed account of how data was collected from the participants, the intervention model used, and a synthesis of limitations of the findings and conclusions reached.

8.6. Implications for Teaching

The difficulty framework developed in this research should provide educators with a way of estimating the difficulty of novice code writing tasks. This framework can be used to appropriately sequence and design learning tasks to promote schema development in novice programmers. Moreover, using programming tasks designed this way supported with explicit teaching of metacognitive techniques to make students aware of how their knowledge is effectively adapted and expanded, should improve student learning.

The adoption of Sfard's concept of developmental phases and the cyclical nature of concept development contributes to deeper understanding of novice programmers' ways of developing schemas and reusing them in solving another programming task. This indicates the ability to view a current problem in terms of old problems. The results of this research therefore suggest a number of core principles which could be applied to the teaching and learning of programming that help promote concept development. These are:

1. Consider what are the critical or core concepts in the course/paper.
2. Consider sequences of tasks. Teach simple concepts first in isolation and ensure that the students are exposed to sufficient examples so that the concept is reified before moving onto a new concept.
3. Ensure that examples and tasks related to a concept include variation. Tasks that are in essence the same as an original example or task can be presented differently.

4. When designing assessments and exercises, carefully consider the number of concepts that are actually involved in each task – educators are prone to underestimating the number of concepts and the number of linkages between concepts.

In teaching novice programmers it is recommended that assessments be designed which establish each student's individual ZPD and that students are provided with individualised formative code writing tasks which are intended to expand their ZPD. If students are not provided with tasks which incrementally and slowly expand their ZPD, it is likely that they will eventually give up or never move forward from the point in the course where the tasks fell outside of their ZPD. While providing them with tasks that expand their ZPD, it is also important to ensure that some tasks are within their CZ in order for students to develop a level of confidence.

The participants who were able to recognise similarities and differences in the tasks presented to them were more able to retrieve relevant prior mental schemas and apply them to new tasks. Teaching students in a way which allows them to develop an awareness of the different ways in which code may be written to solve the same and similar problems should also improve learning. Thompson (2010) discussed the importance of the use of variation theory to assist learners to understand threshold or core concepts and designed programming tasks using variation theory. The research undertaken in this thesis gives weight to Thompson's conjecture that teaching with variation is important for concept development.

It is clear from this study that appropriate scaffolding is critical to student learning in the early stages of learning to program. It is the view of the researcher that developing a set of guiding principles for assisting students in order to promote independent learning capability in novice programmers is essential for effective teaching especially in practical programming laboratories. The scaffolding guidelines used in this research (section 3.7) should provide a useful starting point for teachers.

Software tools, debugging and code tracing play an important role in learning to program and their use must be taught as an important aspect of programming for novice programmers – too often as in this course in which this research was undertaken such aspects are expected to evolve naturally and are never explicitly taught. Such tools can be valuable for scaffolding learning once students have been taught how to use them effectively.

8.7. Future Research

The difficulty metric for this research was developed using code writing tasks from on-line examinations of a course in Java and applied to designing questions in Java and in the context of the Java based Robot World used as an instructional tool in P1 at the time of this research. The applicability of the framework to both programming contexts suggests that the framework should be generalizable to other programming languages and pedagogical approaches. One possible limitation of the framework is the fact that it was developed for a course which used a procedural-first programming pedagogy. For courses adopting an objects-first pedagogy additional software metrics may be required for the objective dimension of the difficulty framework. Further research that explores the objective metric with respect to different pedagogies and programming languages is warranted.

It would be interesting to extend this study to other aspects of programming, including aspects of design. Most empirical studies have focused on the difficulties that novices have when trying to understand object-oriented concepts, but we do not understand, in terms of cognitive processes, why they have difficulties with such concepts. Future research could focus on knowledge acquisition of novice programmers learning in the full object-oriented programming or objects-first paradigm.

It is important that instructors are mindful of the type of scaffolding/assistance they give students in order to promote independent learning while keeping momentum in learning. Thus, one other interesting avenue of research would be to empirically investigate in more depth the effects of the intervention model employed in this study and the impact of the various interventions on student learning.

Understanding where each student's ZPD lies would be useful. Research investigating a more accurate measure of a student's ZPD would clearly be useful in planning courses and programming tasks. Eventually it might lead to a means of providing each student with personally tailored programming exercises targeted at progressing their learning and promoting concept development.

Finally, further research as to how the ZPD and Sfard's phases of concept development relate to each other and might be combined could result in a new refined cognitive framework that describes the ways in which novice programmers integrate new knowledge into their cognitive schema.

References

- Ambrose, S. A., Bridges, M. W., Dipietro, M., Lovett, M. C., & Norman, M. K. (2010). *How learning works: Seven research based principles for smart teaching*. JOSSEY-BASS: A Wiley Imprint. Retrieved from www.josseybass.com
- Anderson, J., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, 8(2), 87–129.
- Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., ... Wittrock, M. C. (2001). *A taxonomy for learning, teaching, and assessing: A Revision of Bloom's taxonomy of educational objectives*. New York: Longman.
- Atman, C. J., & Bursic, K. M. (1998). Verbal protocol analysis as a method to document engineering student design processes. *Journal of Engineering Education*, 87(2), 121–132.
- Barker, R., & Unger, E. (1983). A predictor for success in an introductory programming class based upon abstract reasoning development. *SIGCSE Bull.*, 15(1), 154–158.
- Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). The Goal Question Metric approach. *Encyclopedia of Software Engineering*, 2, 1–10.
- Bennedson, J., & Caspersen, M. (2008). Abstraction ability as an indicator of success for learning computing science? In *Proceedings of the 4th International workshop on Computing Education Research (ICER'08)* (pp. 15–26). Sydney, Australia: ACM.
- Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press.
- Black, T. R. (2006). Helping novice programming students succeed. *Journal of Computing Sciences in Colleges*, 22(2), 109–114.
- Bloom, B. S., Krathwohl, D. R., & Masia, B. B. (1956). *Taxonomy of educational objectives the classification of educational goals, Handbook 1: Cognitive Domain*. New York: Longmans, Green & Co.
- Bogdan, R. C., & Biklen, S. K. (2006). *Qualitative research for education: An introductory to theory and methods* (5th Ed.). Needham Heights, MA: Allyn and Bacon.
- Bormuth, J. R. (1971). *Development of standards of readability: Toward a rational criterion of passage performance*. Final report, U.S. Office of Education, Project No. 9-0237. Chicago: University of Chicago.
- Börstler, J., Caspersen, M. E., & Nordström, M. (2007). *Beauty and the beast — Toward a measurement framework for example program quality*.
- Bower, M. (2008). A taxonomy of task types in computing. *SIGCSE Bull.*, 40(3), 281–285.
- Braarud, P. (2001). Subjective task complexity and subjective workload: Criterion validity for complex team tasks. *International Journal of Cognitive Ergonomics*, 5(3), 261–273.
- Branch, J. L. (2000). Investigating the information-seeking processes of adolescents: The value of using think alouds and think afters. *Library & Information Science Research*, 22(4), 371–392.

- Briand, L. C., Morasca, S., & Basili, V. R. (1996). Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1), 68–86.
- Brown, J. D., & Rodgers, T. (2002). *Doing second language research*. Oxford University Press.
- Bruce, B., Rubin, A., & Starr, K. (2015). Why readability formulas fail. *IEEE Transactions on Professional Communication*, 24(1), 50–52.
- Brush, T. A., & Saye, J. W. (2002). A summary of research exploring hard and soft scaffolding for teachers and students using a multimedia supported learning environment. *Journal of Interactive Online Learning*, 1(2), 1–12.
- Bryman, A. (1984). The debate about quantitative a question of method qualitative research : or epistemology ? *British Journal of Sociology*, 35(1), 75–92.
- Cafolla, R. (1988). Piagetian formal operations and other cognitive correlates of achievement in computer programming. *Journal of Educational Technology Systems*, 16(1), 45–55.
- Campbell, D. (1988). Task Review Complexity : A review and analysis. *Academy of Management Review*, 13(1), 40–52.
- Cardell-Oliver, R. (2011). How can software metrics help novice programmers? In *Proceedings of the 13th Australasian Computing Education Conference (ACE'11)* (pp. 55–62). Darlinghurst, Australia: Australian Computer Society, Inc.
- Carnegie Mellon University. (2006). What is Alice and what is it good for? Retrieved November 16, 2015, from http://www.alice.org/index.php?page=what_is_alice/what_is_alice
- Caswell, R., & Nisbet, S. (2005). Enhancing mathematical understanding through self-assessment and self-regulation of learning: The value of meta-Awareness. In *Proceedings of the 28th conference of the Mathematics Education* (pp. 209–216). Sydney: MERGA Inc.
- Chandler, P., & Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8(4), 293–332.
- Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, 4, 55–81.
- Checkstyle. (2016). Checkstyle. Retrieved February 17, 2016, from <http://checkstyle.sourceforge.net/>
- Chi, M., Glaser, R., & Rees, E. (1982). Expertise in problem solving. In *Advances in the psychology of human intelligence* (Vol. 1, pp. 7–75). Hillsdale, NJ: Erlbaum.
- Chi, M. T. H., & Bassok, M. (1989). How students study and use examples in learning to solve problems. *Cognitive Science*, 13, 145–182.
- Clear, T., Whalley, J. L., Lister, R., Carbone, A., Hu, M., Sheard, J., ... Thompson, E. (2008). Reliably classifying novice programmer exam responses using the SOLO taxonomy. In *Proceedings of the 21st Annual NACCCQ Conference of the National Advisory Committee on Computing Qualifications (NACCCQ'08)* (pp. 23–30). Auckland, New Zealand.
- Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds & Machines*, 17, 169–184.
- Cole, M. (1997). *Cultural psychology: A once and future discipline*. Cambridge: The Belknap Press of Harvard University.

- Collins, A. M., Brown, J. S., & Holum, A. (1991). Cognitive apprenticeship: Making thinking visible. *American Educator*, 15, 6–11.
- Commons, M. L., Richards, F. A., & Armon, C. (1984). *Beyond formal operations: Late adolescent and adult cognitive development*. NY: Praeger.
- Cooper, G., & Sweller, J. (1987). Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, 79(4), 347–362.
- Cooper, G., Tindall-Ford, S., Chandler, P., & Sweller, J. (2001). Learning by imagining. *Journal of Experimental Psychology*, 7(1), 68–82.
- Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some empirical results for Neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the 14th Australasian Computing Education Conference (ACE'12)* (Vol. 123, pp. 77–86). Melbourne, Australia: Australian Computer Society Inc.
- Creswell, J. W. (1995). *Research design: Qualitative and quantitative approaches*. SAGE Publications.
- Creswell, J. W. (2009). Editorial: Mapping the field of mixed methods research. *Journal of Mixed Methods Research*, 3(2), 95–108.
- Davidson, C. (2009). Transcription: Imperatives for qualitative research. *International Journal of Qualitative Methods*, 8(2), 35–52.
- Davies, S. P. (1991). Characterizing the program design activity: neither strictly top-down nor globally opportunistic. *Behaviour & Information Technology*, 10(3), 173–190.
- Davis, K. A. (1995). Qualitative theory and methods in applied linguistics research. *TESOL Quarterly*, 29, 427–453.
- Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons problems. In *Proceedings of the 4th international workshop on Computing Education Research (ICER'08)* (pp. 113–124). Sydney, Australia: ACM.
- Dijkstra, E. W. (1997). *A Discipline of programming* (1st Ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Dougherty, J. P. (2007). Concept visualization in CS0 using Alice. *Journal of Computing Sciences in Colleges*, 22(3), 145–152.
- Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), 57–73.
- Dubinsky, E. (1991). Reflective abstraction in advanced mathematical thinking. In D. O. Tall (Ed.), *Advanced Mathematical Thinking* (pp. 95–123). Dordrecht: Kluwer.
- Eckerdal, A., & Berglund, A. (2005). What does it take to learn “Programming Thinking”? In *Proceedings of the 1st International workshop on Computing Education Research (ICER'05)* (pp. 135–142). Seattle, WA, USA: ACM.
- Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis: Verbal reports as data*. Cambridge, MA: Massachusetts Institute of Technology.
- Falkner, K., Vivian, R., & Falkner, N. J. G. (2013). Neo-piagetian forms of reasoning in software development process construction. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering (LATICE'13)* (pp. 31–38). IEEE Computer Society.

- Field, A. (2009). *Discovering statistics using SPSS* (3rd Ed.). SAGE Publications.
- Fink, A. (2003). *How to sample in surveys* (2nd Ed.). SAGE Publications.
- Fischer, G. (1986). Computer programming: A formal operational task. In *16th Annual Symposium of the Piaget Society*. Philadelphia, PA, USA.
- Fitzgerald, B., & Howcroft, D. (1998). Competing dichotomies in IS research and possible strategies for resolution. In *Proceedings of the International Conference on Information systems (ICIS'98)* (pp. 155–164). Atlanta, GA, USA: Association for Information Systems.
- Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental representations of programs by novices and experts. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI'93)* (pp. 74–79). Amsterdam, Netherlands: ACM.
- Flavell, J. H. (1977). *Cognitive development*. Englewood Cliffs, NJ: Prentice Hall.
- Flavell, J. H., & Piaget, J. (1963). *The developmental psychology of Jean Piaget*. Princeton, NJ: D Van Nostrand Company.
- Flesch, R. (1948). A new readability yardstick. *Journal of Applied Psychology*, 32, 221–233.
- Fleury, A. E. (2000). Programming in Java. *ACM SIGCSE Bull.*, 32(1), 197–201.
- Flower, L., & Hayes, J. R. (1980). The cognition of discovery: Defining a rhetorical problem. In *Landmark Essays on Writing Process* (Vol. 31, pp. 63–74). College Composition and Communication.
- Fuchs, L. S., Fuchs, D., Prentice, K., Burch, M., Hamlett, C. L., Owen, R., ... Jancek, D. (2003). Explicitly teaching for transfer: Effects on third-grade students' mathematical problem solving. *Journal of Educational Psychology*, 95(2), 293–305.
- Fuller, U., Johnson, C. G., Cukierman, D., Hernán-losada, I., Rey, U., Carlos, J., ... Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *SIGCSE Bull.*, 39(4), 152–170.
- Gagné, R. M., Briggs, L. J., & Wager, L. J. (1992). *Principles of instructional design* (4th Ed.). Harcourt Brace College.
- Ginat, D., & Menashe, E. (2015). SOLO Taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)* (pp. 452–457). Kansas City, Missouri, USA: ACM.
- Gluga, R., Kay, J., Lister, R., Kleitman, S., & Lever, T. (2013). Coming to terms with Bloom : an online tutorial for teachers of programming fundamentals, 147–156.
- Gómez-Albarrán, M. (2005). The teaching and learning of programming: A survey of supporting software tools. *Computer Journal*, 48(2), 130–144.
- Gray, E., & Tall, D. (2007). Abstraction as a natural process of mental compression. *Mathematics Education Research Journal*, 19(2), 23–40.
- Gray, S., Clair, C., James, R., & Mead, J. (2007). Suggestions for graduated exposure to programming concepts using fading worked examples. In *Proceedings of the 3rd International workshop on Computing Education Research (ICER'07)* (pp. 99–110). Atlanta, Georgia, USA: ACM.
- Griffin, J. (2016). Learning by taking apart: deconstructing code by reading, tracing, and debugging. In *Proceedings of the 17th Annual Conference on Information*

- Technology Education (SIGITE'16)* (pp. 148–153). Boston, Massachusetts, USA: ACM.
- Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*, 25(2), 199–237. Retrieved from <http://www.tandfonline.com/doi/full/10.1080/08993408.2015.1033142>
- Guba, E. G., & Lincoln, Y. S. (1989). *Fourth generation evaluation*. SAGE Publications.
- Gunning, R. (1952). *The technique of clear writing*. McGraw-Hill.
- Haaster, K. V., & Hagan, D. (2004). Teaching and learning with BlueJ: an Evaluation of a pedagogical tool. In *Information Science + Information Technology Education Joint Conference* (pp. 455–470). Rockhampton, Queensland, Australia.
- Hall, A. (1970). A conversation with Jean Piaget and Bärbel Inhelder. *Psychology Today*, May, 3, 25-32-56.
- Halstead, M. H. (1977). *Elements of software science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc.
- Hannafin, M., Land, S., & Oliver, K. (1999). Open learning environments: Foundations, methods, and models. In C. Reigeluth (Ed.), *Instructional design theories and models* (Vol. 2, pp. 115–140). Mahway, NJ: Erlbaum.
- Harwell, M. R. (2011). *The SAGE handbook for research in education : Pursuing ideas as the keystone of exemplary inquiry*. SAGE Publications.
- Hattie, J., & Purdie, N. (1998). The SOLO model : Addressing fundamental measurement issues. *Teaching and Learning in Higher Education*, 145–176.
- Hogarty, K. Y., Hines, C., Kromrey, J., Ferron, J., & Mumford, K. (2005). The quality of factor solutions in exploratory factor analysis: The influence of sample size, communality, and overdetermination. *Educational and Psychological Measurement*, 65(2), 202–226.
- Hook, P. (2016). The learning process. Retrieved February 22, 2016, from http://pamhook.com/wiki/The_Learning_Process.
- Hsued, Y. (2005). The lost and found experience: Piaget rediscovered. Retrieved November 18, 2015, from <https://sites.google.com/site/assocforconstructteaching/journal/the-constructivist-archive>.
- Huber, L. N. (1988). Computer learning through Piaget's eyes. *Classroom Computer Learning*, 6(2), 39–43.
- Hudak, M. A., & Anderson, D. E. (1990). Formal operations and learning style predict success in statistics and computer science courses. *Teaching of Psychology*, 17(4), 231–234.
- Hunkins, F. P. (1995). *Teaching thinking through effective questioning* (2nd Ed.). Boston: Christopher-Gordon Publishers.
- Hutcheson, G., & Sofroniou, N. (1999). *The multivariate social scientist: Introductory statistics using generalized linear models*. Sage Publications.
- ISO. (2001). *Software Engineering - Product Quality-Part 1 Quality model Geneva: International Organization for Standardization*.
- Izu, C., Weerasinghe, A., & Pop, C. (2016). A study of code design skills in novice programmers using the SOLO taxonomy. In *Proceedings of the 16th International*

- workshop on Computing Education Research (ICER'16)* (pp. 251–259). Melbourne, Australia: ACM.
- Jakoš, F., & Lokar, M. (2015). A language independent assessment of programming concepts knowledge. In *Proceedings of International Conference on Informatics in Schools: Situation, Evolution and Perspectives (ISSEP'15)* (pp. 13–20). Münster, Germany: Springer-Verlag.
- Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition* (pp. 255–283). Hillsdale NJ: Lawrence Erlbaum.
- Johnson, B. E. (2011). The speed and accuracy of voice recognition software-assisted transcription versus the listen-and-type method: A research note. *Qualitative Research, 11*(1), 91–97.
- Johnson, C. G., & Fuller, U. (2006). Is Bloom's taxonomy appropriate for computer science? In *Proceedings of the 6th Baltic Sea conference on Computing Education Research (Baltic Sea'06)* (pp. 120–123). Koli National Park, Finland: ACM.
- Joppe, M. (2014). The research process. Retrieved February 20, 2017, from <http://www.htm.uoguelph.ca/MJResearch/ResearchProcess/home.html>
- Jordan, B., & Henderson, A. (2015). Interaction analysis : Foundations and practice. *Journal of the Learning Sciences, 4*(1), 39–103.
- Kaiser, Henry, F. (1974). An index of factorial simplicity. *Psychometrika, 39*(1), 31–36.
- Kaiser, H. F. (1960). The application of electronic computers to factor analysis. *Educational and Psychological Measurement, 20*(1), 141–151.
- Kaner, C., Member, S., & Bond, W. P. (2004). Software engineering metrics : What do they measure and how do we know? In *Proceedings of the 10th International Software Metrics Symposium (METRICS'04)* (pp. 1–12). IEEE Computer Society.
- Kasto, N., & Whalley, J. (2013). Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the 15th Australasian Computer Education Conference (ACE'13)* (Vol. 136, pp. 59–65). Adelaide, South Australia: Australian Computer Society Inc.
- Kester, L., Paas, F., & Van Merriënboer, J. (2010). Instructional control of cognitive load in the design of complex learning environments. In *Cognitive Load Theory* (pp. 109–130).
- Kintsch, W., Teun, A., & Van, D. (1978). Towards a model of text comprehension and production. *Psychological Review, 85*(5), 363–394.
- Klemola, T. (1978). *Software comprehension: theory and metrics*. Concordia University, Montreal, Canada.
- Klemola, T., & Rilling, J. (2003). A cognitive complexity metric based on category learning. In *Proceedings of the 2nd IEEE International Conference on Cognitive Informatics (ICCI'03)* (pp. 160–165). Bangkok: IEEE Computer Society.
- Kölling, M. (1999). Teaching object orientation with the Blue environment. *Journal of Object-Oriented Programming, 12*(2), 14–23.
- Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. In *Proceedings of the 6th conference on Innovation and Technology in Computer Science Education (ITiCSE'01)* (pp. 1–4). Canterbury, United Kingdom: ACM.
- Kuhn, D. (2008). Formal operations from a twenty-first century perspective. *Human*

Development, 51, 48–55.

- Kuittinen, M., & Sajaniemi, J. (2004). Teaching roles of variables in elementary programming courses. In *Proceedings of the 9th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE'04)* (pp. 57–61). Leeds, United Kingdom: ACM.
- Kumar, A. N. (2013). A study of the influence of code-tracing problems on code-writing skills. In *Proceedings of the 18th conference on Innovation Technology in Computer Science Education (ITiCSE'13)* (pp. 183–188). Canterbury, United Kingdom: ACM.
- Kurtz, B. (1980). Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class. *SIGCSE Bull.*, 12(1), 110–117.
- Lahtinen, E. (2007). A Categorization of novice programmers : A cluster analysis study. In *Proceedings of the 19th annual Workshop of the Psychology of Programming Interest Group (PPIG'07)* (pp. 32–41). Joensuu, Finland: University of Joensuu Department of Computer Science and Statistics, Joensuu, Finland.
- Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4), 325–339.
- Letovsky, S., & Soloway, E. (1986). Delocalized plans and program comprehension. *IEEE Software*, 3, 41–49.
- Li, X., & Atkins, M. (2004). Early childhood computer experience and cognitive and motor development. *Journal of Education Computing Reserach*, 113(6), 1–8.
- Lifelong Kindergarten Group. (2007). Creating with Scratch. Retrieved November 16, 2015, from [http://llk.media.mit.edu/projects/scratch/papers/Creating-with Scratch1 .pdf](http://llk.media.mit.edu/projects/scratch/papers/Creating-with_Scratch1.pdf)
- Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry*. SAGE Publications.
- Lister, R. (2011). Concrete and other Neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the 13th Australasian Computing Education Conference (ACE'11)*. Perth, Australia: Australian Computer Society, Inc.
- Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., ... Thomas, L. (2004). A Multi-National study of reading and tracing skills in novice programmers. In *Working group reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR'04)* (pp. 119–150). Leeds, United Kingdom: ACM.
- Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., ... Thompson, E. (2009). Naturally occurring data as research instrument : Analyzing examination responses to study the novice programmer. *SIGCSE Bull.*, 41(4), 156–173.
- Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 9th conference on Innovation and Technology in Computer Science Education (ITiCSE'09)* (Vol. 41, pp. 161–165). Leeds, United Kingdom: ACM.
- Lister, R., & Leaney, J. (2007). First year Programming : Let all the flowers bloom. In *Proceedings of the 5th Australasian Computing Education Conference - Volume 20 (ACE'03)* (pp. 221–230).
- Lister, R., Schulte, C., Whalley, J. L., Berglund, A., Clear, T., Bergin, J., ... Sanders, K. (2006). Research perspectives on the objects-early debate. *SIGCSE Bull.*, 34(4), 146–165.

- Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees : Novice programmers and the SOLO taxonomy. In *Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITICSE'06)* (pp. 118–122). Bologna, Italy: ACM.
- Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the 14th International workshop on Computing Education Research (ICER'08)* (pp. 101–112). Sydney, Australia: ACM.
- MacCallum, R. C., Widaman, K. F., Zhang, S., & Hong, S. (1999). Sample size in factor analysis. *Psychological Methods*, 4(1), 84–99.
- MacLean, L., Meyer, M., & Estable, A. (2004). Improving accuracy of transcripts in qualitative research. *Qualitative Health Research*, 14(1), 113–123.
- Magel, K. (1981). Regular expressions in a program complexity metric. *SIGPLAN Not.*, 16(7), 61–65.
- Marshall, C., & Rossman, G. B. (2006). *Designing qualitative research*. SAGE Publications.
- Marzano, R. (2000). Designing effective projects: Thinking skills frameworks Marzano's new taxonomy. Retrieved November 18, 2015, from <http://www.intel.com/content/dam/www/program/education/apac/ph/en/documents/project/design/marzo.pdf>
- Mathias, K. S., J.H, C., Hendrix, T. D., & Barowski, L. A. (1999). The role of software measures and metrics in studies of program comprehension. In *Proceedings of the 37th annual Southeast regional conference (CD-ROM) (ACM-SE 37)*. ACM.
- Mayer, R. E. (1977). The sequencing of instruction and the concept of assimilation-to-schema. *Instructional Science*, 6(4), 369–388.
- Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? The case for guided methods of instruction. *American Psychologist*, 59(1), 14–9.
- Mayrhauser, A. V., & Vans, A. M. (1998). Program understanding behavior during adaptation of large scale software. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98)* (pp. 164–172). Ischia: IEEE.
- McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308–320.
- McCracken, M., Almstrum, V., Diaz, D., Thomas, L., Guzdial, M., Utting, I., & Hagan, D. (2001). A multi-national , multi-institutional study of assessment of programming skills of first-year CS students A framework for first-year learning objectives. *SIGCSE Bull.*, 33(4), 125–140.
- McIver, L., & Conway, D. (1996). Seven deadly sins of introductory programming language design. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SEEP'96)* (pp. 309–316). Washington, DC, USA: IEEE Computer Society.
- McLaughlin G. H. (1969). SMOG grading: A new readability formula. *Journal of Reading*, 12(8), 639–646.
- Mcnaughton, S., & Leyland, J. (1990). The shifting focus of maternal tutoring across different difficulty levels on a problem-solving task. *British Journal of Developmental Psychology*, 8, 147–155.

- Meerbaum-Salant, O., Armoni, M., & Ben-Ari. (2013). Learning computer science concepts with Scratch. In *Proceedings of the 6th International workshop on Computing Education Research (ICER'10)* (pp. 69–76). Aarhus, Denmark: ACM.
- Mertens, D. M. (2005). *Research and evaluation in education and psychology: Integrating diversity with quantitative, qualitative, and mixed methods* (2nd Ed.). SAGE Publications.
- Miller, G. a. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63(2), 81–97.
- Moreno, R., & Park, B. (2010). Cognitive Load Theory-Roxana. In J. L. Plass, R. Moreno, & R. Brünken (Eds.), *Cognitive Load Theory* (pp. 9–28).
- Muller, O. (2005). Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the 5th International workshop on Computing Education Research (ICER'05)* (pp. 57–67). Seattle, WA, USA: ACM.
- Murphy, L., Fitzgerald, S., Lister, R., & McCauley, R. (2012). Ability to “Explain in Plain English” linked to proficiency in computer-based programming. In *Proceedings of the 9th International workshop on Computing Education Research (ICER'12)* (pp. 111–118). Auckland, New Zealand: ACM.
- Newman, I., & Benz, C. R. (1998). *Qualitative-quantitative research methodology: Exploring the interactive continuum*. Carbondale: Southern Illinois University Press.
- Ojose, B. (2008). Applying Piaget ’s theory of cognitive development to mathematics instruction. *The Mathematics Educator*, 18(1), 26–30.
- Oliver, D., Dobele, T., Greber, M., & Roberts, T. (2004). This course has a Bloom rating of 3.9. In *Proceedings of the 6th Australasian Computing Education Conference- Volume 30 (ACE'04)* (pp. 227–231). Dunedin, NZ: Australian Computer Society, Inc.
- Olsen, M. E., Lodwick, D. G., & Dunlop, R. E. (1992). *Viewing the world Ecologically Boulder*. Boulder, CO: Westview Press.
- Olson, G. . M., Duffy, S. A., & Mack, R. L. (1984). Thinking-out-loud as a method for studying real-time comprehension processes. In M. A. Kieras & M. A. Just (Eds.), *New methods in reading comprehension research* (pp. 253–286). Hillsdale, NJ: Erlbaum.
- Olson, G., Catrambone, R., & Soloway, E. (1987). Programming and Algebra word problems: A failure to transfer. In G. Olson, S. Sheppard, & E. Soio-Way (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp. 1–13). Norwood, N.J.: Ablex.
- Onwuegbuzie, A. J., & Leech, N. L. (2005). On becoming a pragmatic researcher: The importance of combining quantitative and qualitative research methodologies. *International Journal of Social Research Methodology*, 8(5), 375–387.
- Paas, F. G. W. C., & Van Merriënboer, J. J. G. (1994). Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach. *Journal of Educational Psychology*, 86, 122–133.
- Palumbo, D. B. (1990). Programming language/Problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1), 65–89.
- Paperblanks. (2012). Paper vs. Computers: Which is better to write with? Retrieved December 2, 2015, from <http://blog.paperblanks.com/2012/04/better-to-write-with-journals-or-computers>

- Parker, J. R., & Becker, K. (2003). Measuring effectiveness of constructivist and behaviourist assignments in CS102. In *Proceedings of the 8th conference on Innovation and Technology in Computer Science Education (ITiCSE'03)* (pp. 40–44). Thessaloniki, Greece: ACM.
- Pattis, R. E. (1981). *Karel the robot: A gentle introduction to the art of programming. Paperback* (2nd Ed.). New York, NY, USA: John Wiley & Sons, Inc.
- Patton, M. (1990a). Qualitative evaluation and research methods. In *Designing qualitative studies* (pp. 169–186). Beverly Hills, CA: Sage.
- Patton, M. (1990b). Qualitative evaluation and research methods. In *Designing qualitative studies* (pp. 169–186). SAGE Publications.
- Pea, R. D. (2013). The social and technological dimensions of scaffolding and related theoretical concepts for learning, education, and human activity. *Journal of the Learning Sciences, 13*(3), 423–451.
- Pegg, J., & Tall, D. (2002). Fundamental cycles in learning Algebra : An analysis. In *Proceedings of the 26th conference of the International Group for the Psychology of Mathematics Education*. Norwich, UK: University of East Anglia, School of Education and Professional Development.
- Pennington, N. (1987a). Comprehension strategies in programming. In *Empirical studies of programmers: second workshop* (pp. 100–113). Norwood, NJ, USA: Ablex Publishing Corp.
- Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*(3), 295–341.
- Perkins, D. N., Hancock, C., Hobbs, M., Fay, S., & Rebecca. (1989). *Conditions of learning in novice programmers*. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum.
- Perkins, D. N., & Martin, F. (1985). *Fragile knowledge and neglected strategies in novice programmers*. In Soloway and S. Lyengar (Eds.), *Empirical studies of programmers*. Hillsdale, NJ: Ablex.
- Perkins, D. N., & Salomon, G. (1994). Transfer of learning. In *International Encyclopaedia of Education, Oxford: Elsevier* (pp. 6452–6457).
- Petersen, A., Craig, M., & Zingaro, D. (2011). Reviewing CS1 exam question content. In *Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE'11)* (pp. 631–636). Dallas, TX, USA: ACM.
- Philpott, A., Robbins, P., & Whalley, J. (2007). Assessing the steps on the road to relational thinking. In *Poster paper in the 20th Annual NACCCQ*. Nelson NZ.
- Piaget, J. (1970). *Genetic epistemology*. W. W. Norton, New York.
- Piaget, J., & Inhelder, B. (1969). *The psychology of the child*. London, UK: Routledge & Kegan Paul.
- Pikulski, J. K. (2002). Readability. Retrieved November 18, 2015, from <http://www.eduplace.com/state/author/pikulski.pdf>.
- Pirolli, P. (1986). A Cognitive model and computer tutor for programming recursion. *Human – Computer Interaction, 2*(4), 319–355.
- Pirolli, P., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue Canadienne de Psychologie, 39*(2), 240–272.

- Piwowski, P. (1982). A nesting level complexity measure. *SIGPLAN*, 17(9), 44–50.
- PMD. (2016). PMD. Retrieved February 17, 2016, from <https://pmd.github.io/>
- Pohl, M. (2000). *Learning to think, thinking to learn: models and strategies to develop a classroom culture of thinking*. Cheltenham, Vic.: Hawker Brownlow.
- Ragonis, N., & Ben-Ari, M. (2005). On understanding the statics and dynamics of object-oriented programs. In *Proceedings of the 36th SIGCSE technical symposium on Computer Science Education (SIGCSE'05)* (pp. 226–230). St. Louis, Missouri, USA: ACM.
- Rationale Software Analyzer tool. (2016). Rationale® Software Analyzer tool. Retrieved February 17, 2016, from http://www.ibm.com/developerworks/rational/library/08/0429_gutz1/
- Reeves, T., & Hadberg. (2003). *Interactive learning system evaluation*. Educational technology publication Inc. Englewood Cliffs, New Jersey.
- Reges, S. (2006). Back to basics in CS1 and CS2. In *Proceedings of the 37th SIGCSE technical symposium on Computer Science Education (SIGCSE'06)* (pp. 293–297). Houston, Texas, USA: ACM.
- Reiser, B. J. (2002). Why scaffolding should sometimes make tasks more difficult for learners. In *Proceedings of the Conference on Computer Support for Collaborative Learning Foundations for a CSCL Community (CSCL'02)* (p. 255). Morristown, NJ, USA: Association for Computational Linguistics.
- Rilling, J., & Klemola, T. (2003). Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*. Portland, Oregon, USA: IEEE Computer Society.
- Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, 13, 389–414.
- Rist, R. S. (1991). Knowledge creation and retrieval in program design : A comparison of novice and intermediate student programmers. *Human – Computer Interaction*, 6, 1–46.
- Robins, A. (2010). Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, 20(1), 37–71.
- Rogoff, B. (1984). Introduction: Thinking and learning in social context. In B. Rogoff & J. Lave (Eds.), *Everyday Cognition: Its Development in Social Contexts*. Cambridge, MA: Harvard University Press.
- Role of variables. (2015). Fundamentals of programming: The role of variables. Retrieved March 20, 2015, from https://en.wikibooks.org/wiki/A-level_Computing/AQA/Problem_Solving,_Programming,_Data_Representation_and_Practical_Exercise/Fundamentals_of_Programming/The_Role_of_Variables
- Rossmann, G., & Wilson, B. L. (1985). Numbers and words: Combining quantitative and qualitative methods in a single large-scale evaluation study. *Evaluation Review*, 9.
- Rowland, G. (1992). What do instructional designers actually do? An initial investigation of expert practice. *Performance Improvement Quarterly*, 5(2), 65–86.
- Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computer Languages and Environments*. IEEE Computer Society (pp. 37–39). Virginia, USA.

- Sakhnini, V., & Hazzan, O. (2008). Reducing abstraction in high school computer science education : The case of definition , implementation , and use of abstract data types. *Journal on Education Resource Computing*, 8(2), 1–13.
- Salomon, G., & Perkins, D. (1989). Rocky roads to transfer: Rethinking mechanism of a neglected phenomenon. *Educational Psychologist*, 24(2), 113–142.
- Samurçay, R. (1989). *The concept of variable in programming: Its meaning and use in problem- solving by novice programmers*. In E. Soloway, & J. C. Spohrer (Eds.), *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Scholtz, J., & Wiedenbeck, S. (1990). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, 2(1), 51–72.
- Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., & Paterson, J. H. (2010). An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports (ITiCSE-WGR'10)* (pp. 65–86). Bilkent, Ankara, Turkey: ACM.
- Schunk, D. H. (2012). *Learning theories - An educational prespective* (6th Ed.). Boston : Pearson.
- Seiter, L. (2015). Using SOLO to classify the programming responses of primary grade students. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)* (pp. 540–545). Kansas City, Missouri, USA: ACM.
- Sfard, A. (1991). On the dual nature of mathematical conceptions: reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, 22(1), 1–36.
- Sfard, A. (1992). Operational origins of mathematical objects and the quandary of reification-the case of function. In G. Harel & E. Dubinsky (Eds.), *The concept of function: Aspects of epistemology and pedagogy* (pp. 59–84). Washington DC: Mathematical Association of America.
- Sfard, A. (1998). On two metaphors for learning and the dangers of choosing just one. *Educational Researcher*, 27(2), 4–13.
- Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., Grove, H., ... Whalley, J. (2008). Going SOLO to assess novice programmers. *SIGCSE Bull.*, 40(3), 209–213.
- Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering*, 3(2), 30–36.
- Shuhidan, S., Hamilton, M., & Souza, D. D. (2009). A Taxonomic study of novice programming summative assessment. In *Proceedings of the 11th Australasian Computing Education Conference - Volume 95 (ACE'09)* (pp. 147–156). Darlinghurst, Australia: Australian Computer Society, Inc.
- Simon, H. A. (1979). Problem solving and education. In D. Tuma & F. Reif (Eds.), *Problem solving and education: Issues in teaching and research*. Hillsdale, NJ: Erlbaum.
- Simon, H. A., & Hayes, J. R. (1976). The understanding process : problem isomorphs. *Cognitive Psychology*, 8(2), 165–190.
- Simon, Lopez, M., Sutton, K., & Clear, T. (2009). Surely we must learn to read before we learn to write ! In *Proceedings of the 11th Australasian Computing Education Conference (ACE'09)* (pp. 165–170). Darlinghurst, Australia: Australian Computer Society, Inc.

- Simon, & Sheard, J. (2012). Exams in computer programming : what do they examine and how complex are they? In *Proceedings of the 14th Australasian Computing Education Conference (ACE'12)* (pp. 283–291). Melbourne, Australia: Australian Computer Society, Inc.
- Simon, Sheard, J., Carbone, A., Clear, T., Raadt, M. De, Souza, D. D., ... Warburton, G. (2012). Introductory programming : examining the exams. In *Proceedings of the 14th Australasian Computing Education Conference (ACE'12)* (pp. 61–70). Melbourne, Australia: Australian Computer Society, Inc.
- Skinner, B. F. (1953). *Science and human behavior*. New York: Macmillan.
- Soloway, E. (1986). Learning to program = Learning to construct mechanisms. *Communications of the ACM*, 29(9), 850–858.
- Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5), 595–609.
- Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1983). What do novices know about programming ? *Directions in Human--Computer Interactions*, 6(1), 27–54.
- Soloway, E., & Spohrer. (1989). *Studying the Novice Programmer*. Hillsdale, NJ, Lawrence Erlbaum Associates.
- Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7), 624–632.
- Spohrer, J., Soloway, E., & Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human – Computer Interaction*, 1, 163–207.
- Starsinic, K. (1998). Perl Style. *The Perl Journal, Fall 1998*, 3(3), 1998.
- Strauss, S. (1993). Theories of learning and development for academics and educators. *Educational Psychologist*, 28(3), 191–203.
- Sudol-Delyser, L. A. (2015). Expressions of abstraction: Self-Explanation in code production. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)* (pp. 272–277). Kansas City, Missouri, USA: ACM.
- Sweller, J. (1994). Cognitive Load Theory, learning difficulty, and instructional design. *Learning and Instruction*, 4, 295–312.
- Sweller, J., & Chandler, P. (1994). Why some material is difficult to learn. *Cognition and Instruction*, 12(3), 185–233.
- Teague, D., & Lister, R. (2014a). Longitudinal think aloud study of a novice programmer. In D. D'Souza & J. Whalley (Eds.), *Proceedings of the 16th Australasian Computing Education Conference - Volume 148 (ACE '14)* (pp. 41–50). Auckland, New Zealand: Australian Computer Society, Inc.
- Teague, D., & Lister, R. (2014b). Manifestations of preoperational reasoning on similar programming tasks. In *Proceedings of the 16th Australasian Computer Education Conference (ACE'14)* (Vol. 148, pp. 30–40). Auckland, New Zealand: Australian Computer Society, Inc.
- Teague, D., & Lister, R. (2014c). Programming: reading, writing and reversing. In *Proceedings of the 2014 conference on Innovation and Technology in Computer Science Education (ITiCSE'14)* (pp. 285–290). Uppsala, Sweden: ACM.
- Thomas, L., Ratcliffe, M., & Thomasson, B. (2004). Scaffolding with object diagrams in first year programming classes. In *Proceedings of the 35th SIGCSE technical symposium on Computer Science Education (SIGCSE'04)* (pp. 250–254). Norfolk,

Virginia, USA: ACM.

- Thompson, E. (2008). *How do they understand? Practitioner perceptions of an object-oriented program*. PhD thesis Massey University, New Zealand.
- Thompson, E. (2010). Using the principles of variation to create code writing problem sets. In *Proceedings of the 11th conference of the Higher Education Academy - Information and Computer Sciences* (pp. 11–16). Durham University: HEA ICS.
- Thompson, E., Grove, H., Luxton-reilly, A., Whalley, J., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. In *Proceedings of the 10th Australasian Computer Education Conference - Volume 78 (ACE'08)* (Vol. 78, pp. 155–161). Wollongong, Australia: Australian Computer Society, Inc.
- Thorndike, E. L. (1923). The influence of First-Year Latin upon ability to read English. *School & Society*, *17*, 165–168.
- Trafton, J. ., & Reiser, B. (1993). Studying examples and solving problems: Contributions to skill acquisition. In *Proceedings of the 15th conference of the Cognitive Science Society* (pp. 1017–1022). Colorado: Lawrence Erlbaum Associates.
- Van Merriënboer, J. G. (1990). Strategies for programming instruction in high school: program completion vs. program generation. *Journal of Educational Computing Research*, *6*, 265–287.
- Van Merriënboer, J. G. (1997). *Training complex cognitive skills: A Four- Component instructional design model for technical training*. Englewood Cliffs, NJ: Educational Technology Publications.
- Van Merriënboer, J. G., & De Croock, M. (1992). Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research*, *8*, 365–394.
- Van Merriënboer, J. G., Kirschner, P. A., & Kester, L. (2003). Taking the load off a Learner's mind : instructional design for complex learning, *38*(1), 5–13.
- Van Merriënboer, J. G., & Krammer, H. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high. *Instructional Science*, *16*, 251–285.
- Van Merriënboer, J. G., & Paas, F. (1990). Automation and schema acquisition in learning elementary computer programming : Implications for the design of practice. *Computers in Human Behavior*, *6*, 273–289.
- Van Solingen, R., & Berghout, E. (1999). *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill Publishing Company.
- Van Someren, M. W., Barnard, Y. F., & Sandberg, J. A. C. (1994). *The think aloud method a practical guide to modelling cognitive processes*. Academic Press, London.
- Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the 5th International workshop on Computing Education Research (ICER'09)* (pp. 117–128). Berkeley, CA, USA: ACM.
- Vosniadou, S., & Ortony, A. (1989). Similarity and Analogical Reasoning: A Synthesis. In S. Vosniadou & A. Ortony (Eds.), *Similarity and Analogical Reasoning* (pp. 1–18). NY: Cambridge University Press.

- Vygotsky. (1978). *Interaction between learning and development. From: Mind and Society*. Cambridge, MA: Harvard University press.
- Vygotsky, L. (1981). *The genesis of higher mental functions*. In J. V. Wertsch (Ed.), *The concept of activity in Soviet psychology* (pp. 144–188). Armonk, NY: Sharpe (J. V. Wertsch, trans.).
- Vygotsky, L. (1986). *Thought and language*. Cambridge, MA: MIT (A. Kozulin, trans.).
- Werth, L. (1986). Predicting student performance in a beginning computer science class. In *Proceedings of the 17th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'86)* (pp. 138–143). Cincinnati, Ohio, USA: ACM.
- Whalley, J., Clear, T., Robbins, P., & Thompson, E. (2011). Salient elements in novice solutions to code writing problems. In *Proceedings of the 13th Australasian Computing Education Conference - Volume 114 (ACE'11)* (pp. 37–46). Darlinghurst, Australia: Australian Computer Society, Inc.
- Whalley, J., & Kasto, N. (2014). A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation and Technology in Computer Science Education (ITiCSE'14)* (pp. 279–284). Uppsala, Sweden: ACM.
- Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., & Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Computing Education Conference - Volume 52 (ACE '06)* (Vol. 52, pp. 243–252). Darlinghurst, Australia: Australian Computer Society, Inc.
- Whalley, J., & Philpott, A. (2011). A unit testing approach to building novice programmers' skills and confidence. In *Proceedings of the 13th Australasian Computing Education Conference - Volume 114 (ACE'11)* (pp. 113–118). Darlinghurst, Australia: Australian Computer Society, Inc.
- White, G. L., & Sivitanides, M. P. (2002). A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics. *Journal of Information Systems Education*, 13(1), 59–66.
- Wille, A. M. (2010). Steps towards a structural conception of the notation of variables. In *Proceedings of Congress of the European Society for Research in Mathematics Education (CERME) 6* (pp. 659–668).
- Winslow, L. E. (1996). Programming pedagogy-a psychological overview. *ACM SIGCSE Bull.*, 28(3), 17–22.
- Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, 17(2), 89–100.
- Wu, Q., & Anderson, J. R. (1990). *Problem-solving transfer among programming languages*. Pittsburgh, U.S.A.
- Xinogalos, S. (2010). An interactive learning environment for teaching the imperative and Object-Oriented Programming. In *Knowledge management, information systems, E-Learning, and sustainability research* (pp. 512–520). Springer.
- Yamamoto, M., Sekiya, T., Mori, K., & Yamaguchi, K. (2012). Skill hierarchy revised by SEM and additional skills. In *proceedings of the International Conference on Information Technology Based Higher Education and Training (ITHET'12)* (pp. 1–8). Istanbul: IEEE.
- Yousoof, M., & Sapiyan, M. (2015). Optimizing instruction for learning computer

programming – A novel approach. In *Intelligence in the Era of big data* (Vol. 516, pp. 128–139).

Glossary

Term	Explanation / Definition	
Course (paper)	A unit of teaching that typically lasts one academic term.	
Décalage	A French term meaning a shift, or gap.	
Pattern	Is a recurring schema or plan which is used so often that it becomes a generalised or abstract notion which can be applied to different problems.	
Plan	A set of steps used to solve a programming task. Typically a plan will consist of more than one schema.	
Salient element	This term used was used by Whalley et al. (2011) . Syntactic elements in novice code. For example FOR-loops, IF-statement or variable declaration these are the simple elements which combined form code patterns and schemas.	
Schema	Existing mental structures in long term memory. They represent an organisation and linking of knowledge.	
Strategies for gluing programming plans together (adapted from Soloway, 1986, p.856).	Abutment	Two plans are glued together back to front, in sequence.
	Merging	At least two plans are interleaved
	Nesting	One plan is completely surrounded by another plan.
	Tailoring	Sometimes recalled a programming plan that has already been developed is not quite what is needed in a problem. It must be modified to fit the particular needs of the situation.
The role of variables (adapted from Role of variables, 2015).	Follower variable	Used to keep track of a previous value of a variables, so that a new value can be compared.
	Gatherer variable	A variable that accumulates or tallies up set of data and inputs. It is very useful for calculating totals or totals that will be used to calculate averages.
	Most wanted holder variable	A variable that keeps track of the lowest or highest value in a set of inputs.
	Stepper variable	A variable used to move through an array or other data structure, often heading towards a fixed value and stepping through elements in an array.

Appendix A. Think Aloud Data

Andre: Counting the Number of Beepers in a Single Corridor (Seq2 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Familiar first	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging
	<i>Unit test</i>	
	<i>Time on task</i>	9 minutes and 2 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	Clarify scaffolding – provided on request	
Timing	Week four of the P1 course	
Important observations with respect to prior sessions	Andre solved this question after solving counting the length of corridor (Seq1 – Q1, Chapter 6). For solving Seq1 – Q1, Andre did not read the unit test messages to fix his code.	

Data

1. Think aloud:

After reading the question, Andre verbalised:

Andre: *“Can I run the test first to see the position”*

Interviewer: *“Yes”*

As he completed the above utterance, Andre run the first unit test (Robot World scenario with length 5), then he started writing his code while verbalising (see Figure A.1, step1 (A&B)): *“While item on the ground pickup item, we just [pause], no the first location is not, just set the WHILE loop”*

Then Andre asked for help – *“I need your help”* – he needed the task requirements clarified.

2. Scaffolding:

Andre: *“Is more than one beeper at each locations”*

Interviewer: *“More than one”*

3. Think aloud:

Andre verbalised: *“So I need to define variable to store number of beepers”*

After the above utterance, Andre defined the gatherer variable and set it value to zero, followed by Java command to increment the gatherer variable by one (see Figure A.1, steps2&3). After a long pause, Andre added a WHILE-statement followed by a robot method call that allowed the robot to move forward one step (see Figure A.1, steps4&5). Then Andre verbalised: *“Let me check a bit, after counting beepers, I need forwards, [pause] I need to print”*.

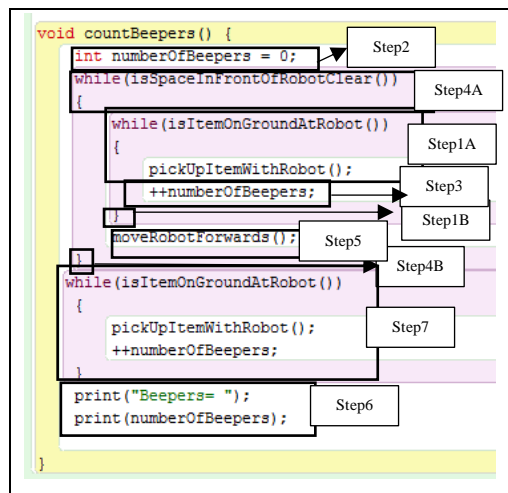


Figure A.1 Andre’s screen image for counting the number of beepers in a single corridor

After the above utterance, Andre added PRINT-statements as shown in Figure A.1, step6. On testing his code, Andre discovered that all tests failed. He lined up the Robot World windows so he could examine all the test results at the same time. Then he verbalised: “For the last one he cannot pick up item what can I do after he facing wall, I can figure out statement, so, ammm, I do not know, ah [pause], ah”.

Andre started to update his code as shown in Figure A.1, step7. Finally, Andre compiled and ran the supplied unit tests to verify the correctness of his solution.

Andre: Comparing the Length of Two Corridors (Seq1 – Q2)

Encoding

Question	Solved
Behaviours	Mover
Emotion	Indiscernible
Strategies	Sequential
Activities	<ul style="list-style-type: none"> <i>Planning</i> Verbalise <i>Tracing</i> Mental tracing <i>Unit test</i> <i>Time on task</i> 17 minutes and 24 seconds <i># of compilation</i> 1 <i># of execution</i> 1
Intervention	None
Timing	Week five of the P1 course
Important observations with respect to prior sessions	Andre solved the counting the length of the corridor (see Chapter 6) with the interviewer’s assistance

Data

1. Think aloud:

Andre began by reading the problem and then he verbalised his plan for solving the question before he attempted to write the code on the computer: “The corridor one it is long seven, and the robot should count both and check which the largest, first is, we need to declare two variables like”.

Andre started with initialising two gatherer variables one for each corridor, and set both of them to zero, followed by a WHILE-loop block for moving the robot and counting the length of the first corridor before closing the WHILE-loop block bracket. Andre started to read his code and

verbalise: *“The number now the length of corridor should be, should be the ah[pause] should be [pause] I should add one to the length because the initial location should be counted, as well, and so, ah [pause] and now let me check it [pause] plus, plus, then the corridor”*.

Therefore, at the end of the while block, Andre closed the bracket and added the Java command that increased the gatherer variable for the first corridor by one. After that, Andre multitasked – viewing the image of the robot on the paper and writing a sequence of commands. The functions of these commands were returning the robot to its starting position, changing the robot’s orientation to face north, moving the robot forwards two steps to the next corridor, and finally changing the robot’s orientation to face east. He then copied and pasted the Java commands for counting the length of the first corridor and renamed the gatherer variable to count the length of the second corridor. Finally, Andre used three IF-blocks to compare the lengths of two corridors. Again, Andre started to read his code for the second time and verbalise: *“Let me check it, first declare two integer values, length of integer corridor zero and length of integer corridor one, zero is the lower and one is upper corridor, first I will check for corridor zero, while space in front of robot clear, move robot forwards, and add one to the counter one, but the final value of the length, ah let me see, should be let me see, I have got typo”*.

Andre fixed the typo error but he lost his thought series therefore he decided to add comments and start reading his code again and verbalise: *“Check it again length of corridor one, length of corridor two, the final value of the corridor one should be the final value plus one, count the first location, and robot left, turn round, go ahead, and turn left, move forwards four time, no two times, and check the corridor one, while is space in front of, move and count just the same, as the corridor zero, finally compare two corridors, if length of corridor zero less than length of corridor one, corridor one is the longest, it is the length, if length of corridor zero is larger than length of corridor one, corridor zero is the longest, if they are equal”*.

Andre ran the supplied unit tests and all tests passed from the first attempt.

2. Retrospection:

Interviewer: *“Have you seen this question before?”*

Andre: *“No”*

Interviewer: *“What was the most difficult part for solving this question?”*

Andre: *“The most difficult part was more steps to go back and check the others and comparing the length of corridor “*

Interviewer: *“Why didn’t you use the nested IF-ELSE block for solving this question?”*

Andre: *“How I could do it?”*

Andre: Counting the Number of Beepers in each Beeper Stack across a Single Corridor (Seq2 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	9 minutes and 15 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week five of the P1 course	
Important observations with respect to prior sessions	Andre solved counting the number of beepers in a single corridor with the interviewer's assistance (Seq2 – Q2).	

Data

Think aloud:

Andre immediately started to verbalise and write his solution as shown Figure A.2, steps 1→8.

"I need to declare variable. I need to check the first location, after that to set the variable to zero [pause]. I need to check the other locations, ammm, so I need WHILE loop then I need copy paste counting the first the robot should facing wall. The question still need to print out the number of beepers".

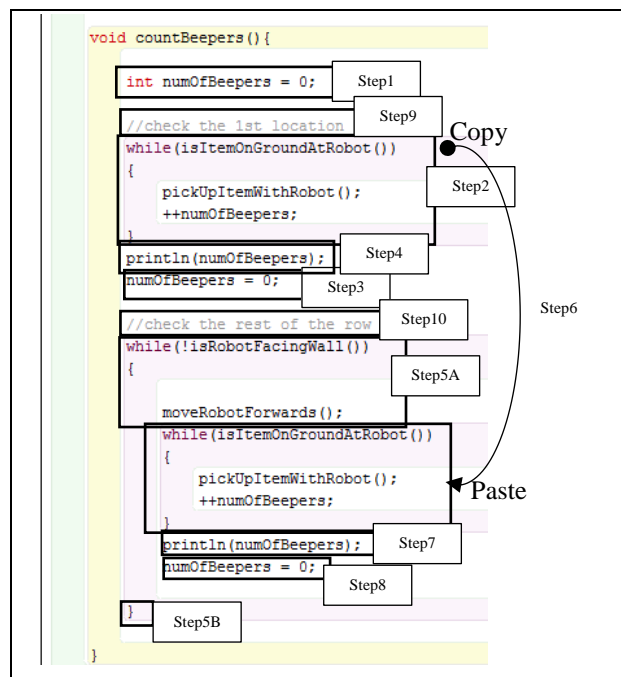


Figure A.2 Andre's screen image for counting the number of beepers in each beeper across in a single corridor

After finishing writing his code, he started to add comments while reading his code (Figure A.2, steps 9 & 10): *"Count equal zero, while is item, pick up and add, pick up and add, print, set counter to zero. While not facing wall. While is item pick up and add, print, set counter to zero".*

He then ran the supplied unit tests and all tests passed from first trial.

Andre: Smallest Stack of Beepers (Seq2 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging, mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	24 minutes and 49 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	1
Intervention	Clarify scaffolding – provided on request	
Timing	Week six of the P1 course	
Important observations with respect to prior sessions	Andre previously solved a far transfer question (the longest corridor, Chapter 6). The code quality of his solution for the longest corridor question may have hindered Andre’s ability to transfer knowledge. Andre solved counting the number of beepers in a single corridor with the interviewer’s assistance (Seq2 – Q2).	

Data

1. Think aloud:

Andre began by reading the problem. This was followed by the method definition and then he verbalised: *“Counting the number of beepers in each stack so the image show below that may be there are no beepers, so each, to compare find the smallest, ah, just from previous question ah, if it is bigger than than that, if is smaller that, ah why, but if it is smaller than that, if there is no beeper, so I need to [pause] let me see, firstly the return value should be assign, then I will set smallest stack equal to zero”*.

After that, Andre defined the most wanted holder variable `smallestStack` and set its value to zero, followed by retrieving the schemas for picking up and counting all the beepers in the current stack (see Figure A.3 (steps1→step5)(left)). Then he verbalised: *“Then the current stack will be equal to the first stack, so the smallest stack, so if ah this is of course smallest stack is zero , we can written no no, if there is zero and the current stack is bigger so now the smallest stack is zero, but if we know that it is no zero, ah, no it is already ,if we always compare with that, with the smallest is zero so it will not work, so is like we need to, it is different from longest corridor [pause], longest is comparing ah corridors, no corridor its length zero, so the first one is to assign the value to the smallest stack to ah yes, so, ah, so may be the first one is not the smallest stack, so we can assign the current stack [pause], let us assume the smallest one but we need to compare with it , ah”*.

Andre’s above verbalisation indicates that he started to struggle again (i.e. how to compare – same problem for solving longest corridor). Therefore he decided to continue with the program plans that allowed the robot to move and pick up beepers if they existed in a corridor (as shown in Figure A.3 (step6) (left)). Andre asked for help (clarify scaffold) at this stage.

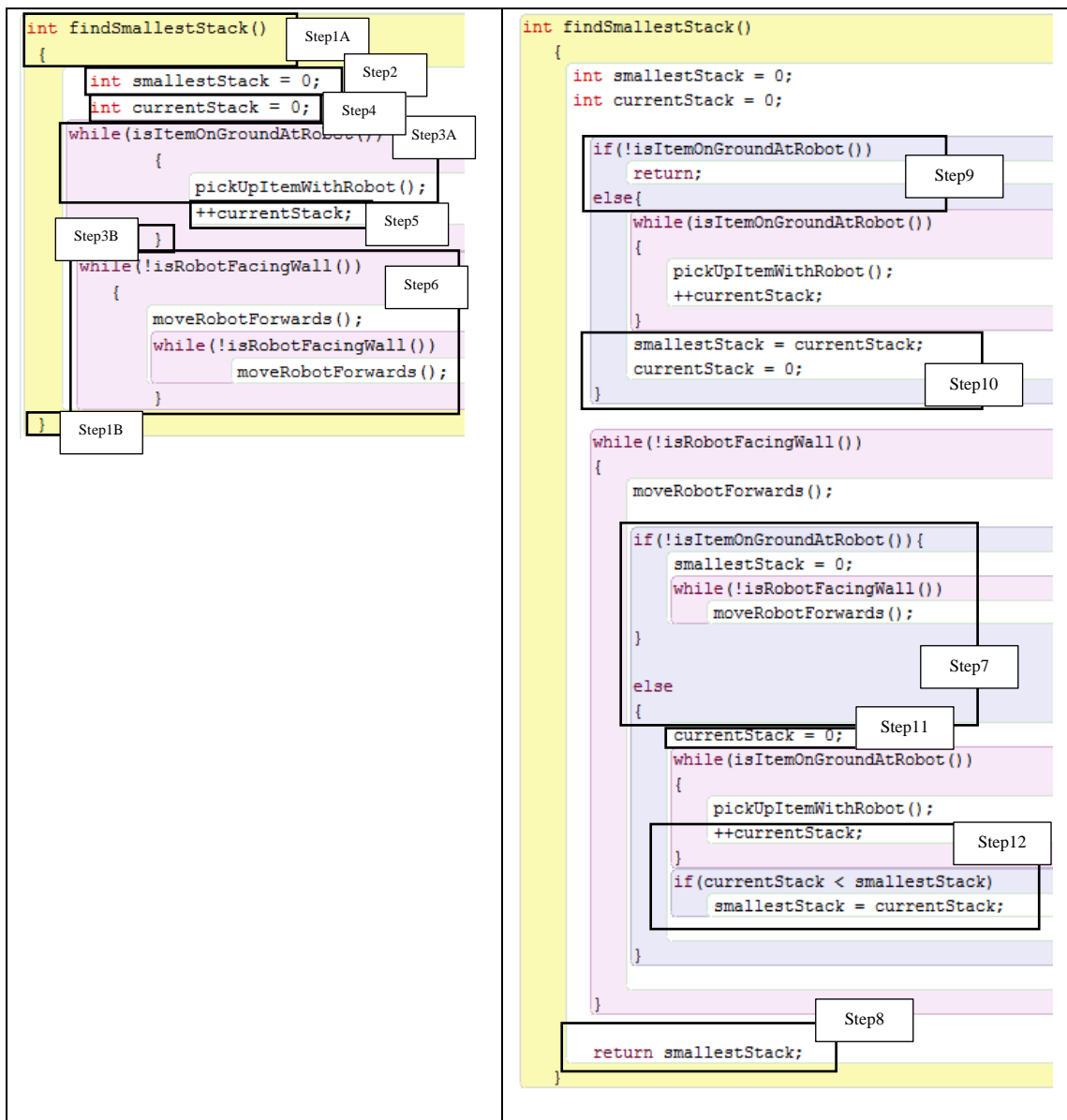


Figure A.3 Andre's first and second screen images for the smallest stack of beepers

2. Scaffolding:

Andre: "I have question, the smallest stack of the beeper, is like there is no beeper no beeper on the ground"

Interviewer: "If there are no beepers at any of the locations, then the smallest will be zero, otherwise, you need to decide what the smallest number of beepers in the stack should be"

3. Think aloud:

Based on the conversation between Andre and the interviewer, Andre immediately, started to update his solution (as shown in Figure A.3 (right)) with verbalising his writing. Andre's fragile knowledge about the differences between iteration and section Java commands (as shown in Figure A.3 (right)). And then he started to trace his code and verbalised: "The current stack is zero, after counting the number of beepers in the first stack, the current stack will be let us say five, so the smallest will be five and current will be zero [pause], counting the second location, let us say the current will be six, and five less than six so current will still have five, but if the current

is zero so smallest stack will be zero and the robot will move forwards forwards until the end of corridor”.

Andre compiled his code and he easily fixed the syntax error (see Figure A.3 (step9) (right) Return without variable). Finally, Andre ran the supplied unit test and all tests passed from the first trial.

4. Retrospection:

Interviewer: “Have you seen this question before?”

Andre: “No, at the beginning I thought it is similar to counting the length of corridor, but the problem um, if there is no beepers”

Interviewer: “That means you recall the program steps for counting the beepers at each location?”

Andre: “Yes, yes, I solve heap of question counting the number of beepers but the problem was if there is no beepers”

At the end of the session, the interviewer gave Andre a feedback about the quality of his code and how he could further develop it.

Andre: Checking if Beeper Stacks are Sorted in Ascending Order by Size of the Stack (Seq2 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Happy	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	Read the code for the test
	<i>Time on task</i>	15 minutes and 48 seconds
	<i># of compilation</i>	3
	<i># of execution</i>	2
Intervention	None	
Timing	Week eleven of the P1 course	
Important observations with respect to prior sessions	In previous meeting sessions, Andre found no difficulty in using his existing schemas for counting beeper and checking whether or not the one-dimensional array was sorted.	

Data

Think aloud:

Andre began by reading the problem. This was followed by the method definition. Andre compiled his code for the first time and he easily fixed the syntax error by adding the RETURN statement. Then he re-compiled and ran the supplied unit tests to visualise robot moving across the corridor (two tests failed and one test passed). Andre started to open the unit test file code and verbalised: “For the first test, the location 0 have two, the first location have seven beepers, the second location have two beepers. So the method return false. For the second test, zero, one, two, three, four six, six. The result is true. For the last one, two, five, four, false”.

After the above utterance, Andre focused on recalling the programming plan for pick up all the beepers across a single corridor (see Figure A.4 (left)). Then Andre verbalised: “After pick up all the beepers from the corridor. Now we need to count each, if we need to compare, we need to count each”.

Therefore, Andre decided to define two gatherer variables `presentStack` and `lastStack` and set both of them to zero. Then he started to update his code as shown in the Figure A.4 (step1→step4) (right). After that Andre verbalised while continue writing his code Figure A.4 (step5) (right): “After that I need to compare. IF present stack [`presentStack`] less than last stack [`lastStack`] [pause], return false”.

```

boolean beeperStacksInSizeOrder()
{
    while (isItemOnGroundAtRobot())
    {
        pickUpItemWithRobot();
    }
    while (!isRobotFacingWall()) {
        moveRobotForwards();
        while (isItemOnGroundAtRobot())
        {
            pickUpItemWithRobot();
        }
    }
    return true;
}

boolean beeperStacksInSizeOrder()
{
    int presentStack = 0;
    int lastStack = 0;
    while (isItemOnGroundAtRobot())
    {
        pickUpItemWithRobot();
        lastStack++;
    }
    while (!isRobotFacingWall()) {
        moveRobotForwards();
        while (isItemOnGroundAtRobot())
        {
            pickUpItemWithRobot();
            presentStack++;
        }
    }
    if (presentStack < lastStack)
        return false;
    lastStack = presentStack;
    presentStack = 0;
    return true;
}

```

Figure A.4 Andre’s first and second screen images for beepers stack are in order

Finally, Andre re-ran the supplied unit tests and all tests passed.

Andre: Column in a 2D Which Contains a Smallest Number (Seq4 – Q3)

Encoding

Question	Solved
Behaviours	Mover
Emotion	Indiscernible
Strategies	Stepwise design
Activities	Verbalise
<i>Planning</i>	
<i>Tracing</i>	
<i>Unit test</i>	
<i>Time on task</i>	5 minutes and 37 seconds
<i># of compilation</i>	1
<i># of execution</i>	1
Intervention	None
Timing	Andre solved this question directly after solving (Seq4 – Q2).
Important observations with respect to prior sessions	Andre took 46 minutes and 13 seconds to solve the smallest element in a one-dimensional array (Seq3 – Q2), while he took less time (9 minutes and 16 seconds) to solve the largest element in a two-dimensional array (Seq4 – Q2, a far transfer problem).

Data

Think aloud:

Andre began by reading the problem and immediately started to plan his solution, and then he verbalised: “It is about find the smallest one, the difference is like the first one is the largest while this one is the smallest, and this one we need to return the index of the column of it, so ah let me see, okay first we need ah”.

After that, he wrote the method name and the array of type integer as a passing parameter. Then he added line by line Java commands as shown in Figure A.5 (left). After typing the word IF. Andre paused and verbalised: “So if ah smaller, should I keep another value if ah maybe”.

```

public int findSmallestIndexColumn(int[][] a1)
{
    int smallest = a1[0][0];
    for(int row = 0; row<a1.length; ++row)
    {
        for(int col = 0; col < a1[row].length; ++col)
        {
            if
        }
    }
}

public int findSmallestIndexColumn(int[][] a1)
{
    int smallest = a1[0][0];
    int smallCol = 0;
    for(int row = 0; row<a1.length; ++row)
    {
        for(int col = 0; col < a1[row].length; ++col)
        {
            if(a1[row][col] < smallest)
            {
                smallest = a1[row][col];
                smallCol = col;
            }
        }
    }
    return smallCol;
}

```

Figure A.5 Andre’s first and second screen images for the column of smallest number in a 2D

As a result, Andre defined a second most wanted holder variable after the first most wanted holder variable definition and without hesitating he set its value to zero, then he continued to type the IF-block and the RETURN statement as shown in Figure A.5 (right). Finally Andre ran the unit tests to verify the correctness of his solution.

Andre: Checking if Sorted Ascending Each Row of a 2D Array Elements (Seq4 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	Read test output and test code
	<i>Time on task</i>	13 minutes and 27seconds
	<i># of compilation</i>	3
	<i># of execution</i>	3
Intervention	None	
Timing	Week five of the P2 course	
Important observations with respect to prior sessions	Andre did not face any difficulties for solving (Seq3 – Q1, Chapter 6).	

Data

1. Think aloud:

Andre began by reading the problem and immediately started to verbalised: “We need to print it is sorted or not, okay [pause]”.

Andre wrote the method name and the array of type integer as a passing parameter. He expressed doubt about the method returned value and finally he decided that the method did not return any value. Then he verbalised: “Is sorted [pause], um, we need to check where the first is sorted or not, okay, we need to check for each row”.

After the above utterance, Andre started to write a nested FOR-loops block while he verbalised his writing. Then Andre verbalised: “Now we need to check if this row sorted or not”.

After a short pause, he verbalised while writing an IF-statement: “If ah [pause], let me see, if if a1 row and column [a1[row][col]] greater than a1 row and column pulse one [a1[row][col+1]], [pause], okay, so , ah if greater than, ah, [pause]”.

Then Andre decided to define a follower variable of type Boolean and set its value to true, followed by adding a Java command inside the IF-block (Figure A.6, steps1 →6). After a short pause, Andre decided to add IF-ELSE block (see Figure A.6, step7)).

Before compiling his code, Andre started to read part of the code: “Let me check a1 row column [a1[row][col]] greater than a1 row column plus one [a1[row][col+1]], after reaching the end go back to the column , then go to the row”.

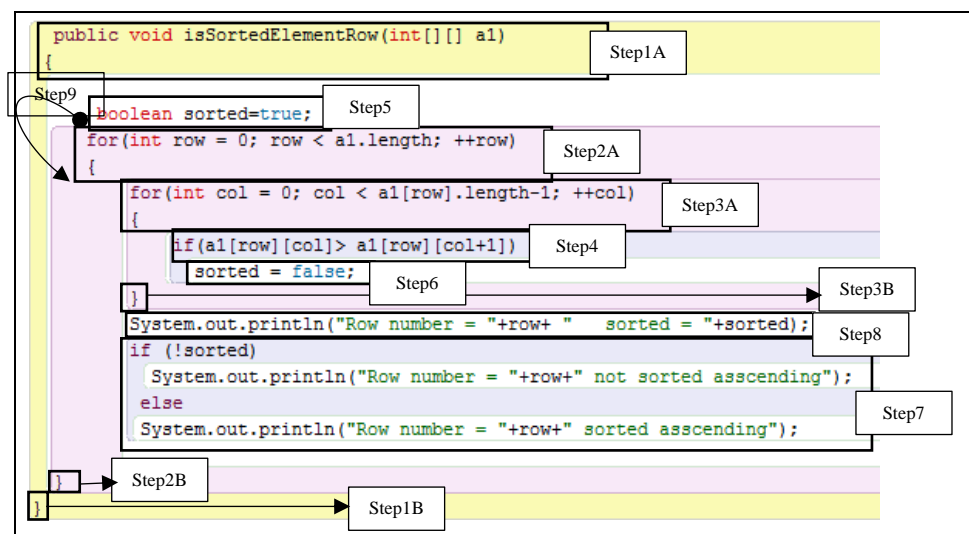


Figure A.6 Andre’s screen image for checking if sorted ascending each row of a 2D

Andre ran the supplied unit tests. He started to compare the terminal results with the results in the question paper. Andre verbalised: “Oh, no, I will check it, not sorted, same, not sorted, not sorted, let me see”.

Andre started to view one of the unit test file (isSortedElementRow(new int[][] {{100, 1, 2}}, {3, 4, 7}, {-1, 2, 3})) and verbalise: “The first row not sorted, the second one yes, the third one as well”.

After the above utterance, Andre added a PRINT Java command (see Figure A.6, step8). Andre re-ran one of the supplied unit test and verbalised: “For the row zero not sorted, yes. For the row one, not sorted wrong, for the row two not sorted wrong, I got it, I need every time to set sorted to true”.

After the above utterance, Andre started to update his code, he changed the position of the follower Boolean variable (see Figure A.6, step9).

2. Retrospection:

Interviewer: “Have you seen this question before?”

Andre: “Ah, check if sorted may be in assignment”

Interviewer: “In the assignment”

Andre: “It is similar, also similar to check if beepers are sorted beeper in the stack. I still remember the image vivid about the robot moving and picking beepers.”

Andre: Sum of all the Odd Marks for Each Student in a Collection of Student Objects (Seq5 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	7 minutes and 7 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week seven of the P2 course	
Important observations with respect to prior sessions	In previous meeting sessions, Andre found no difficulties in using his existing schemas for ArrayList. Also, he found no difficulties for solving summation of odd and even numbers in a one and two-dimensional array.	

Data

Think aloud:

Andre started to write his solutions without hesitation or verbalisation line by line as shown in Figure A.7 (left).

```

public void sumOfOddMarks()
{
    for(int i = 0; i < students.size(); ++i)
    {
        for(int j = 0; j < students.get(i).studentMark.length; ++j)
        {
            if(students.get(i).studentMark[j]&2 != 0) {
                sum= sum+ students.get(i).studentMark[j];
            }
        }
        System.out.println("Student name "+students.get(i).studentName);
        System.out.println("Sum of odd marks "+sum);
    }
}

public void sumOfOddMarks()
{
    for(int i = 0; i < students.size(); ++i)
    {
        int sum = 0;
        for(int j = 0; j < students.get(i).studentMark.length; ++j)
        {
            if(students.get(i).studentMark[j]&2 != 0) {
                sum= sum+ students.get(i).studentMark[j];
            }
        }
        System.out.println("Student name "+students.get(i).studentName);
        System.out.println("Sum of odd marks "+sum);
    }
}

```

Figure A.7 Andre’s first and second screen images for sum of all the odd marks for each student in a collection of Student objects

Andre started to read part of his code: “FOR statement, I need for each student. Then For int [integer] j equal zero, less than student mark [st.studentMark], j plus plus. IF statement to compare, sum plus student mark, [pause] then I need to print”.

After a short pause, Andre updated his code by adding gatherer variable and set its value to zero (see Figure A.7 (right)). Andre ran the supplied unit tests, all tests passed from first trial.

Andre: Students' Marks Sorted in a Collection of Student Objects (Seq5 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	Verbalise
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	10 minutes and 20 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	None	
Timing	Week seven of the P2 course	
Important observations with respect to prior sessions	In previous meeting sessions, Andre found no difficulties in using his existing schemas for the ArrayList and checking whether or not the one-dimensional and two-dimensional array were sorted.	

Data

Think aloud:

Andre began by reading the problem and verbalised: *“First I need to compare the first with the second, second with the third. If all of them in ascending order, the marks is sorted [pause]. I think we need to Boolean”*.

After the above utterance, Andre started to verbalise while writing his code (see Figure A.8 (left)): *“Student mark sorted each [studentMarkSortedEach ()], FOR-statement, I need for each student. Similar to 1D array. Boolean sort equal true. For each student. For int [integer] j equal zero, less than student mark [st.studentMark], the last not included so dot length minus one [st.studentMark.length-1], j plus plus. IF-statement to compare the mark, each with the next. If greater than false. At the end of FOR-loop. I need to check if sorted or not. If sorted print ascending. If not print descending [pause], not sorted”*.

Andre ran the supplied unit tests. He started to compare the terminal results with the results in the question paper. Andre verbalised: *“Sorted, not sorted, not sorted, yes sorted, ah, [pause] I forgot to print the students' names”*.

Andre updated his code by adding the PRINT-statement and re-ran the supplied unit tests (see Figure A.8 (right)).

```

void studentMarkSortedEach(){
    for (int i=0; i<students.size(); i++){
        Students st=students.get(i);
        boolean sort=true;
        for (int j=0; j<st.studentMark.length-1; j++){
            if (st.studentMark[j]>st.studentMark[j+1])
                sort=false;
        }
        if (sort)
            System.out.println("student Marks = sorted ascending");
        else
            System.out.println("student Marks = not sorted ascending");
    }
}

void studentMarkSortedEach(){
    for (int i=0; i<students.size(); i++){
        Students st=students.get(i);
        boolean sort=true;
        for (int j=0; j<st.studentMark.length-1; j++){
            if (st.studentMark[j]>st.studentMark[j+1])
                sort=false;
        }
        System.out.println("Students name "+st.studentName);
        if (sort)
            System.out.println("student Marks = sorted ascending");
        else
            System.out.println("student Marks = not sorted ascending");
    }
}
    
```

Figure A.8 Andre's first and second screen images for students' mark sorted in a collection of student object

Luke: Counting the Length of One Corridor (Seq1 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	3 minutes and 5 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	1
Intervention	None	
Timing	Week four of the P1 course	
Important observations with respect to prior sessions		

Data

1. Think aloud:

Luke started by a WHILE-loop statement, followed by a Robot World command that allowed the robot to move across the corridor, then he verbalised: *“I need to start to declare variable int [integer] n, and after move the robot forward and WHILE loop, I need to increment the variable, let n equal to [pause] one because I need to count the square it start”*.

As he completed this utterance, he defined a gatherer variable and set its value to one, as the first Java command, after that he continued to write a Java command that increased the gatherer value by one inside the WHILE-loop block, then he started to trace his code and verbalise: *“And that should move to end of corridor, each time should increment, at the end we need to print out”*.

Then, he added a PRINT statement after the WHILE-loop block. Luke forgot the correct syntax for the PRINT message; as a result, he got a syntax error when he compiled his code and he easily fixed the error after reading the syntax error message. Luke’s code ran from the first attempt.

2. Retrospection:

Interviewer: *“Have you seen this question before?”*

Luke: *“No”*

Interviewer: *“Have you seen something similar to this?”*

Luke: *“Yes, counting the beepers”*

Interviewer: *“Counting the beepers in a single stack, is that right?”*

Luke: *“Yes”*

Interviewer: *“Did you try to compare and contrast between what you have been seen before and newly presented information?”*

Luke: *“No”*

Luke: Comparing the Length of Two Corridors (Seq1 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Sequential	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	
	<i>Time on task</i>	9 minutes and 34 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	Week six of the P1 course	
Important observations with respect to prior sessions	Luke did not encounter any significant difficulties in solving (Seq1 – Q1).	

Data

1. Think aloud:

Luke began by reading the problem and then he verbalised: “*Start with doing, two integer values*”.

After the above utterance, Luke started writing his code by initialising two gatherer variables, one for each corridor, and setting both of them to zero. He followed this with a WHILE-loop block for moving the robot and counting the length of the first corridor. Then he verbalised: “[Pause] *I just realised that this should be equal to 1*”.

Then he updated the gatherer variables and set both of them to one. Then Luke continued to type the rest of Java commands line by line as shown in Figure A.9 while he verbalised his writing. He gave no indication that he read or traced his code. Luke’s code ran from the first attempt.

2. Retrospection:

Interviewer: “*Have you seen this question before?*”

Luke: “*No*”

Interviewer: “*What was the first plan when you started?*”

Luke: “*I know I need to check the length of the top one and the bottom one, and compare them*”

Interviewer: “*What was the most difficult part for solving this question?*”

Luke: “*I found it ah an easy question.*”

The interviewer asked Luke about changing the value of both counters from zero to one

Luke: “*I remembered that it gonna move forwards and not counting the first square it is started with*”

Interviewer: “*That means you track you solution while writing the program?*”

Luke: “*Yep*”

```

public void findCorr() {
    int length0 = 1;
    int length1 = 3;

    while (!isRobotFacingWall()) {
        moveRobotForwards();
        ++length0;
    }

    turnRobotLeft();
    turnRobotLeft();

    while (!isRobotFacingWall()) {
        moveRobotForwards();
    }

    turnRobotLeft();
    turnRobotLeft();
    turnRobotLeft();

    moveRobotForwards();
    moveRobotForwards();

    turnRobotLeft();
    turnRobotLeft();
    turnRobotLeft();

    while (!isRobotFacingWall()) {
        moveRobotForwards();
        ++length1;
    }

    if (length0 > length1) {
        print("Corridor 0 is the longest. It is " + length0 + " long.");
    } else if (length0 == length1) {
        print("Both Corridors are equal, " + length0 + " is the length of both corridors");
    } else {
        print("Corridor 1 is the longest. It is " + length1 + " long.");
    }
}

```

Figure A.9 Luke's final screen image for comparing the length of two corridors

Luke: Counting the Number of Beepers in Each Stack along the Single Corridor (Seq2 – Q2)

Encoding

Question	Solved
Behaviours	Mover
Emotion	Indiscernible
Strategies	Sequential
Activities	<i>Planning</i> <i>Tracing</i> <i>Unit test</i> <i>Time on task</i> <i># of compilation</i> <i># of execution</i>
	8 minutes and 4 seconds
	2
	1
Intervention	None
Timing	Week six of the P1 course
Important observations with respect to prior sessions	Luke struggled to solve (Seq2 – Q1, Chapter 6) in week four and he was provided with an exact solution by the interviewer.

Data

1. Think aloud:

Luke began by reading the problem and immediately started to code his solution without hesitation or verbalisation. Luke initially set the gatherer variable to zero, followed by a nested WHILE-loop block that allowed the robot to count the number of beepers at its location, print the number of beepers, set the gatherer variable again to zero, and then move the robot forwards until the robot reached the last location. Finally, he added an iteration block for counting and printing the beepers at the last location. Luke forgot to close one of the brackets; as a result, he got a syntax error when he compiled his code and he could easily fix the error. Then he ran the supplied unit tests and all tests passed from the first trial.

2. Retrospection:

The interviewer asked Luke if he had solved a similar question. Luke's response was positive. Also, he added that he had practised in a homework assignment a more complex program that allowed a robot to move in a two-dimensional world, counting the number of beepers at each location, and printing * if the number of beepers were even, otherwise printing #.

Luke: Checking if integers in a 1D Array are sorted in Descending Order (Seq3 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	Read messages and test code
	<i>Time on task</i>	20 minutes and 47 seconds
	<i># of compilation</i>	8
	<i># of execution</i>	7
Intervention	1. Clarify scaffolding – provided on request 2. Hint scaffolding – provided on request	
Timing	Week eleven of the P1 course	
Important observations with respect to prior sessions	Prior to encountering this problem, Luke had undertaken a similar exercise checking to see if numbers in an array were ascending as part of his P1 course work. Luke did not solve his homework assignment.	

Data

1. Think aloud:

Luke began by reading the problem and immediately started to verbalise while writing his code line by line (Figure A.10 (left)): "Boolean *is sorted* [*isSorted*] *int* [*integer*] *int array* [*intArray*], *then ah for int* [*integer*] *i equal zero, i less than int array* [*intArray*], *i plus plus, if int array i* [*intArray [i]*] [*long pause*] *plus one equal equal int array* [*intArray+1*] [*pause*] *not equal* [*Luke change == to !=*] *return false. If not return true*".

Luke compiled his code and he easily fixed the syntax error (cannot find symbol - variable *I*, Figure A.10 (right), step1). Then Luke ran the supplied unit tests and verbalised: "There are one error and two. One of the errors say one, array of index out of bound of exception, so I need to go back and update my code".

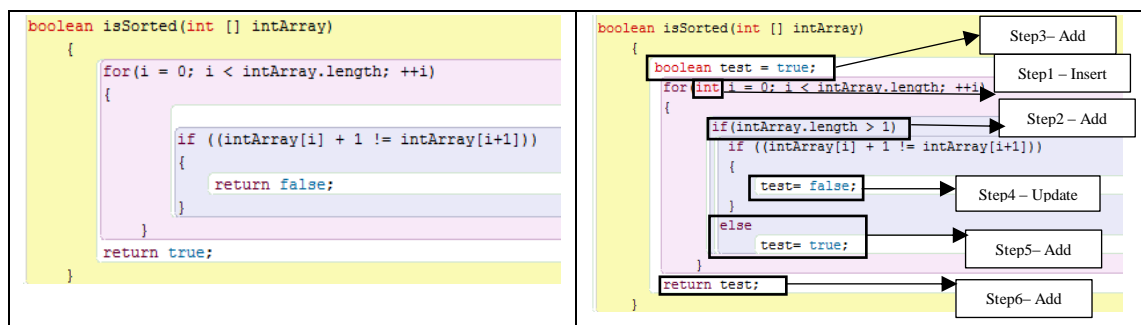


Figure A.10 Luke's first and second screen images for checking whether or not the integers were sorted in descending order

After a short pause, he started to read his code then he viewed the test error message and verbalised: “Array of index out of bound of exception [long pause]. If length of the array more than one this will work so I need IF”.

After the above utterance, Luke started to update his code as shown in Figure A.10 (right), step2. Luke re-ran the supplied unit tests and reasoned about his code’s correctness: “Two of the tests failed. Assertion error, assertion error”.

He started to read his code again. Then he started to update his code as shown in Figure A.10 (right), steps3 → 6. Luke re-ran the supplied unit tests and verbalised: “That cause more problem six tests failed [six out of eight unit tests], I need your help”.

2. Scaffolding:

Interviewer: “What do we mean about descending order?”

Luke: “Descending means sequence of numbers in descending order; is that right?”

Interviewer: “Give me an example of descending numbers.”

Luke: “Ten, nine, eight, seven, six, five, four, three, two, one, zero”

Interviewer: “Descending means the first place is bigger than the second one, the second one is bigger than the third one – as shown in the test file.”

3. Think aloud:

Luke started to update his code as shown in Figure A.11, step7. Luke re-ran the supplied unit tests. He was surprised when six supplied unit tests failed. Luke opened the supplied unit test file and started to read one of the test methods that consists of the following integer values {100, 99, 88, 77, 6,1}. Luke viewed his code: “Hundred, ninety nine, eighty eight, seventy seven, six, one. Compare first with second, second with third, [pause], ah, the result should be true. I need you help”.

```
boolean isSorted(int [] intArray)
{
    boolean test = true;
    for(int i = 0; i < intArray.length-1; ++i)
    {
        if(intArray.length > 1)
        {
            if ((intArray[i] < intArray[i+1]))
            {
                test= false;
            }
        }
        else
        {
            test= true;
        }
    }
    return test;
}
```

The screenshot shows the code with three callouts: 'Step8- Insert' pointing to the '-1' in the for loop condition, 'Step7- Update' pointing to the comparison ' $(intArray[i] < intArray[i+1])$ ', and 'Step9- Delete' pointing to the 'else' block.

Figure A.11 Luke’s third screen image for checking whether or not the integers were sorted in descending order

4. Scaffolding:

The interviewer suggested to Luke to read all the unit test messages: “Array index out of bounds exception, but I already checked if the array length greater than one”.

After the above utterance, the interviewer suggested to Luke updating the termination condition for the FOR-statement (hint scaffolding).

5. Think aloud:

Luke updated the termination condition for the FOR-statement (Figure A.11, step8). He re-ran the supplied unit tests. Luke re-opened the supplied unit test file and started to read one of the test methods that consists of the following integer values {100, 100, 100, 99,100, 99}. Luke verbalised: “True, true, true, true, false, true, ah”.

After the above utterance, Luke updated his code (Figure A.11, step9).

Luke: Checking if Sorted Ascending Each Row of a 2D Array Elements (Seq4 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Pen and paper tracing, and PRINT debugging
	<i>Unit test</i>	Read test code
	<i>Time on task</i>	17 minutes and 5 seconds
	<i># of compilation</i>	5
	<i># of execution</i>	4
Intervention	“General prompt” scaffolding – provided on request	
Timing	Week six of the P2 course	
Important observations with respect to prior sessions	Luke solved check whether or not the one-dimensional array elements were sorted in descending order (Seq3 – Q1) with interviewer’s assistance (clarify and hint scaffolding).	

Data

1. Think aloud:

Luke began by reading the problem and immediately started to verbalise while writing his code (Figure A.12 (left), steps1→4): “So method void, is sorted element row [isSortedElementRow], also I need to define two-dimensional array, for int [integer] i, i should be less than int array dot length [intArray.length], i plus plus. If [after typing the word If], I’m going to add Boolean sorted equal true, I need another for loop”.

Luke continued to verbalise while writing Java commands (Figure A.12 (left), steps5→7): “If array x i [intArray[i][x]] less than int array i x minus one [intArray[i][x-1]], sorted equal false. Then I need to print the result of each row”.

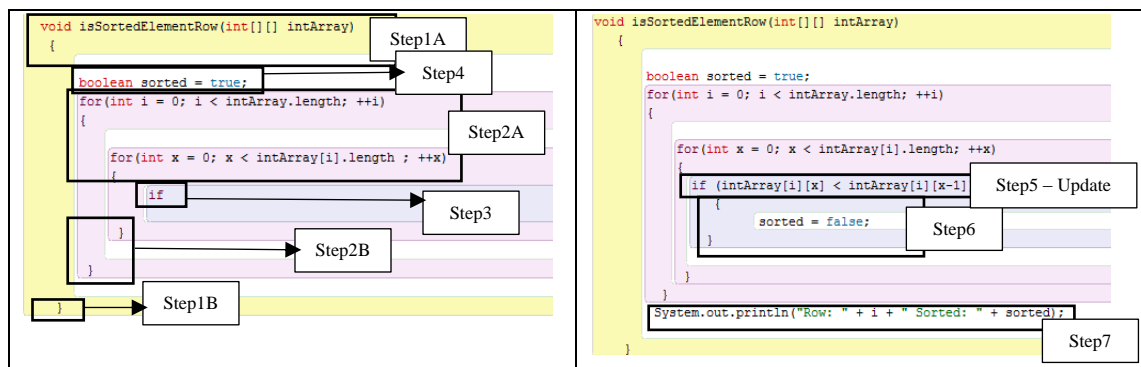


Figure A.12 Luke’s first and second screen images for checking sorted ascending each row of a 2D array

When Luke compiled his code, he got a syntax error (cannot find symbol - variable `i`). Luke self-corrected his error by changing the position of the PRINT statement (see Figure A.13, step8). He ran the supplied unit tests and all tests were failed. He then without hesitation updated the initial value of stepper variable from 0 to 1 (Figure A.13, step9). He re-ran the supplied unit tests, starting by comparing the terminal results with the results in the question paper. He discovered that the output did not match the output given to him in the question paper. Luke verbalised while writing Java commands (see Figure A.13 steps10&11): “I need to add PRINT statement the check the result”. Luke re-ran the supplied unit tests — “I need a pen”.

```

void isSortedElementRow(int[][] intArray)
{
    boolean sorted = true;
    for(int i = 0; i < intArray.length; ++i)
    {
        System.out.println("Test");
        for(int x = 1; x < intArray[i].length; ++x)
        {
            if (intArray[i][x] < intArray[i][x-1])
            {
                sorted = false;
            }
        }
        System.out.println("Test");
        System.out.println("Row: " + i + " Sorted: " + sorted);
    }
}

```

The code is annotated with four callout boxes:

- Step10**: points to the first `System.out.println("Test");` statement.
- Step9 - Update**: points to the `x = 1` initialization in the inner loop.
- Step11**: points to the second `System.out.println("Test");` statement.
- Step8**: points to the final `System.out.println("Row: " + i + " Sorted: " + sorted);` statement.

Figure A.13 Luke’s third screen image for checking sorted ascending each row of a 2D array

Luke selected one of the unit test cases and draw a two-dimensional array (see Figure A.14). Luke started to trace his code and verbalised: “One less than hundred, ah sorted equal false. Two and one, so the output still false. Four and three, sorted true. Seven less than four, true, but false why [pause]? I think, I need your help”.

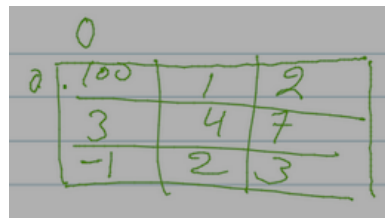


Figure A.14 Luke’s doodle for checking whether or not the integers in 1D were sorted in ascending order

2. Scaffolding:

Interviewer: “Have you seen this question before?”

Luke: “I think so, it is similar to sort ascending one-dimensional array “

The interviewer redirected Luke to write an algorithm for sorting the elements in a one-dimensional array using smartpen and paper (“general prompt” scaffolding – Figure A.15).

After, Luke finished writing his code, he started self-correcting his code by changing the position of the Boolean variable definition and initialisation.

```

for (int i = 1; i < array.length; i++) {
    if (ARRAY[i] > ARRAY[i+1])
        return false;
}
return true;

```

Figure A.15 Luke's doodle for is sorted ascending each row of a 1D array

Luke: Sum of all the Odd Marks for Each Student in a Collection of Student Objects (Seq5 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Sequential	
Activities	Planning	
	Tracing	
	Unit test	
	Time on task	4 minutes and 3 seconds
	# of compilation	2
	# of execution	1
Intervention	None	
Timing	Week seven of the P2 course	
Important observations with respect to prior sessions	In previous meeting sessions, Luke found no difficulties in using his existing schemas for ArrayList. Also, he found no difficulties for solving summation of odd and even numbers in a one and two-dimensional array.	

Data

Think aloud:

Luke began by reading the problem and immediately started to verbalise while writing his code (see Figure A.16): “For integer i equal zero, i less than student size [student.size()], plus plus. sum odd equal zero [sumOdd=0], Another For int [integer] [pause]. For int [integer] j equals 0, j less than [pause], student dot length [students.get(i).studentMark.length, plus plus j. If student mark 2 [modulo 2] not equal zero [students.get(i).studentMark[j]%2 != 0], sum odd plus student mark [sumOdd= sumOdd+ students.get(i).studentMark[j]]. I need also to print the students' names and the result”.

```

public void sumOfOddMarks()
{
    for (int i = 0; i < students.size(); ++i)
    {
        sumOdd=0;
        for (int j = 0; j < students.get(i).studentMark.length; ++j)
        {
            if (students.get(i).studentMark[j]%2 != 0)
            {
                sumOdd= sumOdd+ students.get(i).studentMark[j];
            }
        }
        System.out.println("Student name "+students.get(i).studentName);
        System.out.println("Sum of odd marks "+sum);
    }
}

```

Figure A.16 Luke's screen image for sum of all the odd marks for each student in a collection of student objects

Luke compiled his code and was able to identify the syntax error (cannot find symbol - variable sumOdd). Then he re-compile and ran the supplied unit tests and all tests passed.

Luke: Students' Marks Sorted in a Collection of Student Objects (Seq5 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	Read test output and test code
	<i>Time on task</i>	6 minutes and 46 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	None	
Timing	Week seven of the P2 course	
Important observations with respect to prior sessions	Luke solved check whether or not sorted 1D with interviewer assistance (Clarify, and hint scaffolding). Then he solved whether or not sorted 2D with interviewer assistance (“general prompt” scaffolding).	

Data

Think aloud:

Luke began by reading the problem and immediately started to verbalise while writing his code (see Figure A.17, steps 1 →4): “Boolean sort, sort should be true. For integer i equal zero, i less than student size [student.size()], plus plus. Another For int [integer] [pause]. For int [integer] j equals 0, j less than [pause], j should be less than students dot student mark and dot length [students.get(i).studentMark.length], plus plus j. If student i dot student mark j [students.get(i).studentMark[j]] grater that student i student mark j [students.get(i).studentMark[j+1]], sort false. If sort true I need to print ascending. If true I need to print not sort. I need also to print the students' names”.

```

void studentMarksSortedEach(){
    boolean sort=true;
    for (int i=0; i<students.size(); i++){
        for (int j=0; j<students.get(i).studentMark.length-1; j++){
            if (students.get(i).studentMark[j]>students.get(i).studentMark[j+1])
                sort=false;
        }
        System.out.println("Student name: " + students.get(i).studentName);
        if (sort)
            System.out.println("student Marks = soret d ascending");
        else
            System.out.println("student Marks = not soret d ascending");
        sort=true;
    }
}

```

The screenshot shows the following callout boxes:

- Step 1:** Points to the outer loop: `for (int i=0; i<students.size(); i++){`
- Step 2:** Points to the inner loop: `for (int j=0; j<students.get(i).studentMark.length-1; j++){`
- Step 3:** Points to the sorting logic: `if (students.get(i).studentMark[j]>students.get(i).studentMark[j+1]) sort=false;`
- Step 4:** Points to the printing logic: `System.out.println("Student name: " + students.get(i).studentName); if (sort) System.out.println("student Marks = soret d ascending"); else System.out.println("student Marks = not soret d ascending");`
- Step 5:** Points to the reset of the sort flag: `sort=true;`

Figure A.17 Luke's screen image for checking whether or not Students' Marks sorted in a collection of student objects

Luke ran the supplied unit tests. He started to compare the terminal results with the results in the question paper. Then Luke verbalised: “Nadia's marks sorted, yes. Zain's marks, not sorted, yes. Tom's marks not sorted, yes. Sally not sorted. But It should be sorted. Why? [Pause] Let me see

[pause]. Boolean sort true, for loop to check all the students. The second FOR loop for all marks. Then If greater sort false. Ah. I remember I need to set sort to true before comparing the marks for the second student”.

Luke update his code by adding PRINT Java command as shown in Figure A.17, step5. He re-ran the supplied unit tests to verify the correctness of his solution.

Kasper: Counting the Number of Beepers in a Single Corridor (Seq2 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Familiar first	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging
	<i>Unit test</i>	
	<i>Time on task</i>	14 minutes and 55 seconds
	<i># of compilation</i>	3
	<i># of execution</i>	3
Intervention	None	
Timing	Week four of the P1 course	
Important observations with respect to prior sessions	Kasper solved this question after solving counting the length of corridor (Seq1 – Q1, Chapter 6). Kasper was not confident about counting the length of the corridor – he was confused and this confusion led to a trial and error approach to programming.	

Data

Think aloud:

Kasper began by reading the problem. For this question a method header had been provided so it is possible to run the unit tests before any code has been written. Kasper ran the supplied unit tests to see the initial robot scenarios. After examining the starting Robot Worlds, he started with nested iteration statements that allowed the robot to pick beepers at its location and then move while counting beepers, followed by an iteration statement for picking the beepers at the last location (see Figure A.18, steps 1→2). He then verbalised: “I need to count, [pause]. Also, I need to print”.

```

void countBeepers() {
    int countBeeper = 0;
    while (isSpaceInFrontOfRobotClear()){
        while (isItemOnGroundAtRobot()){
            pickUpItemWithRobot();
            countBeeper++;
        }
        moveRobotForwards();
    }
    while (isItemOnGroundAtRobot()){
        pickUpItemWithRobot();
        countBeeper++;
    }
    print ("Beepers=" + countBeeper);
}

```

The code is annotated with six steps:

- Step 1:** `pickUpItemWithRobot();` (inside the first nested while loop)
- Step 2:** `moveRobotForwards();` (inside the outer while loop)
- Step 3:** `int countBeeper = 0;` (initialization)
- Step 4:** `countBeeper++;` (inside the first nested while loop)
- Step 5:** `print ("Beepers=" + countBeeper);` (final print statement)
- Step 6:** `pickUpItemWithRobot();` (inside the second while loop)

Figure A.18 Kasper’s screen image for counting the number of beepers in a single corridor

After a short pause, Kasper started to update his code as shown in Figure A.18, steps 3→5. Kasper ran the supplied unit tests one test out of two failed (the Robot Word scenario with beepers at the last stack). Kasper verbalised: *“I did not count the last beeper”*.

Kasper started to update his code as shown in Figure A.18, step6 without any evidence that he tried to mentally trace or read his code. Kasper ran the supplied unit tests to verify the correctness of his solution.

Kasper: Counting the Number of Beepers in each Beeper Stack across a Single Corridor (Seq2 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Visual debugging and pen and paper tracing
	<i>Unit test</i>	Read test output
	<i>Time on task</i>	11 minutes and 49 seconds
	<i># of compilation</i>	4
	<i># of execution</i>	4
Intervention	“General prompt” scaffolding – provided on request	
Timing	Week six of the P1 course	
Important observations with respect to prior sessions	Kasper did not encounter any significant difficulties solving counting the number of beepers in a single corridor (Seq2 – Q1).	

Data

1. Think aloud:

Kasper started with nested iteration statements that allowed the robot to count the number of beepers at its location and then move while counting beepers, followed by PRINT statement (see Figure A.19 (left)). Kasper ran the supplied unit tests. Kasper then focused on one of the robot scenarios (Robot World scenario of corridor length 5), ignored the test’s messages, and verbalised: *“Pick up all the beepers in the first position without move the robot”*.

Kasper changed the position of the PRINT statement as shown in Figure A.19 (right), step1 followed by adding a robot method call that allowed the robot to move forward one step (see Figure A.19 (right), step2). Kasper re-ran the supplied unit tests. Kasper again focused on one of the robot scenarios (Robot World scenario of corridor length 5), ignored the test’s messages, and verbalised: *“Why test does not work. Another WHILE loop for the last one”*.

Kasper updated his code as shown in see Figure A.19 (right), step3. Kasper re-ran the supplied unit tests for the third time – *“Wrong answer”*.

Kasper directly asked for help.

Figure A.19 Kasper’s first and second screen images for counting the number of beepers in each beeper stack across a single corridor

2. Scaffolding:

The interviewer gave Kasper a robot image scenario, and a trace table with two columns headed num**Of**Beeper**s**, and PRINT statement. The number of beepers at each location was recorded by the interviewer as [2, 1, 1, 2]. Kasper was asked to complete the trace table. Figure A.20 shows what he wrote in the trace table. Desk checking his code helped Kasper identify the problem and he was able to update his code changing the position of PRINT statement and and then reset the gatherer to zero before counting the beepers in the next stack.

Figure A.20 Trace-table for Kasper’s code for counting the number of beepers in each stack across a single corridor

Kasper: Checking if integers in a 1D Array are sorted in Descending Order (Seq3 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental and pen and paper tracing
	<i>Unit test</i>	Read the unit test message
	<i>Time on task</i>	11 minutes and 22 seconds
	<i># of compilation</i>	4
	<i># of execution</i>	4
Intervention	“General prompt” – provided on request	
Timing	Week eleven of the P1 course	
Important observations with respect to prior sessions	Kasper correctly solved a similar question in the homework assignment that takes an integer array as an input parameter, and returns true if the values in the array are in ascending numerical order. Kasper showed evidence many times during the meeting sessions that he found difficulty in using a nested IF-ELSE block (Seq1 – Q2 and Seq1 – Q3, Chapter 6).	

Data

1. Think aloud:

Kasper started by declaring a method header including parameter and return type, and then he wrote the FOR-loop statement. Then, he decided to define a Boolean variable `sorted` before the FOR-statement and set its value to `true`. Inside the FOR-loop block, Kasper added an IF-ELSE block followed by the RETURN statement (as shown in Figure A.21 (left)).

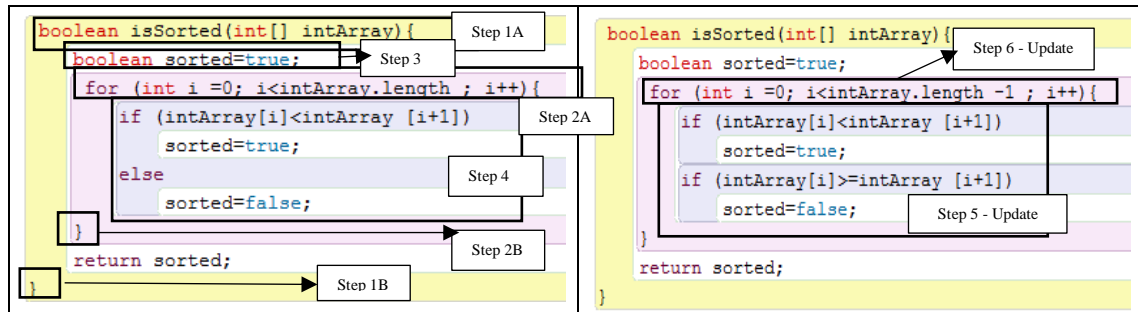


Figure A.21 Kasper's first and second screen images for whether or not an integer ID array were sorted

When Kasper ran the supplied unit tests, he discovered that only two tests passed out of eight. Then he verbalised: “If the first one is less than the second [pause], if I assumed the first element 5 less than 3 sorted false”.

After the above utterance, he started to update his code. Kasper used two IF-blocks inside of IF-ELSE block (see Figure A.21 (step5) (right)). Kasper's fragile knowledge about IF-ELSE block perform the same fundamental logic as two IF-blocks is exemplified here (Figure A.21). Kasper re-ran the supplied unit tests. Kasper started to read on of the unit test messages and verbalised: “Array index out of bound exception”

After that Kasper updated the termination condition for the FOR-loop (see Figure A.21 (step6) (right)). Kasper re-ran the supplied unit tests for the third time. He discovered that only four tests passed. After a long pause, Kasper asked for help.

2. Scaffolding:

The way Kasper wrote his code made the interviewer doubt about his understanding the difference between ascending and descending. Therefore, the interviewer draw two one-dimensional arrays (Figure A.22). These two one-dimensional arrays examples were selected from the unit test file. The following is part of the conversation between the interviewer and Kasper:

Interviewer: “Did you see this question before?”

Kasper: “No”

Interviewer: “Even in the homework assignment?”

Kasper: “I might see the question, but I do not remember the word ascending or descending”

After that the interviewer requested Kasper to trace his code, but as usual Kasper decided to use PRINT statement. Figure A.22 shows how he updated his code using PRINT statement. After re-running the supplied unit – `isSorted(new int[] { 100, 99, 88, 77, 6,1}`).

Kasper started to read the result of PRINT-statement. Without any verbalisation, Kasper started to update his code.

```

boolean isSorted(int[] intArray){
    boolean sorted=true;
    for (int i =0; i<intArray.length -1 ; i++){
        if (intArray[i]<intArray [i+1])
            sorted=true;
        if (intArray[i]>=intArray [i+1])
            sorted=false;
        println("i = "+i+" sorted = "+sorted);
    }
    return sorted;
}

```

Figure A.22 Kasper’s third screen image for whether or not an integer 1D array were sorted

Kasper: Smallest Element in a 1D Array (Seq3 – Q2)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	7 minutes and 2 seconds
	<i># of compilation</i>	1
	<i># of execution</i>	1
Intervention	None	
Timing	After he had finished the P1 course.	
Important observations with respect to prior sessions	For Seq1 – Q4 (Chapter6), Kasper was clearly outside of his ZPD, and he was supplied with the model answer for this question. Kasper was then able to solve this question, which suggests that he learnt from the model answer and was able to apply that learning to new situations.	

Data

1. Think aloud:

Kasper began by reading the problem and immediately started to verbalise and write his solution:

“Do not return everything, void find smallest, and it does take any array of integers. Let us call it array of int [integer]”.

After Kasper wrote the method signature, he changed his mind about the method type and verbalised: *“Oh, it does return the element of the smallest”.*

After the above utterance, he wrote the FOR-loop statement that consisted of the stepper variable x. At this stage, he hesitated about the syntax of FOR-statement - using , or ;, after a short pause he made his decision to use ;. Then, he decided to define the most wanted holder variable smallest before the FOR-statement and set its value to zero:

“So, array int x [pause], I assumed the first element has the smallest element, so I will call it integer smallest equal zero”.

Inside the FOR-block, Kasper added the assignment Java command (see Figure A.23 (left)). After a pause, Kasper expressed doubt about the initial value of the gatherer variable: *“Integer smallest equal zero [pause], so integer smallest equal to array int zero [arrayInt [0]]”.*

As he completed his utterance, he started to update the value of the gatherer variable from zero to the first element of the array, followed by updating the body of the FOR-loop block, and adding an IF-statement. Finally he added the RETURN statement as the last Java command (see Figure A.23 (right)). When Kasper ran the supplied unit tests, they all passed at the first trial.

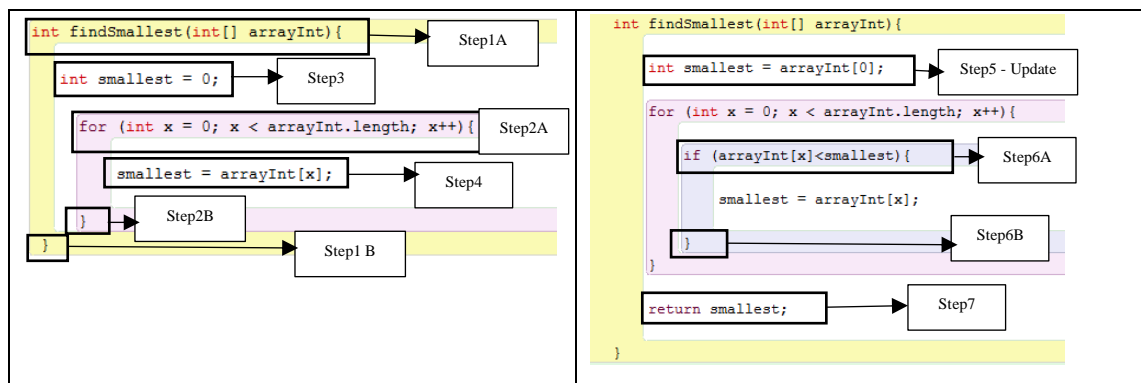


Figure A.23 Kasper's first and second screen images for smallest element in a 1D array

2. Retrospection:

Interviewer: "Have you seen this question before?"

Kasper: "I see similar one"

Interviewer: "Which question was similar?"

Kasper: "Shortest corridor"

Interviewer: "Did you try to compare and contrast between what you had seen before and newly presented information?"

Kasper: "Yes"

Interviewer: "Does this mean that the whole plan was in your mind?"

Kasper: "Ah, the first plan was to use array because I forget it. Then I started to think should I use it with FOR-loop or WHILE- loop, then I remembered in the homework assignment we always use array with FOR-loop"

Interviewer: "Did you remember, how you solved the smallest corridor task?"

Kasper: "I remembered, when I started to think about it. I remembered"

Interviewer: "You initialised your counter by zero then you changed your mind, why?"

Kasper: "Because, I remember that wrong, ah. I remember your advice that we can't compare with zero, I should make the first value equal to the smallest"

At the end of the session, the interviewer discussed the quality of his code with Kasper and how he could further develop it.

Kasper: Index of the Largest Element in a 1D Array (Seq3 – Q3)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	14 minutes and 47 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	1
Intervention	None	
Timing	After he had finished the P1 course. He solved this question directly after solving found the smallest element in a one-dimensional array (Seq3 – Q2).	
Important observations with respect to prior sessions	Kasper did not encounter any difficulties in solving Seq3 – Q2.	

Data

1. Think aloud:

Kasper started to verbalise and write his solution shown in Figure A.24 (left) line by line in sequential order. “Asking for an index of array, int [integer] find largest integer, so, um, int [integer] array largest index. This similar as before. Make the first the largest. So I need to make variable integer largest equal to the first index zero, and then FOR loop, integer x equal 0, x less than largest index length, x plus plus. Find now, which one the largest, which index have the largest. So um, just I need to compare [pause] if um...[pause]. If largest index x [largeIndex[x]] greater than largest then [pause] x is the largest. Return x”.

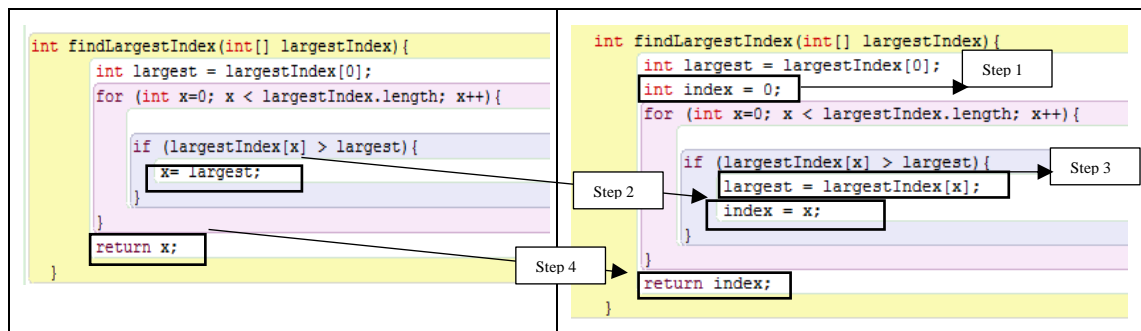


Figure A.24 Kasper's first and second screen images for find the largest index

Kasper compiled his code and was able to identify the syntax error (cannot find symbol – variable x). Kasper verbalised and updated his solution: “Variable x not defined. We should make another variable integer index equal zero. Also I need to updated the IF block”.

Also, Kasper updated the last Java command (RETURN statement) (see Figure A.24 (right)). Finally when he ran the supplied unit tests, all the tests passed.

2. Retrospection:

Interviewer: “Did you try to compare and contrast between what you have been seen before and newly presented information?”

Kasper: “My focus was how to return the index, when I compare how I could return the index therefore at the beginning I solve it in a wrong way.”

At the end of the session, the interviewer discussed the quality of his code with Kasper how he could further develop it.

Kasper: Sum of Even Numbers Stored in a 1D (Seq3 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	8 minutes and 07 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	1
Intervention	None	
Timing	After he had finished the P1 course. He solved this question directly after solving found the smallest element in a one-dimensional array (Seq3 – Q2).	
Important observations with respect to prior sessions	Kasper did not find any difficulties using a one-dimensional array concept.	

Data

Think aloud:

Kasper started with the method header with an array of type integer as the parameter and which returned an integer followed by the FOR-loop that consisted of a stepper variable called x. Inside the FOR-loop block, Kasper added an IF-statement (see Figure A.25, steps1→ 3). After changing the symbol / to % as shown in Figure A.25, step4. Kasper verbalised: “I gonna define integer variable”.

As he completed this utterance, he defined a gatherer variable and set its value to zero. He then continued to writing his code as shown in Figure A.25, steps5→ 7.

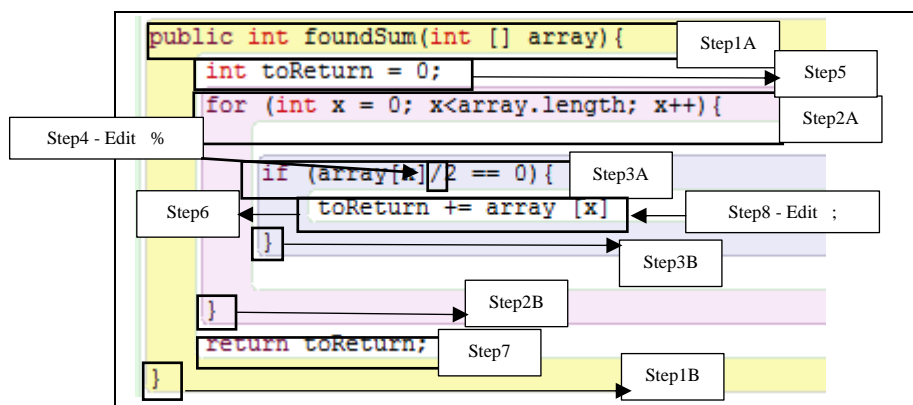


Figure A.25 Kasper’s screen image Sum of even numbers stored in a 1D

Kasper compiled his code which generated a syntax error – he had forget to add ; as shown in Figure A.25, steps6. Kasper could easily updated the syntax error. Kasper ran the supplied unit tests and all the tests passed.

Kasper: Checking if Beeper Stacks are sorted in Ascending Order by Size of the Stack (Seq2 – Q4)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Familiar first	
Activities	<i>Planning</i>	
	<i>Tracing</i>	PRINT debugging and pen and paper tracing (doodle)
	<i>Unit test</i>	
	<i>Time on task</i>	15 minutes and 22 seconds
	<i># of compilation</i>	4
	<i># of execution</i>	3
Intervention	1- Clarify scaffolding – provided on request 2- “General prompt” scaffolding – provided on request	
Timing	After he had finished the P1 course.	
Important observations with respect to prior sessions	The scaffolding given to Kasper during the meeting session (Seq3 – Q1) were effective and enable him to move forward.	

Data

1. Think aloud:

Kasper started problem solving with a mistake in the method definition. He assumed that the method did not return any value, then he decided the method should return a Boolean value, followed by WHILE-block. After that, Kasper decided to define a new method, he call it `countBeeper()`. The function of that method was to count the beepers in a single stack (see Figure A.26 (left)).

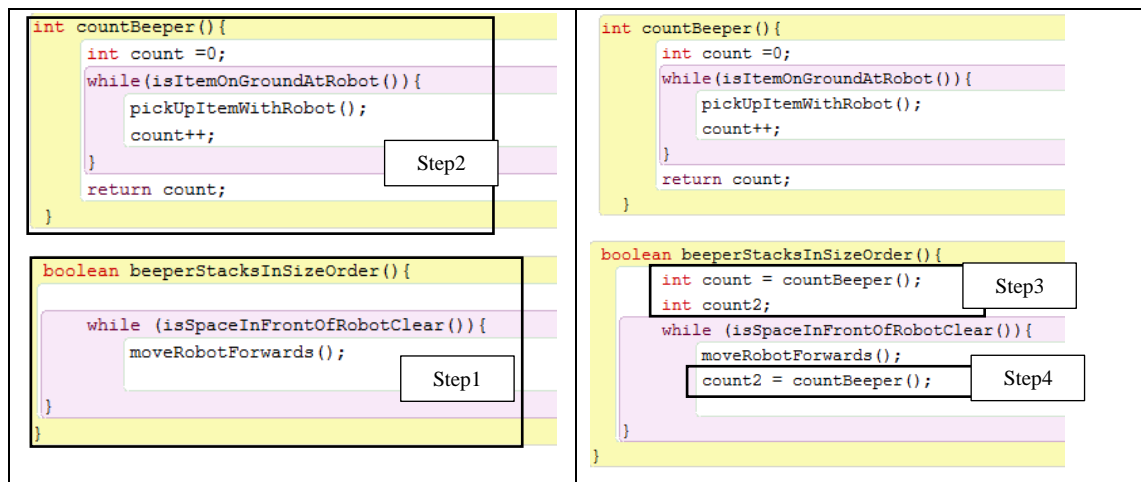


Figure A.26 Kasper's first and second screen images for check whether or not beepers stacks are sorted

After a long pause, Kasper continued to type his code while he verbalised his writing (see Figure A.26 (right)). After yet another long pause he verbalised while writing his code: “Count two [count2] less than count. Can I use pen and paper?”

Kasper immediately started to draw a one-dimensional array (as shown in Figure A.27). Then he asked for help (clarify scaffold).

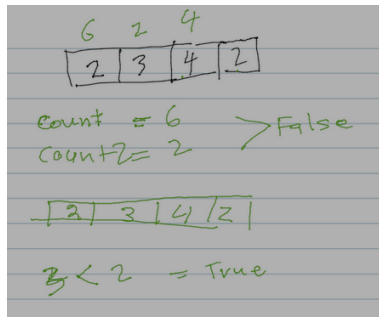


Figure A.27 Kasper's doodle for check whether or not beepers stacks are sorted

2. Scaffolding:

Kasper: "Could I add RETURN at this position [Kasper pointed at the last statement in his code]?"

Interviewer: "What is the function of RETURN Java command?"

Kasper: "To terminate the execution"

Interviewer: "That is right"

After the above conversation, Kasper updated his code (see Figure A.28 (step5) (left)).

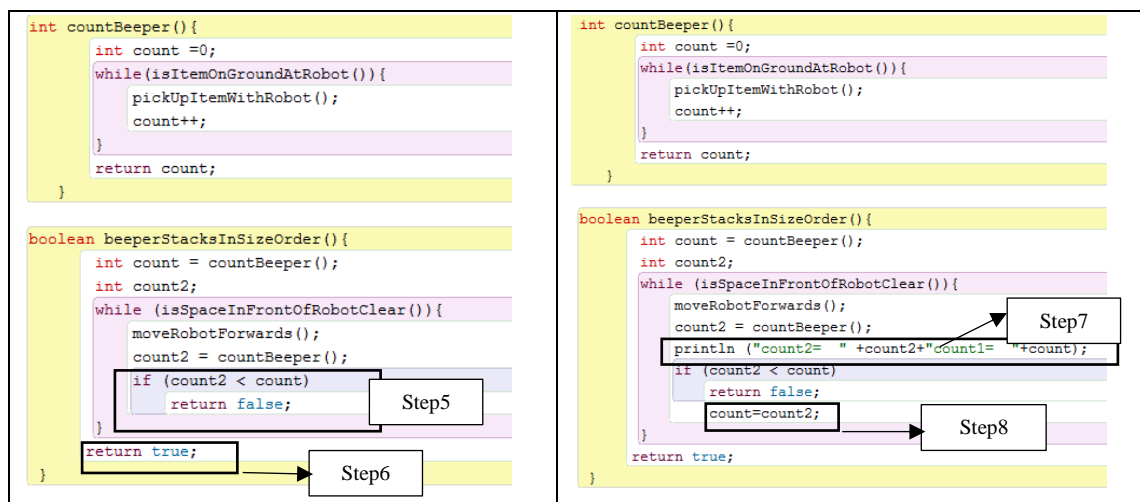


Figure A.28 Kasper's third and fourth screen images for check whether or not beepers stacks are sorted

3. Think aloud:

Kasper compiled his code and was able to identify the syntax error (missing return statement) and quickly self-updated his code by adding return true as the last Java command. Kasper ran the supplied unit tests, He discovered that only two tests passed and one test failed (the test failed – the Robot World scenario which contained beepers stacks not sorted). Then he verbalised: "One test fail. [Pause], I think I need your help".

4. Scaffolding:

The interviewer redirected Kasper to trace his code.

Kasper: "Could I add print statement?"

Interviewer: "As you like"

Kasper updated his code by adding Java command PRINT before IF-statement. After re-ran the supplied unit test case for the failed solution:

Kasper: “Ah, what, the value of count equal to the number of beepers at first location”

After the above utterance, Kasper self-updated his code (see Figure A.28 (step2) (right)).

Kasper: Sum of all the Elements Stored in Odd Indexed Rows of the 2D (Seq4 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	4 minutes and 07 seconds
	<i># of compilation</i>	2
	<i># of execution</i>	1
Intervention	None	
Timing	Week eight of the P2 course	
Important observations with respect to prior sessions	Kasper did not encounter any difficulties in solving sum of even numbers stored in a one-dimensional array.	

Data

Think aloud:

This question was the first using a two-dimensional array concept. During the meeting session, Kasper requested he use a paper that contains the syntax of a two-dimensional array. Kasper began by writing the method header with an array of type integer as a parameter. He then defined a nested FOR-loop block, followed by an IF-block, and finally he defined a gatherer variable toReturn at the beginning of the method and set its value to zero (see Figure A.29, steps1 →4). During writing his code, Kasper was observed from time to time using the paper checking Java commands.

Kasper compiled his code which gave him one error. He easily fixed the syntax error by adding the missing RETURN statement (Figure A.29, step5). He re-compiled and ran the supplied unit tests; all the tests passed.

```

int foundSum (int[][] arr2D){
    int toReturn = 0;
    for (int x = 0; x < arr2D.length;x++){
        for (int y = 0; y <arr2D[x].length; y++){
            if (x % 2 != 0){
                toReturn += arr2D[x][y];
            }
        }
    }
    return toReturn;
}

```

The screenshot shows the code with several callout boxes labeled Step1A through Step5 and Step1B. Step1A points to the method signature. Step4 points to the initialization of 'toReturn'. Step2A points to the start of the first for-loop. Step3 points to the if-statement. Step2B points to the end of the first for-loop. Step5 points to the return statement. Step1B points to the closing brace of the method.

Figure A.29 Kasper’s screen image for sum of all elements stored in odd indexed rows of the 2D

Kasper: Check Whether or not Each Row of a Two-Dimensional Array Elements are Sorted in Ascending Order (Seq4 – Q4)

Encoding

Question	Not solved	
Behaviours	Stopper	
Emotion	Indiscernible	
Strategies	Stepwise design	
Activities	<i>Planning</i>	
	<i>Tracing</i>	
	<i>Unit test</i>	
	<i>Time on task</i>	10 minutes and 20 seconds
	<i># of compilation</i>	0
	<i># of execution</i>	0
Intervention	Exact solution – provided on request	
Timing	Week nine of the P2 course	
Important observations with respect to prior sessions	The scaffolding given to Kasper during the meeting session (Seq4 – Q2) were effective and enable him to move forward (the interviewer intervened with syntax help) For solving this question, Kasper found difficulties to recall his existing schema for whether or not a one-dimensional array is sorted because until this stage the schema for sorting is not condense as a single entity.	

Data

1. Think aloud:

Kasper started with the method name and array of type integer as a passing parameter, and then he defined nested FOR-loop blocks followed by IF-block. Then Kasper verbalised while continued writing his code (see Figure A.30): “We need to call the first one and check if it sorted so we store the first [pause], the first value. I need to compare it with the next one. I need to check [long pause]. If array two $D \times y$ [arr2D[x][y]] greater than first value [firstValue]”.

After the above utterance, Kasper asked for help.

```

void isSortedElementRow(int [][] arr2D){
    int firstValue = arr2D [0][0];
    for (int x = 0; x<arr2D.length;x++){
        for (int y = 0; y<arr2D[x].length; y++)
        {
            if (arr2D[x][y]> firstValue){
            }
        }
    }
}

```

Figure A.30 Kasper’s first screen image for check whether or not each row of a 2D array elements are sorted

2. Scaffolding:

Directly Kasper asked for help. The interviewer redirected Kasper to write an algorithm that check the elements of a one-dimensional array sorted ascending (“general prompt” scaffolding). Kasper composed the list of programming processes listed in (Figure A.31). As shown in Figure A.31, Kasper failed to recall the correct code. The interviewer started to use a stepwise refinement technique to explain the code to Kasper.

```

int store = 0
if (arr[arr.length-1] > store)
    store = arr[arr.length-1]
else
    for (int x = 0; x < arr.length; x++)
        if (firstValue < arr[x])
            firstValue = arr[x];
return Sorted; else Sorted = false;

```

Figure A.31 Kasper’s doodle for whether or not an integer 1D array were sorted

Matthew: Counting the Length of One Corridor (Seq1 – Q1)

Encoding

Question	Solved	
Behaviours	Mover	
Emotion	Indiscernible	
Strategies	Sequential	
Activities	<i>Planning</i>	
	<i>Tracing</i>	Mental tracing
	<i>Unit test</i>	Read the unit test message
	<i>Time on task</i>	He took 7 minutes and 39 seconds to solve the task.
	<i># of compilation</i>	2
	<i># of execution</i>	2
Intervention	None	
Timing	Week six of the P1 course	
Important observations with respect to prior sessions		

Data

1. Think aloud:

After reading the question, Matthew verbalised: “So first one, ah, I need to create integer”.

Then he started to define a gatherer variable and set its value to zero, as the first Java command, followed by a WHILE-loop block that allowed the robot to move across the corridor, then he verbalised (see Figure A.32 (left)): “The length should be, ah, I should count where the robot is stand, should the length should be one”.

Figure A.32 Matthew’s first and screen images for counting the length of one corridor

After the above utterance, he updated the gatherer variable and set its value to one. He then verbalised while adding the Java command that increased the value of the gatherer variable by one, as shown in Figure A.32 (right): “I will continue the while loop after moving the robot forwards, ah, It will plus one to the length”.

Finally, he ran the supplied unit tests and all tests failed. He then started to read one of the unit test messages and update his code according: “Output should be length five. The length is not display so I have to write a statement for this one”.

He then added the PRINT Java command. Matthew ran the unit tests and all tests passed.

2. Retrospection:

Interviewer: *“Have you seen this question before?”*

Matthew: *“Ah, yep in the homework.”*

Interviewer: *“The same question?”*

Matthew: *“As I remember different, was moving the robot around the corners and counting the number of squares.”*

Interviewer: *“You started the problem solving by defining the variable length and set its value to zero, then you changed your mind to one, why?”*

Matthew: *“Ah, because I need to count the first square the robot started with it.”*

Interviewer: *“Which means you were tracing your code while you were writing your code?”*

Matthew: *“Yes”*

Appendix B. AUTECH Ethics Approval

AUTECH Ethics Approval Certificate
Registered Committee Number: Whalley
13332-22032013



A U T E C
S E C R E T A R I A T

22 March 2013
Jacqueline Whalley
Faculty of Design and Creative Technologies

Dear Jacqueline

Re: **13/32 Learning to Program: The development of knowledge in novice programmes.**

Thank you for submitting your application for ethical review. I am pleased to confirm that the Auckland University of Technology Ethics Committee (AUTECH) has approved your ethics application for three years until 18 March 2016.

AUTECH commends the applicant and researcher on the quality of the application.

As part of the ethics approval process, you are required to submit the following to AUTECH:

- A brief annual progress report using form EA2, which is available online through <http://www.aut.ac.nz/researchethics>. When necessary this form may also be used to request an extension of the approval at least one month prior to its expiry on 18 March 2016;
- A brief report on the status of the project using form EA3, which is available online through <http://www.aut.ac.nz/researchethics>. This report is to be submitted either when the approval expires on 18 March 2016 or on completion of the project;

It is a condition of approval that AUTECH is notified of any adverse events or if the research does not commence. AUTECH approval needs to be sought for any alteration to the research, including any alteration of or addition to any documents that are provided to participants. You are responsible for ensuring that research undertaken under this approval occurs within the parameters outlined in the approved application.

AUTECH grants ethical approval only. If you require management approval from an institution or organisation for your research, then you will need to obtain this. If your research is undertaken within a jurisdiction outside New Zealand, you will need to make the arrangements necessary to meet the legal and ethical requirements that apply within their.

To enable us to provide you with efficient service, we ask that you use the application number and study title in all correspondence with us. If you have any enquiries about this application, or anything else, please do contact us at ethics@aut.ac.nz.

All the very best with your research,

Executive Secretary
Auckland University of Technology Ethics Committee

Cc: Nadia Kasto nkasto@aut.ac.nz

AUTEC Ethics Approval Certificate
Registered Committee Number: Whalley 13332-
22032013



A U T E C
S E C R E T A R I A T

30 October 2013
Jacqueline Whalley
Faculty of Design and Creative Technologies

Dear Jacqueline

Re: Ethics Application: **13/32 Learning to Program: The development of knowledge in novice programmes.**

Thank you for your request for approval of amendments to your ethics application.

I have approved the minor amendment to your ethics application allowing the use of a 'smart pen' with video.

I remind you that as part of the ethics approval process, you are required to submit the following to AUTEK:

- A brief annual progress report using form EA2, which is available online through <http://www.aut.ac.nz/researchethics>. When necessary this form may also be used to request an extension of the approval at least one month prior to its expiry on 18 March 2016;
- A brief report on the status of the project using form EA3, which is available online through <http://www.aut.ac.nz/researchethics>. This report is to be submitted either when the approval expires on 18 March 2016 or on completion of the project.

It is a condition of approval that AUTEK is notified of any adverse events or if the research does not commence. AUTEK approval needs to be sought for any alteration to the research, including any alteration of or addition to any documents that are provided to participants. You are responsible for ensuring that research undertaken under this approval occurs within the parameters outlined in the approved application.

AUTEK grants ethical approval only. If you require management approval from an institution or organisation for your research, then you will need to obtain this. If your research is undertaken within a jurisdiction outside New Zealand, you will need to make the arrangements necessary to meet the legal and ethical requirements that apply there.

To enable us to provide you with efficient service, please use the application number and study title in all correspondence with us. If you have any enquiries about this application, or anything else, please do contact us at ethics@aut.ac.nz.

All the very best with your research,

Kate O'Connor
Executive Secretary
Auckland University of Technology Ethics Committee

CC: Nadia Kasto nkasto@aut.ac.nz

Consent Form



Project Title: Investigating Aspects Of the Learning and Teaching Of Novice Programmers

Project Supervisor: Dr. Jacqueline Whalley

Researcher: Anne Philpott, Andrew Smith, Nadia Kasto, Dr. Jiamou Liu, Dr. James Skene, Dr. Stefan Marks

- I have read and understood the information provided about this research project in the Information Sheet dated 18th January 2012.
- I have had an opportunity to ask questions and to have them answered.
- I understand that I may withdraw myself or any information that I have provided for this project at any time prior to completion of data analysis, without being disadvantaged in any way.
- I understand that the data collected will be course work submitted such as exam scripts, test scripts, programmers logbooks, programme tasks and in class activities.
- If I withdraw, I understand that all relevant information stored for research purposes will be destroyed.
- I agree to take part in this research (please, tick one) :- Yes No
- I wish to receive a copy of the report from the research:- Yes No

Date:

Participant's Signature:

Participant's Name:

Participant's Contact Details (if you wish to receive a copy of the report): _____

**Approved by the Auckland University of Technology Ethics Committee on 5th March 2012 AUTC
Reference number Whalley1230_05032012**

Note: The participant should return a copy of this form

Participant Information Sheet

Date Information Sheet Produced:

18th January 2012

Project Title:

Investigating Aspects of the Learning and Teaching of Novice Programmers

An Invitation:

I am Dr. Jacqueline Whalley a Senior Research Lecturer in the School of Computing and Mathematical Sciences at AUT and one of a group of lecturers and postgraduate students' who are interested in education research and who wish to improve the way that we teach you computer programming. I would like to invite you to participate in a research study that investigates the ways that students learn to program. Additionally, this study aims to evaluate tools and techniques that are designed assist you in learning to read and write computer programs. Your participation in this research will contribute to the postgraduate student researchers PhD theses and Honours dissertations. Participation in this study is voluntary and does not involve any additional time or work on your part. You may withdraw, without giving reasons or being disadvantaged, at any time prior to the completion of data collection.

What is the purpose of this research?

During this study extracts of coursework you complete (for example exam scripts, test scripts, programmers logbooks, programming tasks and in class activities) will be gathered and all data that identifies you will be removed. These extracts will be combined with other students' answers to the same questions, and the resulting data will be analysed. The results of studying how you read, write and debug code, or answer questions about code, should enable us to develop a model of the steps involved in learning to program. We will be seeking your feedback and asking you your opinions about the tools we use to teach you and their usefulness. The information you contribute will allow us to determine how to improve our teaching and your learning. The results of this research will be published in the graduate student researchers 'theses/dissertations and may be published in academic journals or presented at conferences.

How was I identified and why am I being invited to participate in this research?

You have been invited to participate in this research because you are taking a computer programming course. All students who are enrolled in introduction to Programming (405708) or Programming 1 (405701) or Programming 2 (405704) have been invited to participate.

What will happen in this research?

All you need to do is complete the consent form supplied, stating whether or not you are willing to allow the researchers to analyse your course work (for example exam scripts, test scripts, programmers logbooks, programme tasks and in class activities). The questionnaires and any tasks will be completed in class time because they form part of your normal required course work for your programming paper. You do not have to do anything else.

What are the discomforts and risks?

Because some of the researchers are your lecturers we need to ensure that that you are protected whether or not you decided to participate in the research.

The RESEARCH TEAM are all from the school of Computing and Mathematical Sciences at AUT:

STAFF: Anne Philpott [Senior Lecturer, teaches Programming 2]; Dr. Stefan Marks [Lecturer, teaches Programming 2]; Dr. James Skene, [Lecturer, teaches Programming 1], Dr. Jiamou Liu, [Lecturer]

STUDENTS: Your anonymised data will be included in their research & theses, they will not be able to identify you individually; Nadia Kasto [PhD student] and Andrew Smith [Honours Student]

Consent Form



Project Title: Learning to Program: The Development of Knowledge in Novice Programmers

Project Supervisor: Dr. Jacqueline Whalley

Researcher: Nadia Kasto

- I have read and understood the information provided about this research project in the Information Sheet.
- I have had an opportunity to ask questions and to have them answered.
- I understand that I may withdraw myself or any information that I have provided for this project at any time prior to completion of data analysis, without being disadvantaged in any way.
- If I withdraw, I understand that the data I contributed up to the date of my withdrawal from the study may be included in the study if it has already been included in an analysis. If analysis has not yet taken place my data will be destroyed at the time of my withdrawal. I am under no obligation to continue attending the think aloud interview sessions when I choose to withdraw.
- I understand that the data collected will be recordings of me thinking-out-loud while performing programming tasks
- I understand that all relevant information stored for research purposes will be destroyed after six years.
- I agree to take part in this research (please, tick one) :- Yes No
- I wish to receive a copy of the report from the research:- Yes No

Date:

Participant's Signature:

Participant's Name:

Participant's Contact Details (if you wish to receive a copy of the report): _____

**Approved by the Auckland University of Technology Ethics Committee on 22nd March 2013 AUTEK
Reference number Whalley1332_22032013**

Note: The participant should return a copy of this for

Participant Information Sheet



Date Information Sheet Produced:

25th February 2013

Project Title:

Learning to Program: The Development of Knowledge in Novice Programmers

An Invitation:

I am Nadia Kasto, a postgraduate student in the School of Computing and Mathematical Science at AUT. I am interested in education research; my PhD thesis focuses on the way that students learn computer programming. I would like to invite you to participate in the research study that investigates the ways that students learn to program.

What is the purpose of this research?

The aim of this research is to investigate how you integrate new programming structures or elements into your current understanding of code. It is anticipated that the results of this study will inform the way in which we teach computer programming and design learning tasks.

How was I identified and why am I being invited to participate in this research?

You have been invited to participate in this research because you are taking a computer-programming course. All students who are enrolled in *Introduction to Programming (405708)* or *Programming 1 (405701)* or *Programming 2 (405704)* have been invited to participate.

What will happen in this research?

During this study, you will be asked to fill in a questionnaire about your prior learning of computer programming. If you volunteer for this study, you will be asked to attend twelve to sixteen 40 minute sessions out of class time over two semesters in your first year of learning to program. These sessions will be scheduled in consultation with you. In these sessions, you will be asked to undertake small programming tasks and to talk to me about what you are thinking as you work on the tasks. The tasks you are asked to undertake are timed so that you will have already covered the concepts in your programming course.

What are the discomforts and risks and how will these discomforts and risks be alleviated?

In the interview session, you may be feeling embarrassed if you find it difficult to solve the programming task. We will assist you to clarify your thinking and your strategy for solving the program task will neither be criticising nor providing negative evaluation of your existing practice.

What are the benefits?

The information you contribute will allow us to determine how to improve our teaching and your learning. The results of this research will be published in the graduate student researchers' theses and may be published in academic journals or presented at conferences. It will also likely improve your programming skills.

How will my privacy be protected?

Participants will not be identified in final report, thesis and resulting publication. Any conversation as a result interviews process will be treated in a strictly confidential manner. All data will be stored in a secure manner in an anonymised format.

What are the costs of participating in this research?

After you have completed 30 minutes interview session, you will be given the opportunity to have a ten minute tutoring session as a form of koha (a Maori tradition of giving a gift as a thank you for your contribution to the research) which focuses on the tasks and related programming concepts that you have just completed.

What opportunity do I have to consider this invitation?

After you have been provided with the information sheet you have a week to respond. If later, you would like to participated place contact the project supervisor. Depending on the stage of the research, it may be possible to join the study.

How do I agree to participate in this research?

If you agree to participate in this research, you need to complete the attached Consent Form or download it using the blackboard course.

Will I receive feedback on the results of this research?

If you want to receive a copy of this research, you need to tick the appropriate box in the Consent Form.

What do I do if I have concerns about this research?

Any concerns regarding the nature of this project should be notified in the first instance to the Project Supervisor, Dr. Jacqueline Whalley, jwhalley@aut.ac.nz, 921 9999 ext. 5203.

Concerns regarding the conduct of the research should be notified to the Executive Secretary, AUTEK, Dr Rosemary Godbold, rosemary.godbold@aut.ac.nz , 921 9999 ext 6902.

Whom do I contact for further information about this research?**Researcher Contact Details:**

Nadia Kasto, nkasto@aut.ac.nz, 921 9999 ext. 5852.

Project Supervisor Contact Details:

Dr. Jacqueline Whalley, jwhalley@aut.ac.nz, 921 9999 ext. 5203.

**Approved by the Auckland University of Technology Ethics Committee on 22nd March 2013 AUTEK
Reference number Whalley1332_2203201**

Appendix C. Prior Knowledge Questionnaire

Part 1: Personal Information

Name ----- Student ID -----

(Please, tick the correct choice below)

Part 2: Background

2. Do you have any programming experience? Yes No

If the answer is 'Yes' please answer 2.1 and the following sections

2.1. Where did you first learn to program?

- Before Higher School High School self-taught
 University other please explain -----

2.2. What programming language do you know?

- C C++ Java Java Script
 Pascal Delphi Perl Basic
 Visual Basic Alice Scratch PHP
 Python Others please explain

2.3. If you have learnt programming, how well do understand the following concepts

Concepts	low	High
1- Assignment concept	<input type="checkbox"/>	<input type="checkbox"/>
2- Selection concept	<input type="checkbox"/>	<input type="checkbox"/>
3- Iteration concept	<input type="checkbox"/>	<input type="checkbox"/>
4- Functions	<input type="checkbox"/>	<input type="checkbox"/>
5- Array	<input type="checkbox"/>	<input type="checkbox"/>
6- Objects	<input type="checkbox"/>	<input type="checkbox"/>
7- Inheritance	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> Others please explain		

Appendix D. The Learning Outcomes of the P1 Course

The learning outcomes of the P1 course are that the students be able to¹⁵:

1. Write syntactically correct program statements.
2. Assemble a program from statements that control the order in which tasks are performed.
3. Assemble a program from statements that control the representation and processing of data.
4. Read programs and predict what they do.
5. Design and write programs to solve simple problems.
6. Find and fix errors in programs.
7. Write programs that interact with the user and the execution environment.
8. Use tests to control program quality.
9. Apply programming and documentation standards.

¹⁵ Taken from AUT website course descriptor

Appendix E. The Learning Outcomes of the P2 Course

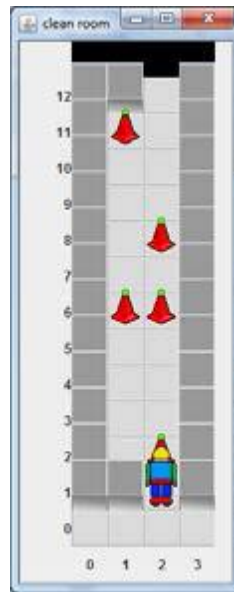
The learning outcomes on successful completion of P2 course is for students to be able to¹⁶:

1. Explain inheritance and polymorphism.
2. Explain the concept and uses of abstract classes.
3. Explain the concept and uses of interfaces.
4. Explain and apply recursion in an appropriate context.
5. Use inheritance in the correct programming context.
6. Use abstract classes in the correct programming context.
7. Use interfaces in the correct programming context.
8. Employ exception handling and create exception classes.
9. Apply the principles of serialization.
10. Interpret software requirements.
11. Develop software with a modular design.
12. Demonstrate the skills used in effective software testing.
13. Explain the processes involved in software testing.
14. Design and write automated unit tests.
15. Develop a simple graphical user interface.
16. Evaluate and use a variety of data structures.
17. Design, evaluate and implement efficient algorithms.

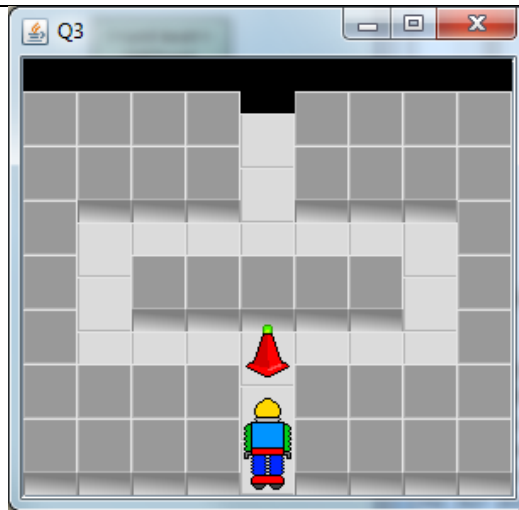
¹⁶ Taken from AUT website course descriptor

Appendix F. Questions for Developing of Writing Difficulty Metric

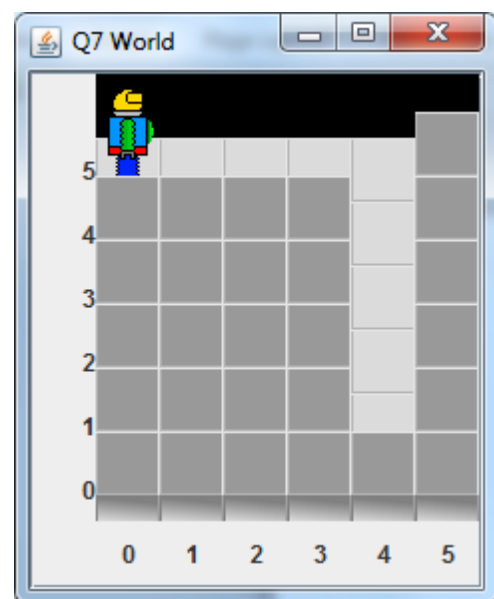
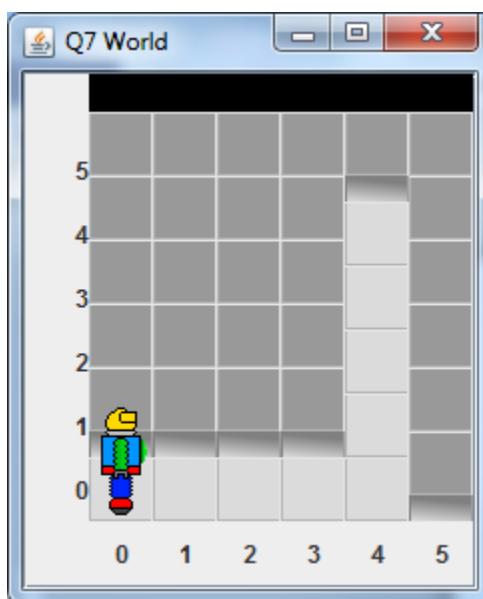
1. This question asked the students to write a method that made the robot clean the room. The robot must pick up all the beepers left lying around and if there were enough beepers to fully load the beeper wash, then they should be loaded into the beeper washer (at location (2,12)) any remaining beepers should be neatly placed at the location (2,0). The students were supplied with the method signature and the Unit tests to test that the beepers have been dropped at the appropriate location(s). The tests included starting worlds with 0, 5, 9, 10, 15 and 20 beepers.



2. This question asked the students to write a method called *advanceRobot()* that had two parameters a robot name and a distance to travel (the number of cells that the robot should advance). The robot should only be able to move if it is alive and if the distance to travel is positive if it is unable to move an appropriate exception should be thrown. If the robot encountered a wall before moving the full distance it should stop rather than crashing. The method should return true only if the robot moved the full distance.
3. In this question the students were asked to write a code to move the robot from a set starting location at (4, 0) to a fixed exit at location (4, 6). In order to do this, the robot must choose one of two paths. If there was a beeper at the first intersection (4, 2) then the robot must follow the eastern path otherwise it should follow the western path.



4. In this scenario, there were two corridors with a gap between them. The length of each of the corridors changes randomly every time the world is created. The question asked the students to compare the length of two corridors and print out a message that either states the corridors were of equal length or gave the length of the longest corridor.
5. The students were provided with a method header and asked to write a summing algorithm that made a robot move forwards until it reaches a wall while picking up any beepers that it encounters and then print out the total number of beepers the robot collected.
6. This question asked the students to complete the method *findBeeper()* that moved the robot through a spiral maze until it reaches a beeper. They should also count how many steps the robot navigates to the beepers and return the number of steps required.
7. A robot starts in one of two possible initial states, as shown in the figures below:



This question asked the students to write a program to move the robot to the end of the corridor. If the robot started at location (0, 0), it must finish at location (4, 4)

facing north. If the robot started at location (0, 5), it must finish at location (4, 1) facing south.

8. In this question, the students were provided with a robot in a cell that contains a number of beepers. The students were asked to write a method called *pickUpNBeepersCheckIfAll()* that took an integer parameter, and made the most recently created robot pick up that number of beepers from the beeper stack at its current location. The student should assume that there were enough beepers in the stack for the robot to do this safely. The method should return true if the robot has picked up all the beepers at its current location, or false if there are still beepers on the ground.

9. This question asked the students to write a method called *pickUpBeeperStack()* that made the most recently created robot pick up all the beepers at its current location. The method should return no value and take no parameters.

10. For this question, the students were supplied with the method header. They were asked to complete the method body so that the robot turns left, then if there is no wall in the way moves forward one cell.

11. For this question, the students were supplied with the method header. They were asked to complete the method body by writing a sequence of three statements to make the robot drop the beeper it is carrying, then move the robot forward one cell and turn the robot left once.

Appendix G. Participants Categorisation According to Their Ability to Solve the Programming Tasks

	Andre	Luke	Chen	Isaac	Harry	Kasper	Matthew
Seq1 – Q1	Prompt (O)	Solve	Hint	Not solved	Solve	Solve	Solve
Seq2 – Q1	Clarify	Not solved	Solve	Not solved	Solve	Solve	Not solved
Seq1 – Q2	Solve	Solve	Hint, Hint	Clarify, Clarify, Hint	Solve	Solve	Solve
Seq2 – Q2	Solve	Solve	Solve	Solve	Hint, Hint, Hint	Prompt (T)	Prompt (T)
Seq1 – Q3	Hint, Prompt (O)	Not solved	Not solved	Not solved	Prompt (T)	Hint	Hint, Prompt (O)
Seq2 – Q3	Clarify	Solve	Not solved	Solve	Not solved	Prompt (#)	Prompt (T), Prompt (T)
Seq1 – Q4	Solve	Prompt (T)	Solve	Prompt (T)	Solve	Not solved	Not solved
Seq3 – Q1	Solve	Clarify, Hint	Hint, Prompt (T)	Not solved	Not solved	Prompt (T)	Not solved
Seq3 – Q2	Prompt (#)	Solve	Prompt (#)	Solve	Prompt (T)	Solve	Prompt (#)
Seq3 – Q3	Solve	Solve	Hint, Hint, Prompt (T)	Solve	Not solved	Solve	Not solved
Seq3 – Q4	Solve	Solve	Prompt (T)	Prompt (#)	Solve	Solve	Prompt (T)
Seq2 – Q4	Solve	Solve	Not solved	Solve	Prompt (T)	Clarify, Prompt (T)	Not solved
Seq4 – Q1	Solve	Hint	Hint	Solve	Clarify	Solve	Prompt (T)
Seq4 – Q2	Solve	Solve	Hint	Clarify, Prompt (T)	Not solved	Clarify, Hint	Not solved
Seq4 – Q3	Solve	Solve	Solve	Hint	Hint	Solve	Not solved
Seq4 – Q4	Solve	Prompt (#)	Not solved	Prompt (T)	Not solved	Not solved	Not solved
Seq5 – Q1	Solve	Solve	Prompt (T)	Clarify	Solve	Not solved	Not solved
Seq5 – Q2	Solve	Solve	Clarify, Prompt (#)	Clarify, Prompt (#)	Solve	Not solved	Not solved
Seq5 – Q3	Solve	Solve	Solve	Solve	Solve	Not solved	Not solved

Solve: Programming task solved by participant independently with or without hard scaffolding.

Not solved: Programming task not solved by participant.

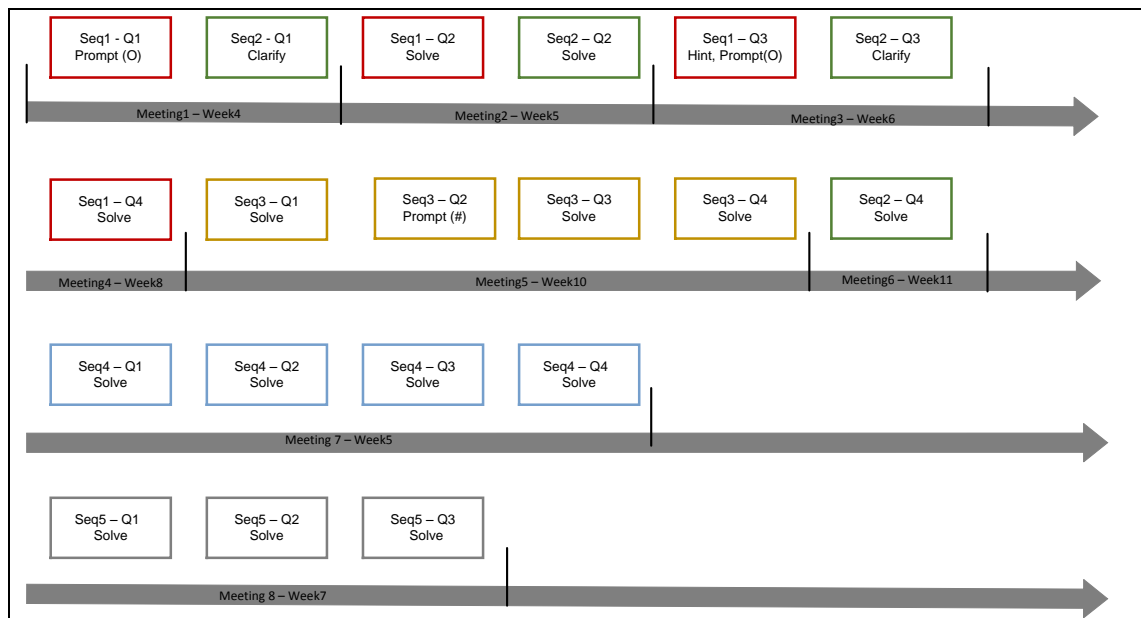
Clarify, Prompt, and Hint: Programming task solved with assistance. T for tracing scaffolding, # for stepwise refinement scaffolding, O for other types of prompts scaffolding

Appendix H. Summary of Think Aloud Recording Sessions

Table H.1 is a summary of the think aloud recording sessions undertaken during this longitudinal study of novice programmers.

Table H.1 Summary of think aloud sessions

Participants	Hours of video recoding	Hours of audio recoding
Andre	6	2
Luke	5	3
Chen	7	3
Isaac	7	4
Harry	7	3
Kasper	5	4
Matthew	6	6
Other participants	22	10



*Figure H.1 Andre's think aloud sessions*¹⁷

¹⁷ **Clarify, Prompt, and Hint:** Programming task solved with assistance. T for tracing scaffolding, # for stepwise refinement scaffolding, O for other types of prompts scaffolding.

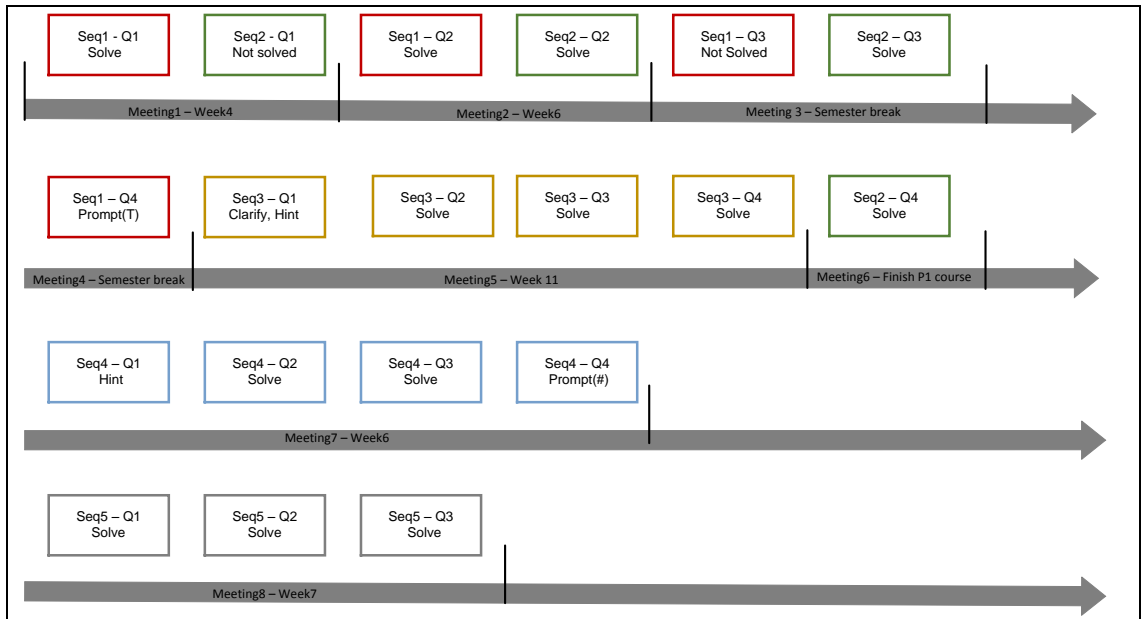


Figure H.2 Luke's think aloud sessions

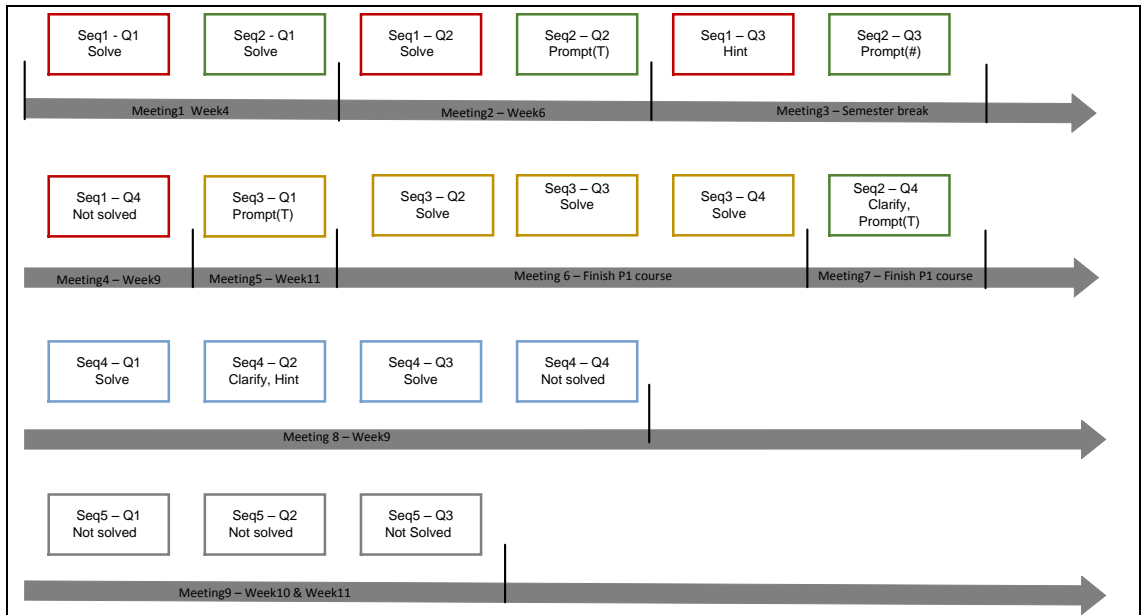


Figure H.3 Kasper's think aloud sessions

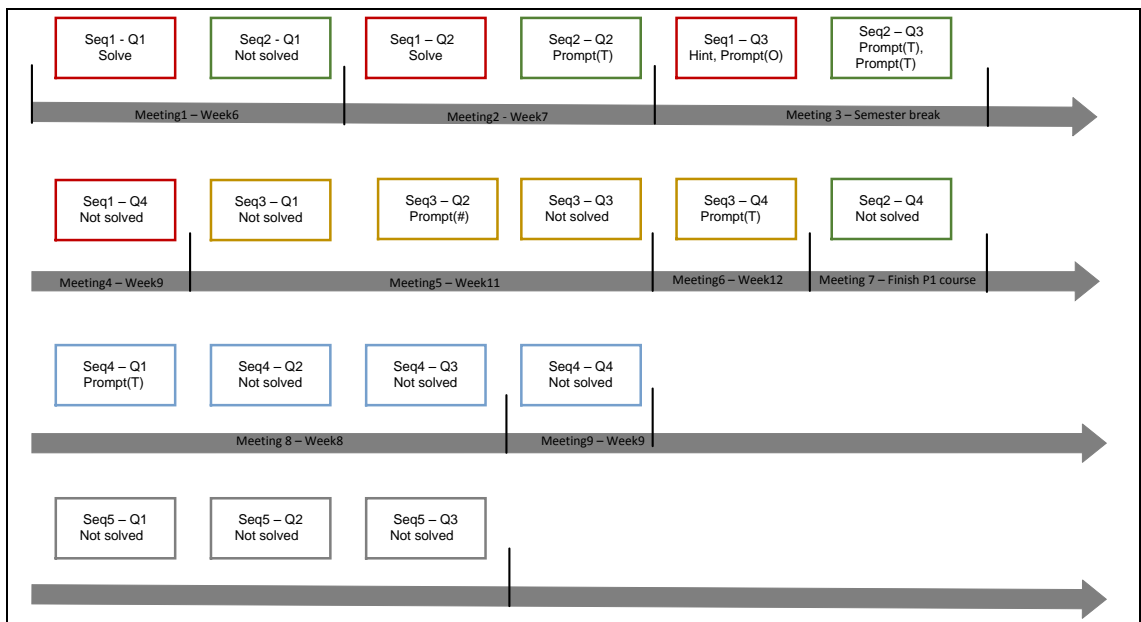


Figure H.4 Matthew's think aloud session