LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH
Ollscoil Átha Cliath

TRINITY COLLEGE LIBRARY DUBLIN
The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

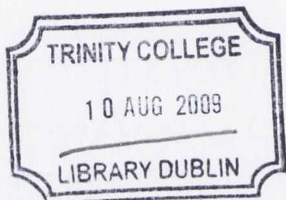
Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.



THESIS
8855

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Learning Object-Oriented Programming from the Students' Perspective

Ioanna Stamouli

A thesis submitted to the University of Dublin, Trinity College

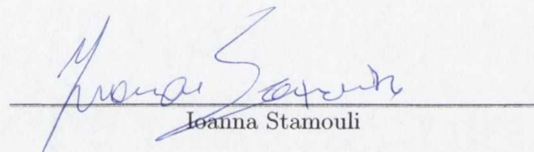
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

February 2009

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.



Ioanna Stamouli

Dated: February 11, 2009

Acknowledgements

I would like to thank my supervisor, Meriel Huggard, for the manner in which she has supervised my research and this thesis. Meriel has always been willing to let me try out new methods and ideas but always demanded justification for the choices I made. She was always there to help me concisely formulate my ideas and identify their strong and weak points. Her support, constructive criticism and patience throughout the many revisions of this thesis are greatly appreciated.

One of the most influential people throughout my life and my studies has been Dr Patroklos Argyroudis. I would not have dared dream that I could do this without his support and constant encouragement. His moral support, especially during all those times that self-doubt almost took over, gave me courage and kept me going.

Many thanks go to colleagues and friends in the Distributed Systems Group (DSG) who were always willing to listen to my ideas and research worries even if they were mostly outside their main research area. Specifically I would like to thank Neil O'Connor, Alan Gray, Shiu Lun Tsang, Andrew Jackson, Jenny Munnely and Daire O'Broin for their support and for their valuable suggestions on this thesis.

I would like to say a big 'thank-you' to all the students who agreed to be interviewed and facilitated this study. Also, I would like to thank Dr Owen Conlan for cross-validating the analysis of a portion of my findings.

Thanks also go to the many people from the Computer Science education research community who were very supportive from the very beginning in helping me with my various research questions. Their feedback on my research allowed me to refine this study while their positive attitude made me feel welcome in the community.

I wish to thank the Irish Research Council for Science, Engineering and Technology

(IRCSET) and the Higher Education Authority (HEA) for providing financial support for my studies.

Finally, I would like to dedicate this work to my family. My parents, Sakis and Maria, and my little brother Nikos, supported me in countless ways. I wholeheartedly thank them.

Ioanna Stamouli

University of Dublin, Trinity College

February 2009

Abstract

Computer programming and programming languages are core modules in most undergraduate Computer Science and Engineering degree courses. However, learning to program is a complex activity as it involves the understanding and use of abstract concepts, as well as the development of problem-solving and general programming skills. This thesis investigates how students experience learning object-oriented programming by following a group of undergraduate students in Computer Science on an introductory object-oriented programming course. In order to understand how they experience learning object-oriented programming, we have investigated: a) the theoretical, b) the object-oriented and c) the general aspects of programming from the students' perspective. The theoretical aspect concerns the students' understanding of the nature of programming as an activity, what it means to learn how to program, and their understanding of program correctness. The object-oriented aspect concerns the students' understanding of the unique constructs of this programming paradigm, namely: objects, classes, attributes, methods and constructors. The general programming aspect concerns the students' understanding of programming constructs that are common to most programming languages, specifically algorithms, arrays, iteration mechanisms and selection.

These twelve research themes form the basis of this thesis. Together they provide a more complete picture of the multifaceted process of learning to program within the object-oriented paradigm from the students' perspective. A longitudinal, qualitative study was carried out on these twelve themes and data were collected through semi-structured interviews over the course of an academic year. The analysis of the data was conducted using the phenomenographic research approach.

The analysis of the twelve themes yielded sets of categories of description reflecting the

students' qualitatively different ways of understanding object-oriented programming and programming in general. The qualitative findings are presented in their structural and referential aspects. Furthermore, they are discussed in relation to relevant research on learning and learning to program. The results depict how students understand object-oriented programming within the theoretical, object-oriented and general aspects of programming. Learning of the various components of programming is characterised by a growing awareness of the field of programming in relation to the development of qualitatively better conceptions of the constructs. The development of abstract thinking, and general programming skills, is experienced as the capability to draw from the learning environment the appropriate elements that enable deeper understanding. Additionally, the students' experience of their learning environment is studied, concluding that more practical work and discourse is necessary to acquire a complete understanding of programming and the constructs that comprise it.

Based on the findings, the study discusses the implications for educators when teaching object-oriented programming. Educators should primarily be aware of the range of understandings held by their students and actively encourage the acquisition of richer conceptions through carefully designed programming exercises and assignments. Teaching should not be limited to the expert presentation of topics, but rather should provide students with a number of ways to discern the desired understanding. Through this students will develop a holistic view of programming by becoming experts through practical work and experience. Continuous discourse and encouragement of students to express their knowledge allows them to become aware of their own understanding and, thus, identify more easily their strengths and shortcomings in learning object-oriented programming.

Contents

Acknowledgements **iv**

Abstract **vi**

List of Figures **xiii**

List of Tables **xiv**

Chapter 1 Introduction **1**

1.1 Research Questions and Objectives . . . 3

1.2 Key Contributions . . . 3

1.3 Terminology . . . 4

1.4 Dissertation Outline . . . 5

1.5 Publication Record . . . 6

Chapter 2 Related Research in Computer Science Education and Programming **8**

2.1 Computer Science Education . . . 9

2.1.1 Small-Scale Studies . . . 10

2.1.2 Cognitive Psychology Studies . . . 11

2.1.3 Learning Environments and Tools . . . 13

2.1.4 Research Based on Educational Traditions . . . 14

2.1.4.1 Constructivism . . . 15

2.1.4.2 Phenomenography . . . 17

2.1.4.3 Critical Research into Gender . . . 18

2.1.4.4 Combined Approaches 19

2.2 Research into Programming 20

2.2.1 Learning to Program 20

2.2.2 Programming Constructs 20

2.2.3 Learning Taxonomies in Programming 21

Chapter 3 Phenomenography **24**

3.1 Background 25

3.2 The Object of Learning 26

3.3 Variation 28

3.4 Data Collection and Analysis 30

3.5 Challenges of Phenomenography 32

3.6 Trustworthiness in Phenomenographic Studies 34

3.7 Why Phenomenography 38

3.8 Summary 39

Chapter 4 The study **40**

4.1 Themes of the study 41

4.2 Selecting the Course 43

4.2.1 Structure of the Course 44

4.2.2 Contents of the Course 45

4.3 Selecting the Students 47

4.4 Data Collection 48

4.4.1 Interviews 49

4.4.2 Observation 52

4.5 Transcriptions 52

4.6 ATLAS.ti 54

4.7 Analysis 54

4.8 Validity, Reliability and Generalisability in Practice 55

4.9 Summary 58

Chapter 5 The Theoretical Components of Programming **59**

5.1 Students' Understanding of Programming 60

5.1.1 Structure and Meaning of Students' Understanding of Programming 67

5.1.2 Changes in the Students Conceptions During the Course 70

5.1.3 Discussion on the Nature of Programming 73

5.2 Learning to Program 75

5.2.1 Structure and Meaning of Students' Experience of Learning to Program 83

5.2.2 Discussion on Learning to Program 87

5.2.2.1 Studies on Learning to Program 87

5.2.2.2 Studies on Learning 90

5.3 Understanding of Correctness 92

5.3.1 Structure and Meaning of Students' Understanding of Program Correctness 97

5.3.2 Learning to Program and its Relationship With Program Correctness 100

5.3.3 Program Correctness in the Literature 101

5.4 Summary 102

Chapter 6 The Object-Oriented Components of Programming **103**

6.1 Understanding Object 104

6.1.1 Structure and Meaning of Students' Understanding of Object 109

6.1.2 Discussion of the Concept of Object 112

6.2 Understanding of Class 113

6.2.1 Structure and Meaning of Students' Understanding of Class 119

6.2.2 Discussion of the Concept of Class 120

6.3 Understanding of Attribute 122

6.3.1 Structure and Meaning of Students' Understanding of Attributes . . 126

6.4 Understanding What a Method is 129

6.4.1 Structure and Meaning of Students' Understanding of Methods . . . 134

6.4.2 Functions and Methods 136

6.5 Students' Understanding of a Constructor 137

6.5.1 Structure and Meaning of Students' Understanding of Constructor . 141

6.6 Shifts in Understanding of Object-Oriented Components 143
6.6.1 Case Studies on Shifts in Understanding 145
6.7 Summary 147

Chapter 7 General Programming Components 151
7.1 On the Understanding of Algorithms 152
7.1.1 Identifying the Variations in Understanding of Algorithms 158
7.1.2 Pre-college Students' Understanding of Algorithms 161
7.2 Arrays 163
7.2.1 Arrays in Java 164
7.2.2 Students' Understanding of Arrays 166
7.2.3 Critical Aspects of The Students' Understanding of Arrays 171
7.2.4 Is an Array an Object? 172
7.3 Students' Views on Iterations 174
7.3.1 Students' Understanding of the Concept of Loops 175
7.3.2 Critical Aspect of The Students' Understanding of Loops 179
7.3.3 Loop Invariants 180
7.4 How Students Understand Selection 182

Chapter 8 Different Perspectives on the Study and its Implications 184
8.1 Relationships Between the Theoretical Components and Implications For Teaching 185
8.2 Object-Oriented Components and Implications for Teaching 190
8.3 General Programming Components 194
8.3.1 Students' Views on Algorithms and Teaching Implications 194
8.3.2 Students' Understanding of Arrays 196
8.3.3 Students' Understanding of the Mechanism of Iteration in Programming 198
8.4 The Effects of the Learning Environment 199
8.5 Impact of the Context on the Findings 203

8.6 Summary 204

Chapter 9 Conclusions **206**

9.1 Significance of this Study 208

9.2 Further Research 210

9.3 Final Remarks 211

Bibliography **212**

Appendix A Empirical study **225**

Appendix B Semi-structured interviews **230**

List of Figures

3.1 Object of study (adapted from Bowden, 2005, p.13). 26

3.2 The experience of learning (Marton & Booth, 1997, p. 85). 27

3.3 Internal and external horizons of the direct object of learning (adapted from Marton & Booth, 1997, p. 88). 30

5.1 Shifts in students' conceptions of the nature of programming. 71

5.2 An effective comparison of four phenomenographic studies on learning to program and the relationships between them. 88

5.3 Learning to program in relation to program correctness. 100

6.1 Shifts in understanding of the object-oriented concepts 144

7.1 Logical Representation of an integer array. 165

7.2 Example of an array of objects. 165

7.3 Logical premises for `while` loops in Java. 174

8.1 Relationships between the theoretical components. 187

8.2 Relationships between the object-oriented components. 192

A.1 Initial background questionnaire, page 1. 226

A.2 Initial background questionnaire, page 2. 227

A.3 Theoretical sample as compared with the class as a whole. (sample size 16, class size 40) (a) age, (b) gender, (c) educational background. 228

A.4 Consent form signed by all participants. 229

List of Tables

4.1 Themes of the study. 42

4.2 Distribution of the themes across the interviews 49

5.1 The categories of description of *programming*. 60

5.2 Understanding of programming; the focus of the conceptions. 69

5.3 Categories of description for *learning to program*. 75

5.4 Understanding of learning to program; the focus of the conceptions. 85

5.5 Marshal *et al.*, on students' conceptions of learning in an engineering context. 90

5.6 The categories of description found for learning to program in relation to the Marshall et al. conception of learning. 91

5.7 Categories of description for *understanding of correctness*. 93

5.8 Understanding of program correctness; the focus of the conceptions. 98

6.1 Categories of description for *understanding of objects*. 105

6.2 Understanding of objects; the focus of the conceptions. 111

6.3 Categories of description for *understanding of class*. 114

6.4 The focus of different understandings of the concept of class. 121

6.5 Categories of description for *understanding of attributes*. 123

6.6 The focus of different understandings of the concept of attribute 128

6.7 Categories of description for *understanding of methods*. 130

6.8 The focus of different understandings of the concept of method. 135

6.9 Categories of description for *understanding of constructors*. 138

6.10 The focus of the understandings of the concept of constructor. 142

6.11 Object-oriented framework of knowledge and its relationship to the object-oriented constructs. 150

7.1 Categories of description for the *understanding of algorithms*. 154

7.2 Aspects of variation in the understanding of algorithms. 160

7.3 Categories of description for *students' understanding of arrays*. 166

7.4 The distribution of understanding of the nature of arrays. 171

7.5 Categories of description for *understanding the nature of iterations*. 175

B.1 Outline of the first interview. 231

B.2 Outline of the second interview. 232

B.3 Outline of the third interview. 233

B.4 Outline of the fourth interview (part 1). 234

B.5 Outline of the fourth interview (part 2). 235

B.6 Exercises given to students to solve out loud during the interviews. 236

Chapter 1

Introduction

Computer programming and programming languages are core modules in most undergraduate Computer Science and Engineering degree courses. It is essential for students on such courses to acquire the necessary programming skills as quickly as possible. However, it is evident that many students find it difficult to acquire these skills, and this has a negative impact on their performance throughout their undergraduate career (Hazzan, 2002; Carter and Jenkins, 1999; Ragonis and Ben-Ari, 2005). A significant number of research studies have been carried out in this field, providing insight into a wide variety of aspects of the teaching and learning of computer programming. In particular, studies have focused on issues concerning introductory programming courses in terms of syllabus, teaching methods and paradigms; while numerous research studies investigate novice students' misconceptions and errors when learning to program. However very limited insights exist into the way students learn and understand programming, particularly object-oriented programming. In order to be able to teach effectively and help students learn programming in a better way it is necessary to understand both how they learn and how they experience the process of learning (Ramsden, 1992). In contributing to the body of knowledge in Computer Science education research, this thesis explores students' understanding of the most fundamental concepts of object-oriented programming and programming as a whole. The study adopts a positive attitude towards learning, primarily investigating students' understanding rather than their misunderstandings, while paying special attention to the variation that brings students to develop a complete appreciation of the concepts involved.

Learning to program within the object-oriented paradigm is a complex and multifaceted experience. It involves: a) a theoretical understanding of the nature of the activity, b) an understanding of the object-oriented constructs that are unique to this programming paradigm and c) an understanding of some of the basic constructs that are common to most programming languages. As this study aims to explore the overall experience of learning to program within the object-oriented paradigm, the chosen themes were selected with this in mind. The study moves from theoretical questions on the nature of programming to the more specific and unique constructs of object-oriented programming and concludes by considering some of the more general programming constructs. Specifically, the theoretical aspects of learning to program are explored in relation to students' conceptions of the *nature of programming*, *what it means to learn how to program* and their *understanding of program correctness*. The object-oriented aspects of the experience investigated are students' understandings of the constructs of *class*, *object*, *attribute*, *method* and *constructor*. The students' experience of the general programming constructs common to most programming languages are explored through their understanding of *algorithms*, *arrays*, *iteration mechanisms* and *selection*. These twelve research themes form the basis of this thesis and together they provide a more complete picture of the multifaceted process of learning to program within the object-oriented paradigm.

These twelve themes were investigated through a longitudinal qualitative study where data were collected via semi-structured interviews over the course of an academic year. Since the focus of the study is on the act of learning and the early experiences that first year university students have when introduced to an object-oriented language, the research framework used is phenomenography. Phenomenography is a research approach that allows researchers to investigate the different ways in which people understand and experience a phenomenon (Marton and Booth, 1997). It allows the researcher to go beyond causal explanations of social phenomena to understand and develop the teaching and learning in complex educational settings such as universities. The use of phenomenography as the underlying research methodology for this study provides an insight into how Computer Science students experience object-oriented programming in the three key areas identified above.

1.1 Research Questions and Objectives

The objectives of this study stem from the original research questions posed at its inception. Thus, the primary aim of this study is to explore the main conceptions undergraduate students have of the most fundamental principles of object-oriented programming. The specific focus of the study is on both the act of learning and the experiences that students have when learning to think and program within the object-oriented paradigm. The main goal of this study is to acquire an in-depth and in-breadth appreciation of students' understanding of programming within the chosen programming paradigm. The research questions that motivated the study are:

- How do students experience learning to program?
- How do students experience object-oriented programming concepts and formal definitions?
- How do students reason about specific programming problems, and programming constructs as a whole?
- How do students perceive their learning environment, and which aspects of it do they consider beneficial or redundant?
- How can students' understandings be used to enhance teaching and therefore improve the quality of learning?

Although these questions were central to this study from the very early stages, they have naturally evolved and have been refined as it progressed.

1.2 Key Contributions

The key contribution of this thesis is an in-depth exploration of first year Computer Science students' understanding of the theoretical, object-oriented and general components of programming. The key contribution of this work is as follows:

- A valuable insight is gained into students' overall understandings of computer programming, providing an un-abbreviated picture of the experience of first year programming which ranges from its theoretical to its technical aspects.

- The results of this study can be used to create an awareness of the educationally critical aspects of learning to program. These aspects have been identified by the learners themselves.

- The findings capture the levels of understanding attained by the students, and thus may be used to enhance the teaching of key programming concepts with the goal of improving student learning.

- The research project contributes to the teaching and learning of programming, as it employs a theoretically anchored research approach to studying students' understanding and can therefore be applied to similar educational settings.

- The project as a whole constitutes a self-contained piece of work within the phenomenographic tradition, a research approach that is continuously evolving.

1.3 Terminology

This section provides a short list of terms that are used frequently throughout the thesis, along with their definitions.

- *Theme(s)* and *construct(s)* are used as synonyms and refer to the phenomena that are investigated in this study. For example, *"learning to program"* is referred to as a theme of this study, while *"class"* is referred to a construct or a programming construct.

- *Conception*, *category of description* and *way of experience* are phenomenographic terms that are used to denote different aspects of students' understanding of the phenomenon under investigation.

- *Understanding* and *experience* are also used as synonyms to denote the students' view of the phenomenon that is in focus.

1.4 Dissertation Outline

This chapter lays out the scope and motivation of the study in broad terms. It also summarises the research questions, objectives and key contributions this study aims to achieve. The structure of the remainder of this work is outlined below.

Chapter 2 presents the field of Computer Science education, of which this study is a part. We discuss the different divisions within the field, alongside notable individual research projects. Greater emphasis is placed on qualitative projects related to programming as these are directly relevant to the research presented in this study. This discussion helps to provide a direct insight into relevant research in the field.

Chapter 3 presents the research approach that is fundamental to this thesis: phenomenography. The theoretical and methodological aspects of this research approach are presented. The guidelines and practises that are employed in phenomenographic projects are also discussed alongside trustworthiness issues that are relevant to all qualitative research studies.

Chapter 4 contains the design of the empirical study, along with detailed accounts of the general methodological considerations taken into account during data collection and the selection of the educational setting. The choices made in both the design of the study and in the analysis of the data are also described.

In Chapters 5, 6 and 7 the analysis of the data collected on the theoretical, object-oriented and general programming components is presented. The qualitatively different ways that students experience the themes of the study are discussed individually with the use of supporting interview excerpts. The themes are then discussed as a whole and the relationships between the categories highlighted. The structural and referential aspects of the conceptions are then further analysed in accordance with phenomenographic guidelines. The variations in awareness that bring out the conceptions are also presented; illustrating the dimensions of variation on which the categories are based. The presentation of each theme is immediately followed by a discussion section where the results are compared and related to the literature in the field. Hence the contribution of the findings for each theme can be immediately assessed in the context of the field.

In Chapter 8 we review the findings and further discuss their relationships in order to

present the full picture of what it means to learn how to program. The effects of the learning environment are also explored based on the students' experience of the elements that had a positive impact on their learning, and their suggestions for improving other aspects of their learning environment. Based on the overall findings of the study, a number of logically-based suggestions are presented for improving teaching and the learning environment.

The thesis concludes with a discussion of the contributions of this study and suggestions for further work.

1.5 Publication Record

As sections of this work were completed the findings were submitted for publication in refereed conferences proceedings and journals. These are listed below, together with other related studies the author contributed to.

- Enda Dunican and Ioanna Stamouli, "Grounded theory and Phenomenography: what, how, and when to use them", Tutorial session in the annual PPIG (Psychology of Programming Interest Group) workshop, Joensuu, Finland, July 2007.

- Ioanna Stamouli and Meriel Huggard, "Object Oriented Programming and Program Correctness: The Students' Perspective", In Proceedings of the 2nd International Computing Education Research Workshop, Kent, Canterbury, UK, 9-10 September, 2006.

- Ioanna Stamouli and Meriel Huggard, "Learning Object Oriented Programming from the Students' Perspective", In Proceedings of EARLI JURE 9th Conference, Estonia, Tartu, July 2006.

- Alan Gray, Andrew Jackson, Ioanna Stamouli and Shiu Lun Tsang, "Forming Successful eXtreme Programming Teams", In Proceedings of the IEEE Agile International Conference, Minneapolis, Minnesota, USA, 23-28 July 2006.

- Ioanna Stamouli, "On Learning Object Oriented Programming", Poster abstract, In JURE – Junior Researchers of EARLI, 11th Biennial Conference, Cyprus, August 2005.

- Norman Reid, Rebecca Mancy, Ioanna Stamouli, Colin Higgins and Marjahan Begum, "ExploreCSEd: Exploring Skills and Difficulties in Programming Education", In Proceedings of the 6th Annual Conference for the Higher Education Academy Subject Network for Information Computer Science, York, UK, 30 August-1 September 2005.

- Ioanna Stamouli, Marjahan Begum and Rebecca Mancy, "ExploreCSEd: Exploring Skills and Difficulties in Programming Education", Poster abstract, In Proceedings of the 10th Annual Conference on Innovation and Technology in Computer Science Education, Universidade Nova de Lisboa, Monte de Caparica, Portugal, 27-29 June 2005.

- Eileen Doyle, Ioanna Stamouli and Meriel Huggard, "Computer Anxiety, Self-Efficacy, Computer Experience: an Investigation throughout a Computer Science Degree", In Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference, 2005.

- Ioanna Stamouli, Eileen Doyle and Meriel Huggard, "Establishing Structured Support for Programming Students", In Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference, 2004.

Chapter 2

Related Research in Computer Science Education and Programming

This chapter describes the field of Computer Science education research and provides a detailed presentation of the domain in which the work is situated. As Berglund et al. (2006) point out about this field of research: *"Computing education research (CER) is a cross-disciplinary field comprising computing - of course - as well as a wide range of other disciplines: pedagogy, psychology, cognitive science, learning technology, and sociology [...]"*. Thus, there is an extremely wide variety of research directions within the field of computer education research, spanning from methods related to naturalistic science conducted in the quantitative research tradition, to qualitative research approaches anchored in theoretical methods that are directly related to pedagogy. However, irrespective of the research traditions that a project follows, the main goal of the field of computer education research is to improve learning and teaching within the discipline and contribute to the development of Computer Science as a whole.

In this chapter we present and discuss educational research that is related to Computer Science and which aims to improve and affect the teaching and learning of that discipline. Although the work presented in this thesis follows a qualitative research approach, both quantitative and qualitative projects are discussed in this chapter to illustrate the difference in nature of such studies. As our study relates to computer programming, greater emphasis is given to research projects in this area.

2.1 Computer Science Education

Research into Computer Science education, and specifically into learning to program, is very diverse in terms of the paradigms taught, methodologies adopted, perspectives considered and focus groups established. In (Clancy et al., 2001), the contribution by Fincher discusses the research practises and communities that have formed within Computer Science education research. The article stresses the need for both new and experienced researchers in the field to find, and get involved with, the established Computer Science research communities and contextualise their research within these settings. Therefore, it is reasonable to expect that different types of research projects exist within the subject area. Fincher identifies four broad areas that characterise different types of research in the field with regard to their content, focus, practises and communities (Clancy et al., 2001). These are:

1. Small-scale investigations of a single aspect of the discipline or practise. As Fincher says "these are often found at SIGCSE-sponsored conferences such as SIGCSE *(ACM Special Interest Group Computer Science Education)*, ITiCSE *(Innovation and Technology in Computer Science Education)* and the ACE *(Australian Computing Education)* conferences".

2. Investigations of specific mental and conceptual skills. Fincher says "these are often found at PPIG *(Psychology of Programming Interest Group)* and ESP *(Empirical Studies of Programmers)*".

3. Investigations based within the educational tradition. These are presented at educational conferences.

4. Investigations motivated by the use of tools in Computer Science teaching and learning. (Fincher in Clancy et al., 2001, pp. 338-339)

There have been a number of other proposed categorisations of Computer Science education research. Greening divides research in this field into *positivistic research into learning Unix*, *interpretivistic research into students' conceptions* and *gender representation in Computer Science* (Greening, 1996). Holmboe et al. also identified six types of studies: *new*

untested ideas, *reports from the trenches*, *discussion of theory*, *computer aided learning*, *expert/novice differences*, and *empirical studies* (Holmboe et al., 2001). More examples of such characterisations of the research conducted in Computer Science education can be found in (Carbone and Kaasbøll, 1998), (Berglund et al., 2006) and (Fincher and Petre, 2004). However, the four categories presented by Fincher in (Clancy et al., 2001) provide a more complete, general categorisation and are more appropriate for the purposes of this chapter. In the remainder of this section the four categories presented above are further analysed with respect to their contribution and aims. As the work presented in this thesis falls within the third category, more emphasis is placed on its description.

2.1.1 Small-Scale Studies

Small-scale investigations are usually performed by classroom practitioners in Computer Science education. Such studies identify problems and report on the experience of teaching courses in the subject area. Reports on these investigations usually appear in the proceedings of conferences such as SIGCSE and ITiCSE[1] and usually take the form of case studies. Educators in Computer Science have first-hand experience of problems that may arise in a course, such as high failure rates, retention, difficulty in learning or teaching specific constructs. These ideas are put forward and possible solutions in relation to the delivery of a course or a change in the structure of a degree program are analysed and evaluated. Methods for gathering data include pre- and post-questionnaires, student examination grades and other methods as reported in (Carbone and Kaasbøll, 1998).

As these studies address problems that have arisen in real teaching situations, the motivation for conducting them usually stems from the desire of computing educators and their institutions to enhance the quality of teaching and learning. Small-scale studies usually provide interesting, insightful observations and ideas on how to teach a particular course (Holmboe et al., 2001). Sharing these among the educational community is crucial as it allows for the development, and further evolution, of Computer Science education.

During her Ph.D. studies, the author was involved in three such projects. In (Stamouli

[1]SIGCSE: The ACM Special Interest Group on Computer Science Education, http://www.sigcse.org/. ITiCSE: Conference on Innovation and Technology in Computer Science Education organised by the ACM Special Interest Group on Computer Science Education (SIGCSE), in co-operation with SIGACCESS (the ACM Special Interest Group on Accessible Computing), http://iticse2007.computing.dundee.ac.uk/.

et al., 2004) the development of a Programming Support Centre, an initiative within the Computer Science department in Trinity College Dublin, is presented. The evaluation of the centre's overall performance, through the use of questionnaires, illustrated that it had a positive impact on the students' learning experience. In (Doyle et al., 2005) we investigated whether Computer Science students of Trinity College suffer from computer anxiety and low self-efficacy and we explored the relationships between these factors. Data for this study were gathered from students in all four years of the degree program. The statistical analysis of this data revealed a positive correlation between prior experience and self-efficacy, and a negative correlation between anxiety and experience, thus verifying the initial hypothesis of the study. The third study, which was funded by the U.K. Higher Education Academy on Information and Computer Science, focused on investigating the skills and difficulties involved in learning to program by gathering data from students and educators in multiple institutions (Stamouli et al., 2005). Further discussions on the findings of this study are continued in (Reid et al., 2005).

Generally, small-scale studies that deal with specific teaching and learning problems are valuable for propagating ideas and sharing experiences. However, these projects are carried out with very specific populations and through experimental evaluation tools such as questionnaires and exam grades (which vary from one institution to another) and are, therefore, hard to generalise. As (Carbone and Kaasbøll, 1998; Berglund et al., 2006) and (Holmboe et al., 2001) point out, they are useful as a basis for discussion, but, in many cases, they do not provide strong, generalisable results.

2.1.2 Cognitive Psychology Studies

From the very early stages of its development Computer Science education research has been heavily influenced by the field of cognitive psychology. Cognitive psychology studies that focus mainly on expertise, proficiency, a general exploration of knowledge structure and knowledge acquisition (Holmboe, 2005; Berglund, 2005) fall into the same classification. Research in this area employs quasi-psychological experiments which aim to describe the characteristics of experts in a specific domain. The use of cognitive style metrics is an example of this. These metrics investigate the variation among individual levels of

achievement based on the individual's mental models. Cognitive dimensions and patterns focus on the interactions between a physical form and the way in which that form inhibits, or facilitates, various kinds of personal and social behaviour (Byrne and Lyons, 2001). According to Green, the cognitive dimensions framework is mostly used in evaluating the effectiveness of various techniques for interactive devices and for non-interactive notations (Green, 1989).

Evidence suggests that cognitive style is independent from other constructs like intelligence, personality and gender, and that it relates to and influences a range of behaviours (Riding and Rayener, 1998). These behaviours include learning performance, social responses and occupational stress. In "Matters of Style", Felder defines cognitive styles as the characteristic strengths and preferences in the ways a person takes in and processes information (Felder, 1996), whereas a learning strategy reflects the actual processes used by a learner to respond to the demands of a learning activity (Riding and Rayener, 1998). Since the mid-1940s, there have been various influences on this field, which led to the emergence of numerous models for measuring cognitive styles. This has resulted in an extensive list of labels for very similar descriptions of styles. This makes the comparison of research studies in this field nearly impossible since they use different cognitive style metrics or different labels for representing them. Some researchers attempted to categorise all these existing style labels into families, but they did not succeed in creating a common terminology for the field.

Programming and learning to program have been extensively explored within this research tradition (Clements and Gullo, 1984; Mayers, 1981; Sime et al., 1973). Most of these studies investigated the learners' styles for different population samples and programming languages; some looked at the various dimensions of cognitive style. Although, as (Bishop-Clark, 1995) points out, some of the results are comparable, there is no conclusive evidence that demonstrates which cognitive styles or learning strategies perform better in programming courses.

A notable study is (Weinberg, 1971), which is considered by many *"the first major contribution in the field of Computer Science education within the cognitive psychology tradition"* (Holmboe, 2005). Later, (Brooks, 1977) used the theory of long- and short-term

memory as the framework for analysing expert programmers' understanding and behaviour with regard to identifying methods and general coding norms. McIver in (McIver, 2000) investigated both syntactical and logical errors made by novice programmers by comparing students who learnt two different procedural languages (namely LOGO and Grail). The results show that the underlying programming language highly affects the rate of both logical and syntactical errors made by students (McIver, 2000). Additionally, (Robins et al., 2003) presented an overview of programmers needs, focusing on the characteristics and differences of effective versus ineffective novice programmers. A substantial amount of work on cognitive psychology and programming can be found in (Hoc et al., 1991), while a critical evaluation of this tradition is outlined in (Détienne, 2002).

Generally, it can be said that research in this area belongs to both cognitive psychology and Computer Science education. Studies in this category explore the learner and his behaviour, independent of the environment and the object of study. Nonetheless as Berglund et. al point out *"this type of research has contributed both through its results and by the rigor by which many research projects have been carried out"* (Berglund et al., 2006).

2.1.3 Learning Environments and Tools

The field of animation and visualisation and, more generally, the development of learning environments and tools, constitutes another bridge between disciplines; namely between the fields of Computer Education, Human Computer Interaction (HCI) and Computer Supported Collaborative Learning (CSCL). Research in this area is quite broad and is motivated by the desire to create change through the use of tools and environments that increase the effectiveness of teaching and learning. The span of this field ranges from algorithmic animations such as Pavane (Romero et al., 2003) and Jeliot (Jeliot-Team, 2006) to novel programming platforms and languages such as BlueJ (Kolling, 2006). The latter aims to help the understanding of programming constructs and hence reduce cognitive load during programming activity. Collaborative learning environments such as COOL (Berge et al., 2003) have been designed to facilitate learning of object-oriented programming through graphical languages such as BlueJ.

This is an important sub-field of Computer Science education which aims to enhance

and improve learning. Its outcomes are particularly relevant in situations with limited teaching resources where it provides a means for encouraging collaboration.

2.1.4 Research Based on Educational Traditions

There are a number of research traditions or theories that can be applied to the exploration of students' understanding and learning of topics in Computer Science. Cognitive psychology, as discussed above, is one such research tradition. These research traditions may be broadly separated into positivistic and anti-positivistic (non-positivistic[2]) in relation to their epistemology. However, many research studies use some combination of these two approaches (Wellington, 2000). The positivistic approach, also called objectivism, considers the truth as hard evidence that is external to the individual, while the anti-positivistic approach challenges this by arguing that there is no, one objective truth (Cohen et al., 2000). Positivistic, quantitative researchers usually employ surveys, experiments, and questionnaires in data gathering and analysis, like many of the small-scale projects presented in Subsection 2.1.1. Anti-positivistic, qualitative researchers *"[...] view the social world as being a much softer, personal and humanly created kind, and will select from a comparable range of recent and emerging techniques - accounts, participant observation and personal constructs, for example"* (Cohen et al., 2000, p. 7). While the two approaches outlined above seem to be polar opposites, they can, and often do, complement each other. Most data collection methods in educational research yield both qualitative and quantitative data, thus allowing for the triangulation of results.

Research that is based within either of these educational traditions is very important as it allows for a better categorisation of relevant studies, and hence for the creation of research communities. As (Berglund et al., 2006) point out *"[e]xplicit adoption of a research approach also facilitates communication with other researchers. A shared terminology becomes available and enables the researcher to learn from others and to judge and compare different projects"*. Thus the results of a project that is conducted using one particular educational tradition may be relevant within the respective community. Moreover, it may

[2]The term *anti-positivistic* research appears in (Cohen et al., 2000), while (Berglund et al., 2006) refer to the same research paradigm as *non-positivistic* and (Gall et al., 1996) call it *postpositivistic*. Even though there are differences between the exact definitions of these, they will not be discussed further in this section as they are beyond the scope of this chapter.

be further compared to other projects, and also critically evaluated as to its validity and generalisability. Comprehensive reviews of projects in this area can be found in (Berglund et al., 2006; Carbone and Kaasbøll, 1998; Holmboe, 2005) and (Fincher and Petre, 2004).

The distinction between quantitative and qualitative research is an important one as it influences which methods a researcher applies and the questions that can be explored within that paradigm (Berglund et al., 2006). Although there is a distinction between qualitative and quantitative research, this does not imply that one is better than the other. Rather it means that they may be used, and are appropriate, for addressing different types of research questions. As (Gall et al., 1996) argue *"[..] qualitative research is best used to discover themes and relationships at the case level, while quantitative research is best used to validate those themes and relationships in samples and populations. In this view, qualitative research plays a discovery role, while quantitative research plays a confirmatory role"* (Gall et al., 1996, p. 29). In (Gall et al., 1996) the distinction between quantitative and qualitative research is discussed in detail. The work presented in this thesis is based on phenomenography, a qualitative research approach, so in the subsections that follow we focus on the evolution of, and trends within, the qualitative, anti-positivistic, area.

2.1.4.1 Constructivism

Research within the constructivist paradigm is diverse in focus. The main claim in constructivism is that knowledge is actively constructed by the learner (Fincher and Petre, 2004). Researchers in this field investigate issues such as the influence of learning environments, the effects of teaching in particular ways and, generally, the role of the learner and others in the construction of knowledge (Berglund, 2004).

Constructivism is based on a number of educational theories. Bruner, in (Bruner, 1960), built upon the individual cognitive schemes outlined by Jean Piaget (Piaget, 1954), to highlight the importance of emphasising the relationship of cognitive structure and the structure of a subject (Fincher and Petre, 2004). Alongside this work, Vygotsky in (Vygotsky, 1986) formulated the notion that *"knowledge and learning are culturally and societally constructed"* (Fincher et al., 2004, p. 35). In this theory, learning is related to, and is part of, the student's environment, thus studies that are conducted within this area

focus on the environment of the learner, along with the use of teaching tools and instruction that affect and facilitate the construction of knowledge by the learner. A comprehensive account of constructivism can be found in (von Glasersfeld, 1995) and (Ben-Ari, 2001).

From the point of view of this thesis, one of the most notable studies within the constructivist framework in Computer Science education is (Ben-David Kolikant, 2001) where the researcher investigated the prior knowledge of high school students who are taking a concurrent and distributed programming course. The study focuses on misconceptions that originate from prior knowledge. Amongst other findings, the results indicate that the students invent conceptual models as they work through an exercise. The work provides suggestions on how the design of a concurrent and distributed systems course can be improved, for instance it stresses the need to take students' prior knowledge into account when teaching new material. A later study (Ben-David Kolikant and Pollack, 2004), investigated the students' beliefs of a correct solution. The findings of the study suggest that students are preoccupied with producing a working program and are tolerant of errors (Ben-David Kolikant and Pollack, 2004). However, these beliefs are not common within the profession of Computer Science, highlighting the need for course changes that will allow for the adaptation of professional norms in programming courses.

Another study in programming that was conducted within the constructivist framework is (Fleury, 2000). This study investigated how novice programmers *"construct an understanding of the syntactical and semantic rules involving the construction and use of objects in Java"* (Fleury, 2000). The findings of this study suggest that students construct their own understanding when learning to program and therefore many students possess partial understanding, even in cases where they are provided with a complete set of information on a construct. To overcome this, Fluery suggests that playing and practising with programs allows students to increase their current level of understanding. Thus discourse at the students' level of understanding can assist them in further building upon their knowledge.

Aharoni (Aharoni, 2000) studied the thinking processes that occur in the minds of students when learning and using data structures. The methods he used were based on constructivism but also focused on the mental models developed by the students. The findings suggest that while data structures should theoretically assist students in developing a

high level understanding of abstraction, this is not the case. To avoid this situation, Aharoni suggests that students should first gain an understanding of the abstraction involved in data structures before being introduced to their implementation. Like Fleury, he argues that *"The students should get assignments from the teachers, so they can "play" with the DS, much like they would do with concrete objects"* (Aharoni, 2000). Hazzan also discussed how students cope with abstraction in mathematics and in Computer Science (Hazzan, 2003), emphasising the need for students to reach a high level of abstraction when learning to program.

Holmboe (Holmboe, 1999; Holmboe, 2000) discusses the different types of knowledge in Informatics. He argues that *"mere understanding has no value without the skills to implement it, and the skills alone, though useful in many situations, can not be seen as knowledge unless accompanied by a mental understanding of the concept at hand"* (Holmboe, 1999, p. 17). Finally, (Box and Whitelaw, 2000) conducted a study within the constructivist theory framework where they concluded that object-oriented technology involves difficult cognitive processes of abstraction; thus highlighting the need to actively encourage students to develop abstract thinking skills.

Many more studies exist within the constructivist paradigm; those presented above were selected for their relevance to Computer Science and programming in particular. In a classic constructivist study Ben-Ari (Ben-Ari, 2004), discusses the implications of the environment on knowledge construction in Computer Science. However, all the projects discussed in this section focus on developing models of the knowledge that the students construct when learning concepts within Computer Science and programming. As the environment within which the students are taught affects the development of knowledge, many studies look at ways of improving it in order to minimise the development of misconceptions when learning Computer Science and programming constructs.

2.1.4.2 Phenomenography

Phenomenography is the qualitative, empirical research approach that is central to this study. The most common and widely used definition of phenomenographic projects comes from (Marton and Booth, 1997). This states that the aim of such projects is to explore a

phenomenon from the students' perspective and to identify the qualitatively different ways in which this phenomenon is experienced, understood, or perceived. As phenomenography is fundamental to this study, this research approach is further discussed in terms of both its methodological and theoretical aspects in Chapter 3.

A number of phenomenographic studies have been carried out in the field of Computer Science. For her Ph.D. thesis Booth (Booth, 1992) studied what it means to learn how to program within a functional programming course. She also investigated students' experience of other elements of procedural programming such as recursion and functions. Bruce et al. also conducted a phenomenographic study into how beginners experience learning to program in an Information Technology course (Bruce et al., 2004). Both studies suggest that the constructs of a programming language should be introduced to the students in as many different ways as possible to encourage variation and enable students to experience programming from all it perspectives. Eckerdal (Eckerdal, 2006) investigates how students experience the constructs of object and class in object-oriented programming within an Engineering course. In his Ph.D. thesis (Berglund, 2005) used phenomenography and activity theory to investigates students' understanding of computer networking protocols. Cope (Cope, 2000), in his Ph.D. thesis, looks at students' understanding of information systems; highlighting the complexity of the constructs and the different levels of understanding that students should achieve. Many more studies have used phenomenography as their research framework, such as (Akerlind et al., 2005), but those detailed above are the ones which explore learning within the field of Computer Science.

2.1.4.3 Critical Research into Gender

Critical research or enquiry is motivated by imbalances that may occur in educational and other settings. The critical research paradigm actively attempts to reveal and address imbalances that can be the effect of, among many other factors, gender, culture, educational background, and environments (Greening, 1996; Berglund et al., 2006). Computer Science and Engineering are two fields that are highly male dominated, thus research into gender issues is highly relevant as it highlights the problem of this bias within the discipline and proposes solutions for addressing it.

Recent research into gender issues in Computer Science shows that the small number of females choosing Computer Science degrees is affected by low confidence in their computing abilities, lack of programming and hands-on computer experience, and negative stereotypes regarding the field (Beyer et al., 2005). Berglund et al. (Berglund et al., 2006), in reference to (Björkman, 2002), point out that it is important for gender research to be performed from within the discipline itself because in order to be effective any changes needed will have to come from within the discipline.

There is a large body of critical enquiry research within the Computer Science education field. An overview of these projects and their effects in the field is provided by Clear in (Fincher and Petre, 2004). The critical research paradigm is very important as it challenges those in academia to rethink and reflect on the teaching and learning paradigms that are associated with Computer Science.

2.1.4.4 Combined Approaches

As stated above, qualitative and quantitative research methods are typically used to explore different facets of research questions. However, these research methodologies are complementary: generally, qualitative research can be used to discover themes and relationships between them, while quantitative research can be used to validate these. Quantitative data, such as background statistics can set the scene for an in-depth qualitative study (Wellington, 2000). Therefore, the combination of these two approaches in studies can provide stronger evidence of the generalisability and validity of the research outcomes. The triangulation of the data gathered helps to illuminate more facets of the research question under consideration. Hence, the combination of the two approaches often presents a more complete picture of learning and teaching (Greening, 1996).

An example of a combined approach study is (Perrenet et al., 2005). Initially this employs quantitative methods to explore students' understanding of algorithms while later in (Perrenet and Kaasenbrood, 2006) the findings are cross-validated with a qualitative analysis of interview data. Another example is (Kinnunen and Malmi, 2004) who related the results of their study to interview data, questionnaire data and course grades. Their study explored how students approached learning to program in a problem-based course

and it identified effective and ineffective groups. Finally, in (Gray et al., 2006) it is argued that the levels of engagement with, or "buy-in" into the ethos of the eXtreme programming methodology is a significant determinant of a projects success. The researchers investigate how team formation influences "buy-in" and how "buy-in", in turn, effects success, learning and working attitude within an academic environment. They employ both quantitative data from surveys, project grades and qualitative interview data and conclude that teams that comprised people with similar understandings of, and attitudes towards, the process, have a higher level of "buy-in" and, as a result, exceed their expected performance.

2.2 Research into Programming

Among the studies presented above, a significant number focus on programming as this appears to prove challenging for many students of Computer Science. In this section we summarise research studies that focus on programming, and therefore are related to the themes of this thesis.

2.2.1 Learning to Program

The question of how students experience programming have been investigated in three other studies namely (Booth, 1992; Bruce et al., 2004) and (Eckerdal, 2006). These studies use phenomenography as the research framework for their analysis and are conducted in educational settings other than Computer Science. Their findings illustrate a number of categories of description that summarise the qualitatively different ways that students experience computer programming. As the studies were conducted at different times and in diverse educational settings, there are, naturally, a number of similarities and differences in their outcome space. These, along with the finding of this study are further discussed, analysed and compared in Chapter 5, 6, and 7.

2.2.2 Programming Constructs

The constructivist paradigm that is dominant within Computer Science education research is based on the premise that knowledge is actively constructed by the students. Thus most studies that focus on the various constructs of computer programming, either within the

object-oriented or procedural programming paradigms, focus on the misconceptions and lack of understanding that students exhibit. Holland et al. (Holland et al., 1997) identified and described a number of misconceptions that are observed in students' learning about object technology. The results of this study provide a number of simple and concrete measures that educators can take to avoid these misconceptions arising in a distance education course. Another relevant study comes from (Ragonis and Ben-Ari, 2005) where a long-term investigation explored the conceptions and misconceptions students build when learning to program.

In this study, we adopt a positive perspective by investigating the understandings, rather than the misunderstandings, that students have of some of the most fundamental concepts of object-oriented programming. Therefore, it is not always possible to relate the results found in this study with the outcomes of constructivist studies. Nonetheless, in the discussion sections of Chapters 5, 6, and 7 the results of studies that are related to the themes investigated in this thesis are further presented and discussed. Thus, where possible, comparisons with previous work in the literature are provided so that the reader may place the findings of the study within the appropriate context.

2.2.3 Learning Taxonomies in Programming

A popular educational framework is Bloom et al.'s taxonomy of educational objectives that was devised in the 1950's as a generic instrument for dividing the cognitive aspects of learning into hierarchical levels (Bloom et al., 1956). Throughout the years it has been, and still is, widely used in course design in higher education, as a way of ensuring that teaching and assessment achieve an appropriate balance between learning of content and the development of high level skills. The learning taxonomy devised by Bloom et al. divides the cognitive aspects of learning into six hierarchical levels:

- Knowledge (recall of facts, et cetera)
- Comprehension
- Application
- Analysis

- Synthesis
- Evaluation

Recent re-evaluation of Bloom et al.'s taxonomy by Anderson and Krathwohl has suggested that the top two or three levels of the hierarchy may be flat (Anderson and Krathwohl, 2001). They have also proposed that the taxonomy should be two dimensional, with the (slightly reconfigured) original categories of Remember, Understand, Apply, Analyze, Evaluate and Create forming the cognitive process dimension and Factual, Conceptual, Procedural and Meta-Cognitive forming a knowledge dimension (Anderson and Krathwohl, 2001). The computer science education literature contains a small number of examples of the use of a taxonomy as an analytic tool. Bloom et al.'s taxonomy in programming has been applied in course design in (Scott, 2003), while Lister and Leaney have used it for structuring assessments in an introductory programming course (Lister and Leaney, 2003).

Another taxonomy that has been used in measuring the learning outcomes in programming courses is Biggs and Collis taxonomy; the Structure of the Observed Leaning Outcomes (SOLO) taxonomy (Biggs and Collis, 1982). SOLO can be used to set learning objectives for particular stages of learning or to report on the learning outcomes, as it charts the increasing structural complexity of student learning outcomes. The taxonomy identifies that learning initially changes quantitatively, as the amount of detail in the students response increases, and then qualitatively, as the detail becomes integrated into a structural pattern (Johnson and Fuller, 2006). Lister et al. have used the taxonomy to describe how code is understood by novice programmers, and describe the five SOLO levels applied to novice programming as follows (Lister et al, 2006, p. 119):

- *Prestructural:* "In terms of reading and understanding a small piece of code, a student who gives a prestructural response is manifesting either a significant misconception of programming, or is using a preconception that is irrelevant to programming."
- *Unistructural:* "[T]he student manifests a correct grasp of some but not all aspects of the problem. When a student has a partial understanding, the student makes [...] an "educated guess""

- *Multistructural:* "[T]he student manifests an understanding of all parts of the problem, but does not manifest an awareness of the relationships between these parts".

- Relational: "[T]he student integrates the parts of the problem into a coherent structure, and uses that structure to solve the task.

It has to be noted that both Bloom et al.'s and Biggs et al's SOLO taxonomies are general theories, which were not originally designed to be used in a programming context. However, both taxonomies have been used and adapted within programming education, providing some illuminating findings mostly focused on the how students understand code in order to improve assessment and teaching. Recent examples of work in this area include (Fitzgerald et al., 2005; Lister and Leaney, 2003; Lister et al., 2004; Lister et al., 2006; Mannila, 2006; Whalley et al., 2006).

Chapter 3

Phenomenography

This chapter describes phenomenography, the research approach, at the core of the work presented in this thesis. This is an empirical study that aims to investigate the ways students understand and experience a number of theoretical, object-oriented and general programming constructs. Thus, phenomenography is a suitable research framework for this endeavour.

In the previous chapter a number of research projects that explored learning and various aspects of programming were described. Some areas of programming, such as novice programmers' errors and misconceptions, have been researched at great length whilst in other areas there is little or no existing research. The goal of this study is to take learning to program within the object-oriented paradigm as a general framework and then to conduct an in-depth and in-breadth investigation into the ways Computer Science students understand and experience critical aspects of learning to program within that framework. Due to the great breadth of the study in terms of the phenomena analysed, a more complete picture of students' experience of object-oriented programming will be obtained.

It is important to note that phenomenography is neither a method in itself nor is it only a theory of experience (Marton and Booth, 1997). However, there are methodological elements that are associated with it, as well as theoretical elements that are derived from it. Phenomenography is a way of identifying, formulating and investigating certain research questions that are related to learning and understanding in an educational setting (Marton and Booth, 1997). Therefore it is referred to as a *research approach* that combines

methodological and theoretical elements.

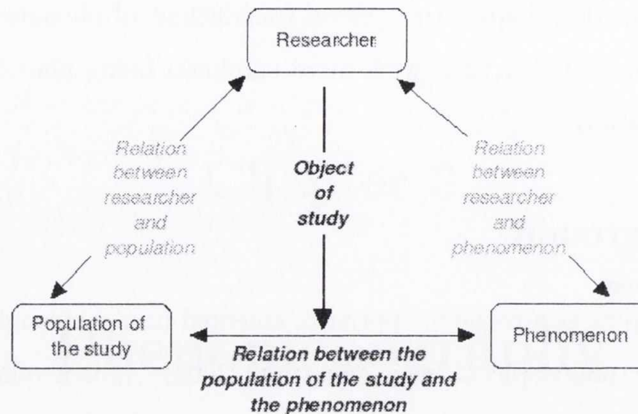
In the rest of this chapter the general foundations of phenomenography as a research approach are presented briefly, with more emphasis being placed on the issues that are related to this study.

3.1 Background

Phenomenography, as a research approach, emerged from a strongly empirical, rather than a theoretical or philosophical basis (Akerlind, 2002). It is a relatively new research approach that emerged in the 1970s in Gothenburg, Sweden from the work of Ference Marton, Lennart Svennson, Lars Owe Dahlgren and Roger Säljö. Based on the general observation that some people are better learners than other, this group of researchers carried out a number of phenomenographic projects on learning that clearly showed that there is a variation in *what* people understand and also in *how* they understand it. Thus, phenomenographic research is defined in terms of the *object of study*, which is the phenomenon under investigation, and in *what* or *how* people are experiencing it (Marton, 1981). This is well defined in (Marton and Booth, 1997) where they say *"A phenomenographic research project reveals the qualitatively different ways in which a phenomenon can be experienced, understood or perceived by a student cohort".*

Phenomenographic research projects most frequently focus on mapping variations in the population's experience. These experiences are then described in terms of a range of qualitatively different ways of understanding a particular phenomenon in the form of categories of description. The relationships between these categories of description are also highlighted in terms of their inclusiveness and encapsulated understanding. In later studies there has been great emphasis on identifying the structure of awareness underlying the varying experience of phenomena (Akerlind, 2002; Marton et al., 1993; Pang, 2003; Marton and Booth, 1997). Thus, newer studies have focused on combining phenomenography and variation theory. Variation theory argues that *"learners can only discern a particular aspect when they experience variation on that aspect"* (Pang, 2003). This constitutes the second-order of variation which is experienced by the learners but is captured by the researcher in a phenomenographic research project.

Figure 3.1: Object of study (adapted from Bowden, 2005, p.13).



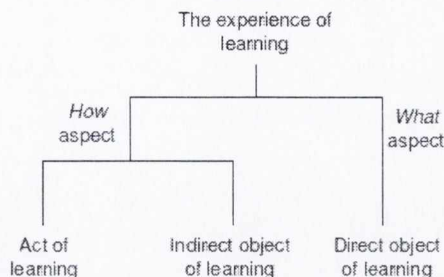
3.2 The Object of Learning

In a phenomenographic project the researcher needs to adopt a non-dualistic stance towards the experience under investigation. The research is not focused on some hypothetical mental structures of the learner or behaviourist actors (Marton and Booth, 1997). In phenomenography there is no division between the *outer* and *inner* worlds. The world and, therefore, its various phenomena, is not constructed by the individual nor imposed from the outside (Bowden, 2005). Rather, as Marton & Booth say: *"it is constituted as an internal relation between them. There is only one world, but it is the a world that we experience, a world in which we live, a world that is ours"* (Marton & Booth, 1997, p. 13).

This means that in a phenomenographic study the *object* of the study is not the phenomenon itself, but rather the relation between the study's population and the phenomenon. As illustrated in Figure 3.1 the phenomenon cannot be seen in isolation as the point of interest in the study is the way this phenomenon is understood and experienced by the learners. The aim of the researcher is to try and find out the object of learning by analysing the relationship between the learner and the phenomenon at hand. For example, in this study we consider how novice Computer Science students experience the construct of class within the object-oriented paradigm.

As can be seen in Figure 3.1 there is an unavoidable relationship between the researcher and the phenomenon that is investigated in any study, since the researcher is required to know and understand thoroughly the aspects of the phenomenon that they are attempting

Figure 3.2: The experience of learning (Marton & Booth, 1997, p. 85).



to analyse. This is necessary so that the researcher, and in most cases the interviewer, should be in a position to discuss and query the learner about the various facets of the phenomenon. However, there are methodological guidelines in the phenomenographic research approach that do not allow the researcher to impose his or her own interpretation of the phenomenon on the student cohort. Similarly, the danger of imposing the viewpoint of the researcher that arises from the relationship between the researcher and the population of the study also exists. This can be avoided through methodological sound procedures that are discussed in a later section of this chapter.

At the centre of any phenomenographic research project is the object of learning, that is how a phenomenon is experienced by a specific group of learners and the variation in the ways this phenomenon is understood. Thus, such studies adopt a positive attitude towards learning by focusing on the understanding and conceptions that people hold rather than on the misconceptions. In order to facilitate a structured way of analysing the understanding of a phenomenon (Marton and Booth, 1997) have developed a model for the analysis and description of learning as presented in Figure 3.2.

The experience of learning is something that can be seen through the *how* aspect and *what* aspect of the experience. The *what* aspect constitutes the direct object of learning which is the contents of the construct that is learnt and, furthermore, the phenomenon that is under investigation. The *how* aspect refers to the learner's approach in achieving his or her task. In other words *how* does the learner go about understanding and learning the construct of class for example. The *how* aspect is broken down into the *act of learning* and the *indirect object of learning.* Where the act of learning refers to *"the experience of the way in which the act of learning is carried out"* (Marton and Booth, 1997) and the

indirect object of learning refers to the goals that the learner is trying to achieve. The latter is also referred to as motives in (Berglund, 2005). This model for analysing the experience of learning is an essential tool for the researcher as it provides guidelines for identifying the critical aspects of the experience as well as providing a complete picture of the ways a phenomenon is understood. However as (Berglund, 2005) points out this distinction between the *what* and the *how* aspects is entirely analytical and is only used by the researcher to assist in the analysis.

3.3 Variation

Experiencing variation is an essential requirement for observing and understanding something in a certain way. Marton argues that *"[b]y experiencing variation, people discern certain aspects of their environment; we could perhaps say that they become "sensitised" to those aspects"* (Marton et al., 2004, p. 11). The point they are making is that unless you have a point of reference and a variation from it, discerning and understanding something in its entirety is not possible. One of the many examples they present in the paper is the following:

> *"[...] for instance, knowing what red is presupposes the existence of other colours, that is, a variation in colours. Even knowing what colour is, presupposes an experienced variation of colour. Imagine for a moment that there was no variation of colours, that everything around us had the same colour. It would be impossible for us to know what red, green, or yellow were, just as it would be impossible for us to discern colour as a feature."* (Marton et al., 2004, p. 14)

Thus, in order to be aware of something and understand it in a certain way, it is required that one experiences variation in it. The aspects that are considered important in understanding a phenomenon are called *critical features* (Marton et al., 2004). In the case above where the phenomenon is the colour red one should experience the variation of other colours, and possibly textures and objects of that colour, to gain a complete understanding of the phenomenon. Four different types of variation, or rather patterns of variation have

been identified. These signify the difference between the elements that stay invariant and those that do not in a learning situation. According to (Marton et al., 2004), these are:

1. *Contrast:* A person needs a point of reference to compare something with something else. As in the case above, in order to understand red you need to know about green, for example, so that you can discern the contrast.

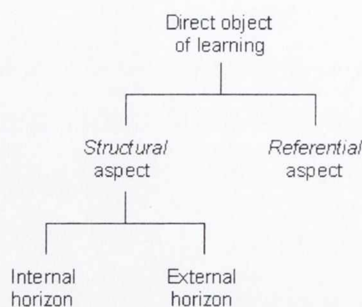
2. *Generalisation:* Variation in values of that aspect is necessary to discern the phenomenon. Thus, *cherry* red and *strawberry* red would constitute values in this pattern of variation.

3. *Separation:* In order to experience certain aspects of the phenomenon, and in order to be able to separate these aspects from other aspects, it must vary while other aspects remain invariant.

4. *Fusion:* In cases where the phenomenon has to be experienced in its entirety, meaning that there are several critical aspects to it that have to be considered at the same time, it is necessary that a situation should be present where these aspects are all experienced simultaneously. Therefore, there is a fusion of variation in the dimensions of variation of the specific critical aspects.

In order to identify the variation that facilitates certain types of understanding, the *direct* object of learning (the "what" aspect in Figure 3.2), is further analysed by (Marton and Booth, 1997) and illustrated in Figure 3.3. The ways of experiencing something, for example the construct of a class in object-oriented programming, differ in both their *meaning* (what a class is understood as) and *structure* (the relationship of its parts). The meaning is referred to as the *referential aspect* while the structure is referred to as the *structural aspect*. The latter denotes the parts of the understanding and their relationships (Marton and Booth, 1997). The referential and structural aspects are constructs that have been defined to assist the researcher in analysing the ways the object of learning is experienced and understood; their use is only for analytical purposes. The structural aspect of the learning experience is further broken into the *internal* and *external* horizons of the experience. The former denotes the focus of the attention and the relationships between the

Figure 3.3: Internal and external horizons of the direct object of learning (adapted from Marton & Booth, 1997, p. 88).



aspects and elements that are in focus. The latter denotes the elements and aspects that surround the phenomenon along with the elements that the phenomenon is related to and is a part of.

The meaning and use of the models presented in this section, along with the variation in the critical aspects of the experience, are further discussed in Chapters 5, 6, and 7 where the phenomena that are investigated in this study are decomposed into their referential and structural aspects.

3.4 Data Collection and Analysis

In phenomenographic projects the main source of data are the interviews that the researcher has conducted within the population of interest. The phenomenon that the study aims to analyse should be clear prior to the commencement of the interviews. However, before the main data gathering takes place, a pilot study conducted through interviews or by other means, such as questionnaires, is usually recommended (Akerlind et al., 2005). This allows the researcher to observe the interest of the phenomenon in question. The interviews usually start with a number of key questions that are used to set the theme of the interview and make sure that all the members of the sample population are discussing the same phenomenon. The questions are usually phrased in terms of the students' perceptions, understanding and experience (the set of questions that were used for this study can be found in Appendix B), but the interviewer can deviate from those when interesting angles of understanding are exposed through the discussion. These types of interviews are referred

to as *semi-structured* interviews (Kvale, 1996).

The selection of the theoretical sample for the interviews is also an important influence on the data collected. When selecting the sample population in a study the aim is to cover as broad a range of relevant characteristics as possible (Eckerdal, 2006). The relevant characteristics among the sample population can be their background, prior experience, gender, age and so on. These characteristics should be representative of the group under investigation, as well as of other similar groups in different educational settings. For example, in our study the sample population is comprised of novice Computer Science students. Thus the researcher is expected to make an effort to choose the theoretical sample, both in terms of how representative it is within this group of students and also among other novice Computer Science groups. This is fundamental to phenomenographic research since its main goal is to identify broad representations of the experience and not just to map characteristics of a group of people (Bowden, 2005).

After the interview data have been gathered they are transcribed verbatim and the highly iterative process of the analysis commences. In order to identify the different ways people are experiencing the phenomenon in question the text of the interview data are read repeatedly and the patterns that are particular to distinct understandings are noted. Initially the transcripts are seen as a whole and within the context of the particular subject, while in later stages the interview excerpts that are relevant to an understanding are decontextualised and compared to each other, thus forming groups. The excerpts are grouped and regrouped as the researcher further reads and compares the quotations, until the data remain in a stable condition and the outcome space is formulated. By taking the individual transcripts into account and through the comparison of the decontextualised excerpts, the researcher expresses the meaning of the group or patterns that have emerged by the formation of categories of description. It is important to note at this point that phenomenography assumes that there is only a limited number of qualitatively distinct ways a phenomenon can be understood (Eckerdal, 2006). These qualitatively distinct ways of experiencing a phenomenon are summarised in the categories of description. Marton and Booth describe the finding of a phenomenographic project in the following way:

"To be more precise, the outcome space is the complex of the categories of de-

scription comprising distinct groupings of aspects of the phenomenon and the relationships between them. The qualitatively different ways of experiencing a particular phenomenon, as a rule, form a hierarchy. The hierarchical structure can be defined in terms of increasing complexity, in which the different ways of experiencing the phenomenon in question can be defined as subsets of the component parts and the relationships within more inclusive or complex ways of seeing the phenomenon." (Marton & Booth, 1997, p. 125).

Therefore, categories of description form a hierarchy that extends from basic to more complex understandings. However, this is not passing judgement on better or worse ways of understanding (Akerlind et al., 2005). The hierarchy of the categories is formed based on both logical premises (i.e. which understanding is more basic compared to another) and, more often, on the inclusiveness of the understanding. Some categories that are more complex often presuppose the understanding that is encapsulated in a simpler category, and this imposes a hierarchical structure.

The issue of inclusiveness in understanding within the categories of description is a point of debate point among phenomenographers. There are two ways to determine the inclusiveness of meaning within the categories of description: a) logical argument and b) evidence from the interview transcripts (Akerlind, 2005). There are phenomenographers that believe that non-inclusive categories should not be allowed in the outcome space (Barnacle, 2005; Bowden, 2005; Green, 2005), while others pay little or no attention to it (Berglund, 2005; Eckerdal, 2006). We argue that pure logical inclusiveness could be seen as an imposition of the researcher's views on the results. In this study inclusiveness in the categories of description is solely based on the transcripts. Where evidence of the inclusiveness of some categories is weak this is clearly stated in order for the readers to be able draw their own conclusions.

3.5 Challenges of Phenomenography

As phenomenography is a qualitative research approach it is subject to the same criticisms levelled at other qualitative methodologies. In particular, issues relating to validity and reliability need to be addressed. In this section we discuss the potential problems that may

arise in a phenomenographic study and we focus, in particular, on the challenges faced in this study.

One limitation of phenomenography, and of most qualitative methodologies, is that it is impossible to say when there is no further information to be captured and, thus, when there is no more variation to capture. Marton and Booth (1997) discuss whether the results of a phenomenographic project apply to the individual, or to the group of individuals or the wider population. The objective of a phenomenographic study is to capture the variation and hence it is difficult to know when saturation has occurred as it heavily depends on the theoretical sample and how it represents the general, or specific, population studied. Thus, as Marton and Booth state: *"To the extent that the group represents the variation of individuals in a wider population (or is a theoretical sample of this population) the categories of description can also be said to apply to that wider population"* (Marton & Booth, 1997, p.124). Thus, the issue of saturation in variation cannot be easily verified in either phenomenographic or any other qualitative studies. As regards to the theoretical sampling of our study group, we carefully select the sample to be as representative of the divergence of the study's population as possible. The theoretical sampling procedures followed are further discussed in Chapter 4, while the generalisability of phenomenographic studies is detailed in Section 3.6.

The role of the researcher and the conditions under which the interviews are conducted are also very important in phenomenography. The interview conditions affect what people say and how they say it. Interviews can be partly influenced by the expectation of what the interview conditions are and what one is expected to describe in it, and partly by the relationship between the interviewer and the participant (Kvale, 1996). From the latter we conclude that research of any kind (qualitative, quantitative or mixed) is not independent of the people involved, both the researcher and the participants affect the process and hence the results (Eriksen, 2001). The theoretical and practical background of the researcher in the subject area influences the analysis no matter how open minded one tries to remain during the interviews and analysis.

To deal with this very important challenge a number of precautionary measures were taken during the interview process. Firstly the interview questions that were used were

carefully formulated with accordance to the phenomenographic philosophy presented by (Sandberg, 2000). The researcher involved in the study presented in this thesis was both the interviewer and a laboratory demonstrator to the class. Thus the students were familiar with the researcher from their laboratory classes and so they were open and relaxed during the interview sessions. The procedures that were followed during the preparation of the interviews and their execution along with the safeguards that were employed are discussed in more detail in Chapter 4.

In order to minimise the influence of the researcher's preconceptions on the outcomes of the study multiple data gathering methods were used. The interview transcripts were combined with observational data from exercises that the participants had to complete during the interview sessions. All of this information was then taken into account during the analysis process. The most important safeguard against bias during the analysis is for the researcher to be self-effacing in their attitude so that they can try to understand how the participants views the phenomenon under study in reality. In phenomenography the researcher needs to learn from the interviewee; to listen and accept the different understandings and not try to correct the participants conceptions. This cannot be easily shown to a third party in a phenomenographic study, thus a clear presentation of all the analytical steps carried out is essential as these then allow the reader to draw his own conclusions with regards to the validity and reliability of the results. A more comprehensive discussion on the trustworthiness of phenomenographic results is presented in the following section.

3.6 Trustworthiness in Phenomenographic Studies

As phenomenography is a qualitative research approach, the issue of trustworthiness in relation to the results that derive from such studies needs to be addressed. Generally, trustworthiness in qualitative studies is usually discussed in terms of the validity, reliability and generalisability of the results. The *scientific holy trinity* of validity, reliability and generalisability (as (Kvale, 1996) characterises it) derives from a more classical positivist approach to research than that associated with interview-based qualitative research such as phenomenography. Nevertheless, it must still be addressed in relation to this study. Thus, the three aspects of trustworthiness are re-framed within the epistemology of the

phenomenographic research framework.

Generalisability, from the perspective of qualitative research, refers to the extent to which the outcomes of a study can be expected to apply to environments other than the original one (Akerlind, 2002; Booth, 1992; Cuba, 1981). Kvale presents three forms of generalisability that are relevant to qualitative research in general, and not just to phenomenographic studies in particular. These are the following:

- *Naturalistic generalisation:* This develops for the person as a function of experience; it derives from tacit knowledge of how things are and leads to expectations, rather than formal predictions; it may become verbalised, thus passing from tacit knowledge to explicit propositional knowledge.

- *Statistical generalisation:* Is formal and explicit: it is based on subjects selected at random from a population. With the use of inferential statistics the confidence level of generalising from the selected sample to the population at large can be stated in terms of probability coefficients.

- *Analytical generalisation:* Involves a reasoned judgement about the extent to which the findings from one study can be used as a guide to what might occur in another situation. It is based on an analysis of the similarities and differences between the two situations. (Kvale, 1996, pp. 232-233).

Obviously not all of these forms of generalisation are applicable to phenomenography, clearly *statistical generalisation* is not relevant to phenomenography as the population of the study is not selected randomly, nor is it big enough to allow the derivation of probability coefficients. As the aim of phenomenographic research is to identify the range of ways in which the desired population experiences a phenomenon, the emphasis is on the *variation* of the experiences within the sample population. Thus, in order to capture the range of meaning, the sample population is selected on the basis of *divergence* rather than *representativeness*, since this approach is more likely to yield information on the range of ways of experience (Akerlind, 2002). However, the range of variation derived from the population is indicative of the range of variation expected in a similar population (Booth, 1992; Akerlind, 2005; Bruce et al., 2004; Berglund, 2004; Eckerdal, 2006). Naturalistic

generalisation in phenomenography is achieved through the development of the outcome space and further dissemination of the results. Analytical generalisation suggests that the results of a phenomenographic study should be generalisable to other groups of people drawn from similar environments (e.g. in our study: first year Computer Science students that learn object-oriented programming), however, this does not mean that the generalisability would be quantitative in the sense of any particular distribution that is present in the results. Rather, it means that the range of the experiences is quintessential to the ways of, and variations in, the experience of the phenomenon.

Reliability, in the context of a qualitative research project, is seen as *"the use of appropriate methodological procedures for ensuring quality and consistency in data"* (Akerlind, 2002) based on (Cuba, 1981; Kvale, 1996). The most common criterion for measuring the extent to which the results of a research project are reliable is *repeatability* or *replicability* (Sandberg, 1996). The central idea of replicability is that if two or more researchers arrive at findings similar to the original researcher when studying and analysing the same data, then the original researcher's results are reliable.

Kvale (1996) defines two forms of reliability that apply to interview-based research approaches, such as phenomenography and are based on different forms of replicability:

- *Coder reliability check* : Where two researchers independently code all or a sample of the interview transcripts and compare categorisations.
- *Dialogic reliability check:* Where agreement between researchers is reached through discussion and mutual critique of the data and of each researcher's interpreted hypotheses.

Therefore there are two issues that concern the reliability, and thus replicability, of results in a phenomenographic study. The first concerns the original discovery of the qualitative variation in understanding among the group of individuals. Whereby the question to be asked is, would other researchers reach the same categories of description as the original researcher? The second concerns the extent to which other researchers can recognise the understanding captured in the categories of description identified by the original researcher (Sandberg, 1996).

The first issue deals with the replicability of the original discovery of the categories. According to Marton *"The original finding of the categories of description is a form of discovery and discoveries do not have to be replicable"* (Marton, 1986). Also Le Compte and Preissle, when discussing reliability in social studies, say that no researcher in the social world can achieve total reliability. To be more specific they describe reliability as *"the extent to which studies can be replicated. It assumes that a researcher using the same methods can obtain the same results as those of a prior study. This poses an impossible task for any researcher studying naturalistic behaviour or unique phenomena"* (Le Compte and Preissle, 1984, p. 332).

As for the second issue, replicability is reasonable to expect in phenomenographic studies since it should be possible for independent researchers to reach a degree of agreement concerning the presence, or absence, of the given categories of description. To facilitate this reliability check, the researcher needs to make his/her interpretative steps clear to the readers by fully detailing each step and by presenting examples to illustrate them. This provides transparency throughout the whole process, from the initial data collection to the presentation of the final analysis. Thus, a clear presentation of the analysis phase with careful use of examples and supportive quotes, helps to justify the reliability of the study. As Sandberg argues reliability should be seen as the interpretative awareness:

> "Following the epistemology of intentionality underlying the phenomenographic approach implies first and foremost that the researcher must demonstrate how he/she has dealt with his/her intentional relation to the individual's conceptions being investigated. That is, in order to be as faithful as possible to the individuals' conceptions of reality, the researcher must demonstrate how he/she has controlled and checked his/her interpretations throughout the research process: from formulating the research questions, selecting individuals, analysing that data obtained from those individuals, analysing the obtained data and reporting the results." (Sandberg, 1996, p.137)

In Chapter 4 we further discuss how this reliability check has been applied in this study and further measures that were taken to establish the reliability of the findings presented.

The fundamental concept of validity in the context of qualitative research is considered

as the extent to which a project has adequately reflected the phenomenon under investigation and the degree to which the objectives of the research have been realised in the actual study (Akerlind, 2002). In other words, validity in this context is assessed in terms of whether a study successfully answers its research questions through the presentation of suitably justified findings. The issue of validity may be discussed in terms of communicative and pragmatic validity checks (Akerlind, 2002; Kvale, 1996):

- *Communicative validity check:* Involves the researcher's ability to defend the interpretations of the data. In this check both the research methods and final interpretations are under consideration. Acceptance of these through their dissemination among the educational community constitutes the validity check.
- *Pragmatic validity check:* Deals with the usefulness of the research outcomes and the extent to which they are meaningful for their intended audience.

Both these validity checks are relevant to phenomenographic projects and are applied rigorously.

In this section the theoretical aspects of the issue of trustworthiness in phenomenographic research have been discussed. In Chapter 4 we further discuss how the various checks of generalisability, reliability and validity have been applied in this study.

3.7 Why Phenomenography

The nature of the research questions that this study set out to explore meant that a post-positivistic or naturalistic research approach was needed. Possible methodologies to be adopted included Grounded theory, constructivist theory and general ethnographic methods. Grounded theory is an inductive qualitative research method. In this approach, the researcher begins by collecting data in the field and allows the theory or phenomena emerge or emanate from the data (Glaser, 1992). This differs significantly from an approach where the researcher starts with an hypothesis and then sets out to test it. Although grounded theory appears an attractive alternative to phenomenography, the focused nature of our study to the specific themes meant that grounded theory was not suitable for this project.

The constructivist approach from the field of cognitive psychology and socio-cultural studies may also be used within educational and learning studies such as the one presented in this thesis. The basic premise of constructivist theory is that an individual learner must actively "build" knowledge and skills and that information exists within these built constructs rather than in the external environment (Bruner, 1990). Many studies within this framework have produced interesting and noteworthy findings, some of these (Aharoni, 2000; Hazzan, 2002; Holmboe, 2000) were discussed in Section 2.1.4.1. However, constructivism describes *how* a learner comes to learn something while phenomenography focuses on the students' relationship to the object of their learning, the latter being more appropriate to the aims of this study.

The research questions that this project set out to investigate are such that phenomenography is the most suitable research approach to realise them. Unlike other research approaches phenomenography enables individuals to voice their perceptions of a given phenomenon, but undertakes the analysis in a way that cuts across individuals and across contexts (Green, 2005).

In this chapter, the theoretical, methodological and practical aspects of phenomenography were presented along with the challenges that a phenomeographic researcher is faced with. The challenges and limitation of phenomenography are taken into consideration and appropriate safeguards put in place during the design and implementation phases of this study in order that the findings are reliable and useful to educators in Computer Science.

3.8 Summary

In this chapter we have detailed phenomenography in both its theoretical and methodological aspects. Although the phenomenographic research approach is relatively new, it has been very well documented and the evolution of the approach can be traced through the literature, as described above. The methods and guidelines that this research approach suggests, and how these were applied in this study, are further discussed in Chapter 4.

Chapter 4

The study

This chapter details the empirical study that forms the core of this thesis. As presented in Chapter 2, there have been some investigations of how students learn to program in various educational settings (Booth, 1992; Bruce et al., 2004; Eckerdal, 2006). However, these studies focus on either individual aspects of learning to program (Bruce et al., 2004), or on a limited number of programming constructs (Eckerdal, 2006). In this study our aim is to present an integrated picture of the complex activity of learning to program from the perspective of novice Computer Science students. When learning to program, students are presented with a number of key principles that are not necessarily all linked to a specific programming language or programming paradigm. Thus, in order to gain a complete picture of what it means for the students to learn how to program in the object-oriented paradigm, the study is broken down into three distinct areas. The first of these is the general question of the nature of programming as an activity; the second concerns the more specific and unique constructs of object-oriented programming; the third relates to the general programming constructs that are not necessarily related to any programming paradigm. Another focus of this study is on the early experiences that first year university students have when they are introduced to object-oriented languages; in particular on the act of learning and thinking in the object-oriented way.

The themes that were chosen for inclusion in this study were carefully selected to reflect the main concepts and ideas of object-oriented programming and programming as a whole. The choice was based not only on the informal results of the pilot study, but also on the

author's informed choice of themes of interest. In making these choices, the primary goal of this study, which is to present a comprehensive insight into students' experience of learning to program as a whole, was used as the main frame of reference. The themes of this study are grouped under three headings: *theoretical*, *object-oriented* and *general programming* components. The headings provide a clear separation between the themes that are more general to the experience of learning to program and the themes that relate to the technical aspects of programming.

The students selected for this study were drawn from a Computer Science degree program, and consequently they were highly motivated to learn how to program. The first year programming module of this course is highly intensive and stimulating, providing students with many opportunities for self-study. Furthermore, this study is not merely based on interview data, but also on a wide variety of material gathered through observation and from the exercises that the students were asked to complete during the interviews.

The selection of the themes, courses and sample population, along with the methodological issues of this longitudinal phenomenographic study, are discussed in detail in the remainder of this chapter.

4.1 Themes of the study

The themes of the study are grouped under the three headings: theoretical components, object-oriented components and general programming components. Theoretical components or themes are those which are more generally related to the experience of learning to program such as "the perception of program correctness". As this study aims to investigate students' experiences when learning to program within the *object-oriented paradigm*, the object-oriented components include the most fundamental and unique constructs of this paradigm such as objects, classes, methods, attributes and constructors. To construct a complete representation of students' experience, the general programming components were also included in this study. The general programming components considered are not specific to object-oriented programming but rather they are essential to all programming paradigms. The general programming components include students' understanding of an algorithm, array, iteration and selection. This distinction between themes is a nat-

ural boundary which provides a deeper understanding of the results obtained. Table 4.1 presents the main themes that were included in the interviews.

Table 4.1: Themes of the study.

Themes	*Theoretical Components*	*Object-Oriented Components*	*General Programming Components*
Programming	√		
Learning to program	√		
Understanding of correctness	√		
Object		√	
Class		√	
Method		√	
Attribute		√	
Constructor		√	
Iteration			√
Array			√
Algorithm			√
Decision making			√

The selection of the themes for this study was based on a number of factors. Since the main objective of most research projects in the field of Computer Science education is to improve teaching and learning, the themes of this study had to be relevant to both the educator and the learner. To determine the themes, and their relevance to both the educator and the learner, an informal pilot study was conducted. Part of the pilot study was a survey conducted among the lecturers, research assistants and tutors involved with programming courses. The particular courses were the first and second year programming courses in the Computer Science and Engineering degree programs provided by Trinity College Dublin in 2004. This established a set of constructs that students find most challenging to understand and use. An interview with the lecturer of the first year programming course selected for this study took place after the survey had been completed. In this interview the set of chosen themes was discussed to ensure their relevance and researchability. As this study attempts to construct a complete picture of the experiences of students when learning to program, some of the theoretical themes (for example, students' perception of correctness) were added at a later stage based on observation and informed personal

interest. Additionally, the course syllabus was thoroughly studied to ensure that all of the themes investigated were part of the curriculum.

Not all themes and constructs on the course syllabus were included in this study. The omitted themes included: object orientation, object-oriented design, variables, data types, strings, Boolean operators, vectors, inheritance, applets, graphical user interfaces, abstraction and encapsulation. There were two main reasons for not including these. Firstly some of the theoretical themes such as object orientation, abstraction and encapsulation were not explicitly taught during the course rather they were introduced and applied throughout the course implicitly. Although these are interesting themes, there was concern that they would be too vague and difficult for students to conceptualise, which would lead in turn to incoherent data. The second reason for excluding some of the themes mentioned above was the fact that they were explicitly linked to the programming language. Applets and graphical user interface are extensions of Java and are part of its API (Application Programming Interface). Since our study hopes to explore students understanding of object-oriented programming in general, rather than Java programming in particular, these themes were not considered appropriate for this study. As for the more technical constructs such as data types, variables, strings, these were not included due to the fact that they were too simplistic and easy to understand based on our pilot survey and on the subsequent interview with the lecturer concerned.

4.2 Selecting the Course

The study's focus is on novice programmers' understanding of the concepts involved in object-oriented programming and programming in general, thus the desired student group for inclusion in this investigation were first year programming students. The course from which students were selected to participate in the interviews was *Introduction to Object Oriented Programming* taught at Trinity College Dublin. The official description of the programming course is given below (Dukes, 2005):

> *"[...]This course takes a practical approach to teaching the fundamental concepts of computer programming with a strong emphasis on tutorial and laboratory work and is an important vehicle for developing students analytical and*

problem-solving skills. This course aims to give students an understanding of how computers may be employed to solve real-world problems. Specifically, this course introduces students to the object-oriented approach to program design and teaches them how to write programs in an object-oriented language" (in this case, Java).

The course is part of the four year Computer Science degree program offered by the college. This degree program attracts highly motivated individuals from various backgrounds who intend to pursue a career in the field of computing and computer science research. The reasons this course was considered an attractive option for this study were that firstly it is a pure Computer Science degree program and secondly, as previously mentioned, these students have a high motivation to learn more about Computer Science and computer programming. However, the students' prior knowledge of programming exhibits much variation since no programming experience is required in order to be admitted to study for this degree. Many students had very limited or no prior experience with programming, while others had extensive familiarity with computers and programming. Hence, this course was considered representative of first year programming classes in Computer Science and so likely to yield interesting and significant results. The programming language that is used to introduce the students to object-oriented programming is Java (SunMicrosystems, 2006), a well known and widely used programming language.

4.2.1 Structure of the Course

Trinity College Dublin traditionally has two terms of nine weeks and one of six weeks, with an additional "study week" at the end of each term for independent reading. Although some students may have prior programming experience, the course is an introduction to object-oriented programming suitable for complete novices.

The course is taught by a lecturer with the help of a teaching assistant and two demonstrators who provide support for students. The lecturer on the course is a very experienced Professor in the Computer Science department who has been teaching first year programming for over eighteen years. Both the demonstrators and the teaching assistant for this course are postgraduate computer science research students who are familiar with Java at

the development level.

The weekly schedule for this course consists of the following:

- Two hours of lectures, where the students are introduced to the programming concepts at a theoretical level.

- Two hours of tutorials, where the students are required to complete a set of programming exercises using pen and paper. These are then graded and returned to the students with appropriate feedback.

- One hour laboratory class, the students are given small programming problems that they have to implement and compile; these are also graded.

The students are introduced to new programming concepts during lectures, starting from basic constructs and gradually expanding to include more complex and abstract ones. The material is presented in the form of PowerPoint presentations that are then made available to the students through the course web page. Additionally, students are given a book written by the lecturer. This book includes all of the material that is covered in the lectures, together with extensive examples and additional exercises (Cahill, 2001). The emphasis of the course is highly practical: in the three hours of the course devoted to tutorial and laboratory work, the students are presented with programming problems and are required to produce solutions to them. At the end of each session the demonstrators provide possible solutions to the programming task. During the sessions the demonstrators provide feedback and assistance to students. In the laboratory sessions the programming environment used is *Eclipse*. Eclipse is an open source programming platform widely used in educational and industrial settings (Eclipse-Foundation, 2005).

4.2.2 Contents of the Course

The main objective of the course is to teach students how to program in the object-oriented way using Java. However, the course is not limited to this. In order to learn how to program, it is not sufficient to know the syntax of a language; logical thinking and problem decomposition are also essential and form part of the course content. However, due to the fact that these are considered implicit skills, they are not explicitly taught

but rather it is hoped that students develop them during the programming course. The contents of the course include an introduction to objects and classes, variables, methods, basic types, expressions, selection, iteration, arrays, applets and graphical user interfaces. At the end of the course the students should be able to (Dukes, 2005):

- Design algorithms using sequence, selection and iteration.
- Design object-based programs using class-based decomposition.
- Write, compile, test and debug computer programs in an Interactive Development Environment (Eclipse).
- Understand how programs are written in a high-level programming language and how this is then translated into a form that is executed and understood by a computer processor.
- Recognise the software engineering concerns that give rise to the use of classes and other abstraction mechanisms.

To ease the transition into the logical thinking style that is essential for programming, the lecturer spends the first four weeks of the course teaching the development and construction of algorithms through the use of commands in plain English. After the students have grasped the notion of creating algorithms by breaking the solution to a problem down into very simple steps, the course moves straight into object-based programming using the Java syntax.

As well as the weekly coursework assignments, the students are given four relatively large programming problems throughout the year. The weekly coursework and programming assignments together make up 30% of each student's overall grade, while the remaining 70% is awarded for the written examination at the end of the academic year. For students to pass this course it is required that they pass both the coursework and the examination i.e. they receive a grade of 40% or higher in each component. Thus, it is important for the students to achieve good grades on both and, as a consequence, most of them are very diligent throughout the year as well as during the examination period.

4.3 Selecting the Students

As was pointed out in Chapter 3, the selection of any study's population is essential to its success, thus diversity among the selected population was desired to ensure that a rich set of results was obtained. The students taking the programming course in the academic year 2004-2005 were asked to complete a background questionnaire and this was used to assist the selection process. This questionnaire included sections on the students' educational background, the programming languages that they might have learnt prior to commencing the course and their motivation for choosing this particular Computer Science degree program (see Appendix A). The questionnaire was also used to query the students on whether they were willing to participate in the study for four paid hours of interviews.

The total number of the first year computer science students in October 2004 was 40, including those who were repeating the course from the previous year and those who had transferred into the course from another degree program. Initially 19 students expressed an interest in participating in the study, from which 16 were selected following the theoretical sampling method described in Chapter 3. The diversity among the sample population, in terms of gender, prior programming experience, repeating students etc. is presented in more detail in Appendix A. Although participation in the study was voluntary, to motivate the students to participate to the end of the study it was decided to offer them a modest remuneration.

The students were interviewed four times during the academic year, once late in the first term, once in the middle of the second term, and twice in the final term. Since all the students had timetabled classes at the same time, the hours that they were available for interviewing were limited. Thus, about two weeks were required to complete a full set of interviews. All students participated in all interview sessions. However some rescheduling was required as some students missed their scheduled interview appointments.

To ensure human-subject consent and address any confidentiality and ethical issues involved, all the students who participated were asked to sign a consent form (see Appendix A). The consent form assured the students that all the contents of the interviews were confidential, and that they would appear anonymous in any publications based on the study. In this thesis, the participating students have been given fictional names. Since the

female participants were few in number, they were also assigned male names to render the population gender neutral.

4.4 Data Collection

The data used in the analysis were collected in the academic year of 2004 - 2005. The complete set of data comprises the following:

- The background questionnaires that were completed by the students in the first week of term.
- Four sets of interviews with all 16 students who were selected to participate in the study.
- Laboratory and tutorial exercises that the students brought with them for some interview sessions.
- Four think-aloud exercises that the students completed during the interview sessions.

The information gathered from the background questionnaire was used for selecting the students that participated in the study. The interviews were the main source of data for the phenomenographic analysis of the constructs presented in Table 4.1. The tutorial and laboratory exercises that the students brought with them were used during interviews as a starting point for further discussion on their experience of the programming constructs. Finally, the students were asked to complete some think-aloud exercises during the interview sessions. These were used to assess and investigate their understanding of the object-oriented and general constructs of programming. The think-aloud exercises were transcribed and used as part of the interview data. The main purpose of these exercises was to ensure that the understanding they were expressing when answering to the interview sessions was what they really perceived the construct to be. Thus, during the analysis the think-aloud exercises are used in the same way as the interview transcripts. The strategies that the students employed when solving the programming problems and their success or failure in completing them is also mentioned during the analysis when it becomes relevant.

4.4.1 Interviews

The 64 interviews conducted with the students were semi-structured and followed the guidelines for phenomenographic interviews described in Chapter 3. The distribution of the investigated themes over the four interview sessions was determined by the order in which students were introduced to new concepts during the course. Some of the theoretical, object-oriented and general programming constructs were discussed at more than one interview session. A theme was discussed at more than one interview session when the first interview session did not yield enough variation or when the students did not appear confident when expressing their understandings. Thus the students' experience of a construct could be observed both when it was first introduced to them and at a later stage when they had more experience with it. Generally, students appeared to be more confident in describing programming constructs after having had the opportunity to practise them. As the study investigated a large number of constructs (see Table 4.1), among a relatively large (for a longitudinal, qualitative research study) sample population, it was not feasible to discuss all of the constructs on multiple occasions. Table 4.2 presents the distribution of the themes discussed over the four interview sessions.

Table 4.2: Distribution of the themes across the interviews

Themes	**Interview 1**	**Interview 2**	**Interview 3**	**Interview 4**
Programming	√			√
Learning to program	√		√	
Understanding of correctness				√
Object	√	√		√
Class	√	√		√
Method	√	√		√
Attribute	√	√		√
Constructor		√		
Iteration			√	
Array				√
Algorithm	√			
Decision making			√	

Each interview had a basic structure with a number of questions designed to capture information on the themes of interest. However, the interview process was relaxed in nature and therefore allowed for some deviations from the basic structure depending on

the points of interest that arose in the discussion. The main purpose in having an interview outline was to ensure that the themes of interest would be addressed during the session. The interview sheets used for the four sessions can be found in Appendix B. All the students who participated in the study were Irish thus the interviews were held in English. Although the researcher is not a native English speaker she has a high level of proficiency in the language. In order to ensure that the interview questions were phrased in a manner that would help to reveal the specific themes of interest, they were further discussed with the researcher's supervisor and colleagues prior to the commencement of each interview.

From the first session, the students were informed of the purpose of this research project, which was to capture *their* understanding of the programming concepts. However, it was observed that when students were asked about their understanding of the object-oriented and general programming constructs many tended to repeat definitions from the book and the lecture notes. Hence it was evident, in some cases, that their answers were not really *their own* understanding. To address this problem the researcher regularly reminded the students that she was not looking for correct answers but rather for their own understanding. Additionally, after talking about a concept in an interview session the students were sometimes given small programming tasks where they had to use the underlying concept and explain its use and purpose out loud. These small programming tasks were insightful as there were times when the students realised that what they had earlier expressed verbally was not how they used the construct in reality. Four of these programming tasks were completed during the four interview sessions and these can be found in Appendix B. These programming tasks were selected from their course textbook (Cahill, 2001) and some were adapted from the tasks they completed during their laboratory or tutorial sessions, specifically problems 1 and 2. The other two programming problems (3 and 4) were selected from the textbook after discussing their appropriateness for the constructs investigated with the lecturer and my research advisor. The main criterion used for selecting the programming problems was their potential to open up further discussion on the construct and whether their completion would signify that the student knew how to use the construct investigated. Even though these programming tasks enhanced the interview data, it was not feasible to include one for every technical and object-oriented construct. In some in-

terview sessions students were asked to describe their understanding based on assignments and tutorial exercises that they have previously completed. Many of the concepts in the study, such as object and class, are very abstract and students found it easier to describe their understanding through the use of examples.

The four interview sessions were distributed throughout the academic year. The timing of the interview sessions was strongly dependent on the syllabus and times at which new constructs were introduced. Thus, the first interview session was conducted late in the first term at which stage the students had had extensive experience of constructing and working with algorithms and a first experience of object-oriented constructs. The second interview session was completed in the middle of the second term after the students had handed in their first assignment and completed an informal examination. The second interview session focused solely on the object-oriented constructs, as at that point in time the course was very focused on their use and capabilities. At the third interview session their notions of iterations and decision making were discussed as by that stage of the course students had extensive experience with them. At the third interview the theme of learning to program was revisited in order to see how the students' understanding had progressed during the academic year. At the beginning of the final term the students had learnt, about and were using, two-dimensional arrays; thus in the last interview session their understanding of arrays was discussed along with their notion of what constitutes a correct program. This final interview session also included a discussion on object-oriented constructs along with the student's overall evaluation of the programming course (see the Table B.4 and B.5).

The majority of the interviews went according to plan. The interviews took the form of friendly conversations in a relaxed and comfortable atmosphere. The fact that the students were familiar with the researcher as their laboratory demonstrator set the students at their ease. Some of the students had strong, local, accents and one of them spoke very quickly. In these cases the students were asked to repeat some of their answers, just to make sure that the researcher had understood exactly what they were explaining. However, none of these difficulties were insurmountable, and all of the 64 interviews were valid and included in the analysis.

4.4.2 Observation

Being part of the teaching staff for the course was of direct benefit to the researcher in this study. Firstly, the daily contact with the students helped the researcher to build up a constructive relationship with them and provided her with the advantage of knowing how each student usually approached a programming task. This meant that questions asked during the interviews could be tailored to a student's individual style. For example, in a tutorial session students were given a scenario for creating a banking management system and were asked to define the classes that would be required for their design. This exercise generated a variety of solutions that were based on different assumptions. These were clarified during the interview sessions, illuminating the students' different understandings of classes. Another advantage of participating in the course was that the researcher was aware of exactly what was taught each week, thus the interview content could be appropriately modified when necessary.

Although notes were kept of discussions and events during the class, these were not used as evidence for supporting or contradicting the outcome space that was formed based on the interview data. However, when observational data strongly supported or contradicted the findings this is mentioned in the analysis, allowing the reader to draw his or her own conclusions accordingly. Observational data were mainly used for prompting questions during the interview sessions. Observational data allowed the researcher to relate to the students' understandings more easily, while the sources of influence on a student's understanding were often apparent to the researcher from her knowledge of the course.

4.5 Transcriptions

The interview transcriptions began as soon as the interviews were completed. The interviews ware captured electronically using a Dictaphone device and later transcribed verbatim into Microsoft Word format. Both the audio files, and the transcriptions can be made available to researchers on request. The number of interviews was very large and thus it took over four months to transcribe them all. The researcher personally transcribed the interviews, allowing her to become familiar with the data. By the end of the transcription

process, the first sorting of the data by theme was also complete. There were a number of interviews where the transcription was challenging due to the interviewees accent and speed of speech. These were handled more carefully, with native English speaking colleagues used to verify the transcriptions.

The task of transcribing the interviews was not taken lightly. Since the transcribed interviews formed the basis for establishing the outcome space it was essential that the process of transcription was carried out to the highest standards. The interviews were transcribed verbatim, which means that in some cases there were repetitions and incoherent sentences. However, in order to assure the validity of the transcripts these passages were not altered to extract the subject's intended meaning. According to Kvale:

> "To *trans*-scribe means to *trans*form, to change from one form to another. [...] The problems with interview transcripts are due less to the technicalities of transcription than to the inherent differences between an oral and a written mode of discourse" (Kvale, 1996, pp.166-167)

This problem, as described by Kvale, was a fundamental concern during the audio transcription. In the phenomenographic tradition, the interviews are always transcribed verbatim and used as the basis for the outcome space. However, the transcriptions were not used as static written words in this study. During the transcriptions the researcher tried to capture the ongoing conversation as realistically as possible. In many cases, apart from the actual verbatim transcriptions, notes were included to describe the setting and the mood the subject was in during the interview. Some students tended to use their hands while talking, especially when they had difficulties expressing something as well as they would have liked to. This material could not be transcribed, although notes were kept during the interviews when this was the case. A student's hesitation when expressing something was marked with a group of dots ("...") while "thinking sounds" such as "Hum", "Hem", "ahh" where also included when encountered. Other emotional aspects of the conversation, such as laughter and giggling, were also annotated when appropriate. Thus substantial effort was devoted to capturing as much of the essence of the emotions as possible in the transcriptions, while always keeping in mind the issue of quality.

4.6 ATLAS.ti

Due to the large volume of material contained in the text of the transcriptions, it was decided that a software tool should be used in the analysis of the interview transcripts. The software tool used was ATLAS.ti, *" [a] powerful workbench for the qualitative analysis of large bodies of textual, graphical, audio, and video data"* (Muhr and Freise, 2003). The software was not used to automate the process of analysis but to assist in identifying the themes of interest more effectively. ATLAS.ti allows for the representation of the interview data as *primary documents* that can be later grouped according to session, student name/identifier, or theme of interest. It also allows for the creation of selected quotations from the interviews that can then be grouped and referenced at a later stage. Additional features of the software include utilities such as *codes*, *memos* and *network groups*. Although the features of codes and memos were relatively straightforward, network groups were an altogether new and very useful utility for the researcher. Network groups allow for the creation of complex conceptual structures that connect similar elements together using diagrams. These were extensively used during the analysis of this study since the categories of description were represented as codes. The supporting quotations and memos assisted in the visualisation of hierarchies of understanding. As phenomenographic analysis is highly iterative, network diagrams helped in grouping and regrouping the categories as they were reaching stability. These diagrammatic representations illustrate the phases of the process of analysis. However, these were not included in the analysis presented in this thesis, since they would distract the reader. This material, along with the notes from the analysis, are available to researchers on request.

4.7 Analysis

Each interview transcript was between five and twelve pages long, giving the researcher a very rich (and large) set of data. The analysis had to be conducted in a very structured manner due to the large volume of data and the number of phenomena under investigation. Initially each interview transcript was read as a whole and the preliminary analysis involved sorting the interview sessions according to theme. Thus, the analysis was conducted on a

theme by theme basis, with one theme under examination at a time. After the primary documents were collected for each theme, the relevant content-oriented passages were identified. These quotations were first read in relation to the whole interview and were then decontextualised and separated from the rest of the document. Subsequently, preliminary codes were assigned to these quotations and the preliminary categories of description were identified. These preliminary categories of description were then refined and examined for inclusiveness whilst being tested against the rest of the textual data. As many themes were closely related, reflecting on the different quotations that supported them allowed for further refinement of their meaning. This was achieved by considering the differences and similarities between each quotation.

The analysis started in November 2005 and continued until December 2006. The analysis of each construct took between two and three months during which time the outcome space for each one was re-read in relation to the interview transcripts and it was documented in this thesis.

4.8 Validity, Reliability and Generalisability in Practice

The issues surrounding the trustworthiness of phenomenographic studies were discussed in Section 3.5. Issues relating to how validity, reliability and generalisability were addressed in this study are discussed below.

Generalisability

The generalisability check that applies to phenomenographic studies is that of *analytical generalisation* (Kvale, 1996). This is also affected by the selected population of the study. To address the issue of generalisability in this study the background of the participating population was thoroughly examined and the sample chosen to be as heterogeneous as possible. Appendix A provides information on the background of the students in terms of their previous programming experience, gender and education. This collective information on the students' characteristics together with a detailed description of the environment, the course and the course content (presented in Section 4.2) would allow the interested reader to determine whether the collective characteristics of the study's results are applicable to

any particular student population.

The length and depth of the analysis did not allow for the repetition of the empirical study at other institutions with a similar sample population, therefore no hard evidence can be presented to support the generalisability of the specific results. Nevertheless, all possible methodological guidelines were rigorously applied for the duration of this study to ensure that the results would be generalisable. According to (Marton and Booth, 1997) if we consider a less similar population then the ways of experiencing the phenomenon in question that are captured in the categories of description should still be relevant, but possibly with a less complete range of the experienced variations. Since some of the phenomena that were investigated in this study have been investigated previously in different educational settings, it was possible to compare and contrast our findings with these. Illustrating the similarities and differences between these studies and the one documented here reinforces its generalisability.

Reliability

As discussed in Section 3.5, replicability of the original discovery of the categories of description does not constitute a meaningful reliability check in social studies such as this one (Le Compte and Preissle, 1984). It must be noted here that most phenomenographic studies towards a Ph.D., such as (Booth, 1992; Berglund, 2005; Cope, 2000; Eckerdal, 2006), have not employed this coder reliability check. The dialogic reliability check "where agreement between researchers is reached through discussion and mutual critique of the data" (Kvale, 1996) has been employed extensively in this study. All the procedures that were involved in this study, from the formulation of the research questions, to selecting the course and later collecting and analysing the data, have been discussed with the researcher's supervisor and other members of the TCD Department of Computer Science and School of Education. The outcome space for each phenomenon of this study was discussed mainly with researcher's supervisor, but additionally part of the results were further cross-analysed by Dr. Owen Conlan, a member of the university's academic staff who has expertise in learning studies.

However as Sandberg (Sandberg, 1996) argues, the most appropriate form of reliability

check in phenomenographic studies is the clear presentation of the interpretative steps with the help of examples and excerpts to illustrate them. Therefore, this study is presented following a logical structure that allows the reader to clearly identify the processes and the factors that influenced the interpretation of the data and, hence, the outcomes. In Chapters 5, 6, and 7 the categories of description are individually analysed, supported by the appropriate quotations and methodical tables that illustrate the findings in all their dimensions.

The findings of this study are presented in a clear and systematic manner in the subsequent chapters, while great emphasis has been placed on discussing the findings in the light of other research in the field. Despite that fact that many of these other studies followed different research paradigms, nevertheless we discuss and, when appropriate and feasible, compare them with our findings. It was found that these results corroborate and hint at the generalisability of the categories of description found in this study, however this cannot be used as a measure of reliability as it does not show the repeatability of the results. However it provides strong indications that the understandings captured by the categories of description are recognised, and present, in other educational settings.

Finally, our analysis is not solely based on interview data. In order to safeguard against any possible bias or misunderstanding on the part of the researcher, written exercises were used and observational data was gathered. These were taken into account throughout the analysis of this study and are mentioned when appropriate in order to enhance the presentation and discussion of the categories of description.

Validity

Validity in the context of qualitative research is considered as the extent to which a project has adequately reflected the phenomenon under investigation and the degree to which the objectives of the research have been realised in the actual study (Booth, 1992; Kvale, 1996; Akerlind, 2002). In other words, validity in this context is relative to whether a study successfully answers the research questions by presenting sufficient justifications in support of the findings. In Chapters 5, 6 and 7 the findings of each of the themes presented in Table 4.1 are discussed and supported by excerpts from the interviews. Additionally,

comprehensive discussion and comparison with the literature articulates the relevance of the results within the field. The results of this research have been discussed with other colleagues both in the TCD Computer Science department and in other institutions. This work has been peer reviewed, presented and debated at conferences and workshops within the Education and Computer Science community.

4.9 Summary

In this chapter, the structure of the empirical study that underpins this thesis has been presented. The selection of the themes analysed and the details of the educational setting considered have been thoroughly discussed. The methodological aspects of phenomenography and the way these were practised in this study have been detailed so that the reader is provided with all the necessary information on which the trustworthiness of the results is based. In the following three chapters, the findings and outcomes of the project are discussed in detail.

Chapter 5

The Theoretical Components of Programming

In this chapter the theoretical components of the study are analysed and discussed. The students' conception of the *nature of programming*, *learning to program* and their understanding of *program correctness* are not strictly confined to the object-oriented paradigm, thus the students were asked to discuss these with respect to their current and previous experience. The students' general understanding of these themes is considered to be an important aspect of their overall experience since we believe it affects the general approach they employ towards learning the technical components of the language. In this chapter the themes are initially discussed individually and then the relationships between them are examined to provide an insight into the overall learning experience.

The first theme considered is the students' understanding of programming. This was discussed with the students in two interview sessions: once very early in the course and again at the end of the course. By doing this we sought to capture the development of student awareness and observe how their understanding develops as the course progresses, thus providing a more complete insight into each conception. The second theme, students' understanding of learning to program, was discussed in the third interview towards the end of the course; while their understanding of program correctness was examined in the last interview session when the students had received feedback on almost all their assignments and laboratory work.

In the following we analyse these three theoretical themes, presenting the categories

of description as they were identified among the study's population. The conceptions are then discussed in terms of their focus and structure. The outcome space that has been developed for each theme is then related to the literature and compared to the findings of other studies, thus providing a complete presentation of our results within the field of Computer Science education.

5.1 Students' Understanding of Programming

The main focus of the study is to explore and document the experiences of novice students when learning to program and think within the *object-oriented* paradigm. To fully capture this, the interviews start with the foundations of programming in the broader sense and consider how programming is understood in its essence as an activity. This theme was visited in two different interview sessions: once in the very first session and again in the third one. The interview questions that were used to investigate the theme were: *"What is programming for you?"*; *"How do you understand it?"*; and *"How would you describe it as an activity?"*. Four different categories that form the outcome space have been identified and are summarised in Table 5.1. The categories are clearly distinguished in terms of their focus, their meaning and the relationships between them.

Table 5.1: The categories of description of *programming*.

Category Label	**Category Description**
1. Coding	Programming is experienced as putting together a set of instructions; as the act of coding in a programming language.
2. Manipulation of hardware	Programming is experienced as manipulating hardware in terms of low-level binary operations in switches and circuits.
3. Interaction	Programming is seen as a form of interaction or even as communication between the person who is programming and the computer.
4. Problem-solving	Programming is experienced as a problem-solving activity that involves breaking down the problem and devising an algorithm to solve it.

Category 1: Programming as coding

In this category programming is experienced as putting together a set of instructions so that the computer will work in a certain way. Thus, programming is experienced as the act of coding toward a goal. The statements that belong to this category stress the point of instructions and code as a means to achieve a desired behaviour. As Eamonn states in his first interview:

> **Interviewer:** So what do you think programming is?
>
> **$\mathbf{Eamonn_1}$:** Programming would be defined as putting code to the machine to make the machine work in certain applications.

Another comes from Liam:

> **$\mathbf{Liam_1}$:** Give a computer a set of instructions. To do whatever you want.
>
> **Interviewer:** So you see the act of programming as writing a set of instructions then?
>
> **$\mathbf{Liam_1}$:** Yes I would say so ... programming is coding towards a target which could be any problem really.

Another from Mark:

> **$\mathbf{Mark_1}$:** It is a series of commands really ... later these are translated by the computer to solve a problem.

and a last one form Neil:

> **Interviewer:** How would you describe programming?
>
> **$\mathbf{Neil_1}$:** The writing of a program.
>
> **Interviewer:** Could you be a bit more specific please?
>
> **$\mathbf{Neil_1}$:** It is code mostly... but I suppose since we've done a lot of things in English it could extend to that as well. So writing an algorithm really is a sequence of commands to do things.

All these show a clear conception of programming as an activity involving writing code, or as it is expressed in the quotations "programming is a series of commands". Although

Eamonn, Liam and Mark clearly state that programming is about giving instructions to the computer to do something, Neil is a bit more specific in his expression saying that it is about writing an algorithm. Neil has a more developed conception of programming, however the primary focus is still on code and instructions. A more mature view of programming experienced as coding comes from Tim and Sean.

> **Tim3:** ...Emm programming is doing things through a computer... through a language writing an algorithm to do a certain something ... to make a computer do something.

> **Sean3:** Hm ... ah (programming is) writing out in a programming language... methods and instructions that the computer can perform.

The last two passages bring out the use of the programming language as a means of describing the instruction that one gives to a computer while programming. These come from the third interview session. The primary focus of the students in this category is on code. Thus, code is viewed as the means one uses when programming to achieve something. However, in all the passages the goal of programming has been unspecified.

In this category programming is experienced as a one-way relationship with the computer that involves writing code for the computer in order to achieve a desired behaviour. There are, however, several aspects to this understanding. When compared with other excerpts, Eamonn, Liam and Mark's views focus on the code in an abstract and vague way. While Neil's view of "programming as coding" focuses on the development of algorithms through coding. Tim and Sean are more specific when they refer to coding involving the use of a programming language to describe the instructions. Thus, there are three distinct aspects in this category; coding as a set of instructions, coding as the development of algorithms, and coding as the use of the programming language. These are discussed in Section 5.1.1.

Category 2: Programming as manipulation of hardware

Here programming is experienced as the activity of manipulating the computer hardware in order to achieve a desired behaviour that would be of benefit to the user. The statement

in this category is concentrated on the internal operations that take place during the processing of a program. The whole activity of programming is experienced as an intangible process that is centred on the low level processes that follow the execution of a program. As Patrick explains in the following interview excerpt:

> **Interviewer:** Can you tell me what is programming for you?
>
> **Patrick$_1$:** I guess it is a higher level of abstraction of how a computer works. It is a way of hmm ... I need to phrase this carefully to get it right. I guess when you are programming it is a way of utilising the hardware of the PCs.
>
> **Interviewer:** So you say that programs in general and thus the action of programming results in utilising the hardware of the computer?
>
> **Patrick$_1$:** Yeah, but in an abstract way.
>
> **Interviewer:** What do you mean when you say abstract?
>
> **Patrick$_1$:** Essentially it means that at the end of the day the computer can work only with ones and zeros and all these integers and stuff... all these terms we create for programming are basically abstract, they descend to zeros and ones... I mean that programming is a very abstract procedure because whatever you do at the end of the day it is translated to zeros and ones so what we are doing is that we are creating our own language to make the computer understand more complex things than zeros and ones.

This understanding is also voiced by Anthony:

> **Anthony$_3$:** Programming is a set of instructions so that the computer knows how to do hmm use the hardware. With some sort of instructions formatted ... like you get your compiler... ehm that would translate English or the language you are using into machine code because the computer can only understand machine code.

Similar to Category 1, students here perceive programming as a one-way relationship with the computer that aims to create something (unspecified again) useful for the user utilising the computer hardware. However the distinction is that the focus in this category is on low-level operational processes compared to the high-level understanding presented in Category 1 where the conception was targeted on the code itself.

Category 3: Programming as interaction

This category encapsulates an understanding of the nature of programming as an interaction between the programmer and the computer itself. It incorporates the views from Categories 1 and 2 but instead of a one-way relationship with the computer, programming here is experienced as a two-way relationship in terms of input and output. Thus the act of programming is experienced as a means to communicate the programmers' wishes to the computer that in return produces an output. Patrick expresses this in his first interview as follows:

> **Patrick$_3$:** Well I guess it's writing algorithms the computers understand so when somebody basically is writing a program I guess like in a way... ehm... I suppose it involves user input and the program uses the input to produce an output, I guess this would be the best way to describe it.
>
> **Interviewer:** So you say that programming is creating a program that takes an input and produces an output.
>
> **Patrick$_3$:** Pretty much yes ... it is a sort of communication between you and the computer in a way.

Alan who first expressed his understanding of the nature of programming as learning a set of instructions, elaborates in his response to uncover yet another facet of his understanding:

> **Interviewer:** So you see the act of programming as learning a set of commands then, right?
>
> **Alan$_1$:** Yes, just how to write, how to tell a computer in the simplest terms as you can, to do something you want it do. So, it is a general interaction between you and the computer since you give it something and then you get something back as a response.

Cormac, on the other hand, views the act of programming in two levels:

> **Cormac$_1$:** At the very basic (level) it is turning on math switches inside the computer, in a higher level it is just a way of communicating how we react to different things, and how we communicate and... how computers communicate and it is the connection between the two.

Interviewer: Between the person and the computer?

Cormac$_1$: Yeah.

These clearly indicate the presence of a distinct category that describes the nature of programming as a two-way interaction between a person and the computer. All the above statements focus on the communication that results from programming a computer. When comparing the statements from Patrick and Alan with the ones from Cormac we can see that the latter conception is based in Category 1 while the former two include the understanding presented in Category 2. Patrick and Alan view programming as communication however they seem to concentrate on the low-level, more grounded elements like input and output. Cormac on the other hand has a more abstract understanding of this interaction. Thus there are two different approaches in this category; one suggests interaction through user input while the second is more conceptual, presenting programming as a communication of human behaviour.

Category 4: Programming as a problem-solving activity

This fourth conception of the nature of programming focuses on the problem-solving aspect of programming, while it also incorporates the understandings described in the previous categories. The primary focus here is on the problem that the student has to deal with. The conceptions described in Categories 1 and 2 are present in this category, but they have moved to the background and thus are not in focus. Although the main focus of the passages that belong to this category is on problem-solving and the various techniques that one uses to do that, the computer is still central in their conception. As Sean argues in the following, programming is all about the construction of algorithms:

Sean $_1$: I would say programming is writing algorithms for a computer to solve a problem I suppose. I guess it is a way to make the computer to solve problems using various algorithms.

Colin focuses more on the techniques involved in problem-solving:

Interviewer: So what is programming then for you?

Colin 3: Hmm it's a process of doing complex... ehm writing an algorithm... an algorithm to... do you mean programming in general?

Interviewer: Yeah.

Colin 3: The process of performing complex tasks ... solving the problems by breaking them down to smaller steps... hmm... That's about it really.

Cormac is more specific in his description of programming; breaking down the problem into particular programming constructs like objects and their attributes.

Cormac 3: Hmm breaking down stuff to what they are and what their attributes are, and putting all that in the form that a computer can understand. Hmm basically it is just making algorithms... I don't know...

Interviewer: You said that is about breaking down the problem...

Cormac 3: Yeah so you basically have this problem and you break it down to what is it made up... like you break it down to objects, classes... it is just breaking down into parts and then you solve each part. And then you put it all together and then you have a program.

Cormac is a bit hesitant in his answer, but it is clear that his conception includes the low level description of programming as coding also found in Category 1. The integrating approach that is described by Cormac - of decomposing a problem and then composing parts of the solution to ultimately solve the problem - is also expressed by Brian in his third interview.

Brian 3: [...] like you are given like a problem and just having like sort of a set of things that you can achieve and just trying to build layers of these ... achievement to achieve whatever [...] basically it is scaling down into not only just one big thing but into smaller things.

Karl on the other hand seems to unveil another aspect in understanding programming where the motivation is for the programmer to gain a reputation within a community:

Interviewer: So what is programming for you?

Karl1: Ahm mmm on what level do you mean?

Interviewer: When you are programming what are you doing?

Karl_1: Well I suppose solving a problem. Well by using specific tools.

Interviewer: So you see it as a problem-solving activity?

Karl_1: Yes in one level.

Interviewer: What other levels are there?

Karl_1: Hmm I suppose there is more to programming for some people they do it for fun or reputation or hacking.

Students that experience programming as the activity of solving a problem, focus primarily on the problem at hand and the various strategies that one can employ to solve this. Many students when describing programming in this category said that it is about writing *algorithms* for the computer. For that reason their understanding of algorithms was taken up as a theme for the study and it is discussed in Chapter 7 along with other general programming constructs.

Although problem-solving is central in the students' responses in this category, some students like Sean focus more on the construction of algorithms, while others such as Cormac have a more technical or even low-level view that deals with the specific object-oriented programming constructs of object and class. Brian, on the other hand, focuses more on problem decomposition and the incremental construction of a solution. Thus programming, when experienced as a problem-solving activity, incorporates three different approaches: problem-solving as algorithm creation, problem-solving as object and class identification, and finally problem-solving as problem decomposition.

5.1.1 Structure and Meaning of Students' Understanding of Programming

The structure of the students' experience of programming as an activity is presented in Table 5.2 following the guidelines in (Marton and Booth, 1997). As is observed in the referential aspect of the categories, the meaning gradually evolves from an incomplete experience that deals only with the tangible aspects of programming (Category 1), to a broader view that takes into account the user and the programming problem (Categories 3 and 4). The internal horizon (column 3) shows how the different elements that form

the foundation of the experience are related to one another and how they connect to the category and the theme itself. In this column the shifts from one category to another are minor. In comparison, we see that parts of the foundations of Category 1 (coding) appear in the rest of the categories.

In Category 1, "coding", the internal horizon is narrow and disconnected; while in 2 the notion of computer hardware is introduced, giving a purpose to the activity of programming. In 3 the internal horizon extends to include the user in the foundations of the experience; providing an actor for programming. Thus the ultimate reason for the activity of programming is to facilitate the user in utilising the computer through software. Finally, in the last category the foundations have become discernable from the manifested aspects of the nature of programming, and the conception is directed beyond the current experience to that of real-world problems.

The focus in Category 1, *"programming is experienced as putting together a set of instructions as a description of coding in a programming language"*, is on the instructions that the programming language provides and the actual act of using those in combination to achieve something (unspecified). The variation that conditions this understanding comes from the different instructions that exist in the programming language and the ways these are combined to produce different programs. Students that share this understanding are primarily concerned with learning the syntax of the language and the program layout, frequently using sample programs as guides.

Category 2, *"programming is experienced as manipulation of the hardware in terms of low-level binary operations in switches and circuits"*, deals mostly with the internal operations that result from the act of programming. This category is similar to the previous one in that here programming is experienced as an intangible process centred on the low-level operations that follow from the execution of a program, while the purpose of programming is the utilisation of the hardware. The variation in this category is brought out by the different binary operations and usage of the computer hardware. As in the previous category, the students' experience of programming focuses more on learning the instructions of the language.

In Category 3 programming is seen as a form of interaction, or even as communication,

Table 5.2: Understanding of programming; the focus of the conceptions.

	Referential aspect	**Internal horizon (foundation)**	**Focus**
1. Coding	Coding in a programming language.	Code, programming instructions, programming language.	Coding as a set of instructions; coding as the use of the programming language.
2. Manipulation of the hardware	Manipulation of the lower level operations of the computer.	Code, compilers, binary operations, computer hardware.	Manipulation of the hardware through programming.
3. Interaction	Interacting with the computer.	Programs, user, computer, interaction.	Interaction through user input; communicating human behaviour.
4. Problem -solving	Solving problems through the use of the programming language and the computer.	Programming language, problem specifications, real-world problems.	Problem-solving as creating algorithms; problem-solving as identifying objects and classes; problem-solving as decomposing the problem.

between the person who is programming and the computer. The concepts presented in the previous categories are still present but they have moved into the background. Here programming is viewed as the means for a two-party communication between the programmer and the computer. This involves input from the programmer/user and a response or output from the program. Variation here is brought out by the different situations where programs require input to produce meaningful outputs. Thus students in this category program by keeping in mind this two-way interaction and how this can be achieved.

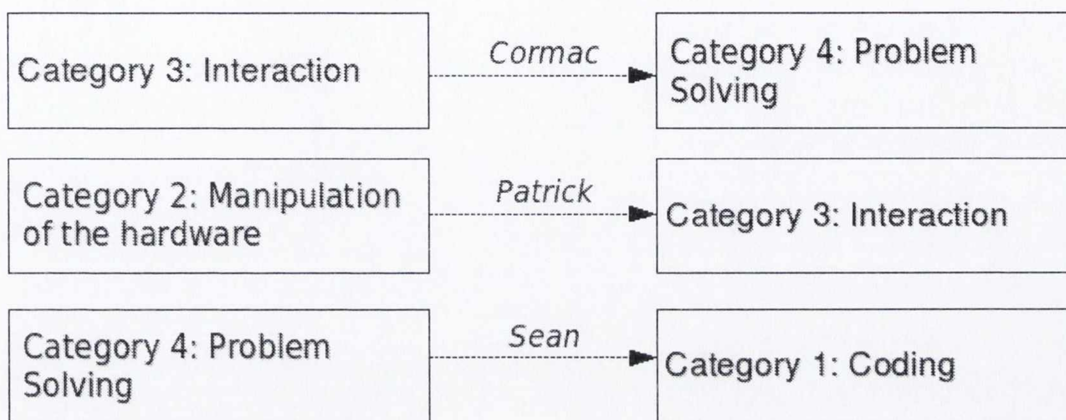
Finally, in Category 4 programming is experienced as a problem-solving activity that involves breaking down the problem and devising an algorithm to solve it. The focus here has moved to the actual problem to be solved through programming. The conception has developed to include both the user of the program and the problem itself. The variations within the conception itself are that programming is experienced as a problem-solving activity that targets the creation of algorithms, the identification of objects and classes through the problem definition, and finally the decomposition of the problem. Students that belong to this category are not concerned with the syntactical details of the language but rather concentrate on the problem and the techniques that they can use to solve them in an efficient manner. This does not mean that the syntax of the language is not important rather that it has now moved to the background of their experience.

5.1.2 Changes in the Students Conceptions During the Course

The structure of meaning that was presented in the previous sections comprises the full breadth of the outcome space that reveals the perceptual boundaries of the categories. In many phenomenographic studies the act of learning is seen as the development of the students' perception of the phenomenon to a richer and more complete understanding (Booth, 1992). In terms of our study this means the transition from one conception to another that is qualitatively "better" with respect to the logical hierarchy that has been identified. That is not to say that a category is better than any other in terms of the potential learning outcome. A successful student needs to acquire and adapt to the conceptions encapsulated in each category at different stages of their studies. A potential problem would arise if the students' experience does not evolve further than Categories 1

and 2. In such cases additional assistance would be required to encourage the students to experience the different ways of viewing the act of programming.

Figure 5.1: Shifts in students' conceptions of the nature of programming.



In this study there were 3 different cases of a shift in the understanding of the nature of programming as shown in Figure 5.1. All except for one case (Sean) have shifted towards a category that would be considered qualitatively enhanced in the outcome space. Since the categories are inclusive this only highlights that the students were focused on the different aspects of their understanding in the two interview sessions. The shifts are considered in isolation in this section.

Cormac in his first interview says that programming is communication; in his exact words: *"At the very basic (level) it is turning on math switches inside the computer, in a higher level it is just a way of communicating how we react to different things, and how we communicate and... how computers communicate and it is the connection between the two."* The understanding presented here is very abstract, and it is focused around the programmer/user and the computer while the act of programming is perceived as the medium that is used to achieve this interaction. In the third interview session Cormac responds to the same question as follows:

> **Cormac $_3$:** Hmm breaking down stuff to what they are and what their attributes are, and putting all that in the form that a computer can understand. Hmm basically it is just making algorithms[...] so you basically have this problem and you break it down to what is it made up of... like you break it down to objects classes... it is

just breaking down into parts and then you solve each part. And then you put it all together and then you have a program.

Cormac's response describes a technique that one could follow to simplify the problem and ultimately solve it. The notion of the programmer and the computer are presented here as well, however the emphasis is on the programming problem and the problem-solving techniques. Thus the elements of his previous conception are retained, but now a more specific understanding is observed that deals with the tangible aspects of programming. It is common to observe shifts in phenomenographic studies where two interview sessions have taken place (Booth, 1992; Berglund, 2005); a comprehensive analysis of the types of such shifts can be found in (Pong, 1999).

In Patrick's first interview he voiced his understanding of the act of programming as manipulating the computer hardware (see Section 5.1, above). There he talked about the way a program is translated into binary in order to be executed by the computer. The focus of his understanding is on the computer and how hardware can be utilised through the programming language. Additional emphasis is given to the internal operations of the computer when it executes the programs. In his third interview his focus has shifted to the interaction that is achieved through user input and the program's output.

> **Patrick3:** Well I guess it's writing algorithms the computers understand. So when somebody basically is writing a program I guess like in a way... ehm... I suppose it involves user input and the program uses the input to produce an output, I guess this would be the best way to describe it.
>
> **Interviewer:** So you say that programming is creating a program that takes input and produces an output.
>
> **Patrick3:** Pretty much yes ... it is a sort of communication between you and the computer in a way.

The shift in focus towards writing programs is evident. The answer stresses the importance of input and output as the means of interaction with the computer. The significant difference between this and the previous excerpt is that here the conception of the act of programming is extended towards the user while the reasons for writing programs are made more specific.

Sean's shift in understanding is in marked contrast to that of Cormac and Patrick in that his shift in understanding was from a higher to a lower qualitative category of description in the hierarchy of the outcome space. Sean initially experienced programming as a problem-solving activity. Specifically in his first interview he said, very simply, that programming is *"writing algorithms for a computer to solve a problem I suppose. I guess it is a way to make the computer solve problems using various algorithms."* The emphasis in his short answer is on writing algorithms while the goal of the act of programming is to solve problems. The main aspects of his understanding include the programmer, the computer and the problem. This conception clearly illustrates a "problem-solving" understanding of the nature of programming. In his third interview Sean says *"hm ... ah (programming is) writing out in a programming language ... methods and instructions that the computer can perform."* Here the focus has changed and it has moved towards the more tangible elements of programming, like the programming language, and the specific concepts of programming, for example methods. Thus, initially Sean experienced programming in a conceptual way, looking at the big picture; while later, when he had to deal with the specific elements of programming, his understanding changed and shifted to the concrete features of the process. This should not be viewed negatively, since for a student to be successful it is necessary that he discovers the meaning encapsulated in each category in order to develop a complete understanding of the theme.

5.1.3 Discussion on the Nature of Programming

In 1992, Booth conducted a phenomenographic study where the nature of programming was investigated as one of the themes (Booth, 1992). The study's population was drawn from a first year Computer Engineering degree program. The programming paradigm that was presented to the students was functional programming and the language that was taught in the course was standard ML. The conceptions that were identified among the study's population were the following (Booth, 1992):

- **Programming as *a computer oriented* activity**, in which programming is conceived of as an activity that focuses on the computer.
- **Programming as a *problem oriented* activity**, in which the main focus is on the

problem that programming activity is intended to solve, rather than the computer itself.

- **Programming as a *product oriented* activity**, in which the main focus is on the program as a product, in the sense that the programming is an activity for producing programs for potential users, and/or accessed for maintenance by other programmers.

These conceptions encapsulate a rather global and more generic experience of the nature of programming, particularly when compared to the ones found in our study which could be characterised as more pragmatic and focused. Nevertheless the focus of Booth's categories is reflected in the categories of description found among this study's population and vice versa. In the foundations of the first and second categories (coding and manipulation of the hardware, respectively) programming is experienced as a one-way relationship with the computer. Both coding and manipulating the computer hardware through programming encapsulate the conception found in Booth's first category of description, i.e programming as a computer oriented activity.

The second conception that was found among the Computer Engineering students summarises an experience where programming is seen as a problem oriented activity. The same understanding has been identified in the fourth category of description in this study where programming is seen as a problem-solving activity. In both studies, the categories show that the experience has moved from the computer and is now focused on the problem and the techniques that can be employed to solve it.

The conception of programming as a product oriented activity was not explicitly voiced by the student cohort of our study. However in the third category, where programming is seen as an interaction with the user, the notion of an end result is present. In this category the sense of programming as an activity where the user/programmer interacts with the computer to get a result reflects some of the properties of the third category found in Booth's study, but the complete conception as articulated by Booth is never present within the studied population.

Given that the two studies have been conducted independently separated in time by over fifteen years, and with different populations and courses, the fact that such strong parallels of understanding can be seen between the studies strengthens their validity and

generalisability. The difference in the course and background of the studied population can explain the difference in the detail of the conceptions encountered.

5.2 Learning to Program

Learning to program is another of the theoretical themes taken up by our study. While learning to program is a phenomenon that has been investigated in previous studies (Booth, 1992; Bruce et al., 2004), all of these have been in educational disciplines other than Computer Science. The relationships between their results and those of this study are further compared and discussed in Section 5.2.2. The main questions that were used to investigate the students' conceptions of this theme were: *"What do you think learning to program is all about?"*; and *"What do you think it takes to learn how to program?"*. Six distinct, and in some cases inclusive, ways of experiencing learning to program were identified. These are presented in Table 5.3.

Table 5.3: Categories of description for *learning to program.*

Category Label	**Category Description**
1. Learning the syntax of the language	Learning to program is experienced as learning the syntax of the language.
2. Learning and understanding the programming constructs	Apart from learning the syntax of the language, the focus here also includes learning and understanding the constructs involved in programming in general.
3. Learning to write programs	As above, but also utilising all these to write programs.
4. Learning a way of thinking	In addition to the above, learning to program is also experienced as learning how to think logically in general.
5. Learning to "Problem Solve"	As above, and also utilising this way of thinking to solve programming problems.
6. Acquiring a new skill	The whole process is experienced as learning a new skill that affects the way one thinks in real-life.

The first three categories focus on the programming language and its constructs while

the next two bring out the unique way of thinking that is required when programming. The last category describes how learning to program involves the acquisition of a new skill that can be utilised in everyday life. A more elaborate presentation of the categories follows, along with supporting excerpts from the interviews.

Category 1: Learning the syntax of the language

Here learning to program is experienced as learning the syntax of the language. The students that belong to this category seem to direct their efforts towards learning the keywords and the details of the syntax, in some cases even by heart. Knowing the details of the programming language provides the student with the ability to write a piece of code that compiles without any syntactical errors and this is what they experience learning to program as.

As Alan emphasises in his response, the important factor is knowing how to fix small, syntactical errors that may occur.

> **Alan$_3$:** Hmm well it is necessary to like computers because you have to like them otherwise you don't enjoy it... and for me learning Java and learning programming is just all about sitting down writing code and just practicing and at the end it starts to make sense. It is about learning... and learning everything about the grammar how to phrase things and where you put the marks like the semicolons and stuff... and it is more about learning the structure first how it works and then you actually learn the specifics. That's how I did it.
>
> **Interviewer:** Can you explain to me a bit more what you mean about the specifics?
>
> **Alan$_3$:** Oh yeah, the syntax, you know, learning to fix the small errors that you would... you know, you have the basic knowledge but then you still need to refine it... you know, you may put the wrong brackets at some point and you may not know the `equals` method to compare the two strings, you know, stuff like that.

Alan refers to the basic knowledge one has about programming, but the main focus of his experience is on learning the grammar details of the language. Although he briefly mentions the structure of the program the emphasis in his response is on the syntactical operations.

Another student, Liam, points out that the only thing you need to do to in order to learn how to program is to get a book from which one can learn the syntax and features of the language.

> **Liam$_3$:** It's about getting a couple of books, really... Well it is the language that you are interested in anyway and the syntax of the language is what you learn from the book. Then you have to mess around with the syntax and the features of the language and having fun you learn more, really [...]. So what you can do depends really on the language and this is, really, what you learn from a book, the syntax of the language, that's all you need anyway.

Here we get the perceptions of a complete novice (Alan) and a relatively experienced programmer (Liam) that both focus their experience of learning to program on learning the syntax of the language. However, this does not necessarily mean that Alan and Liam have the same level of programming understanding. Liam, as a more experienced programmer, has reached a level where learning to program is not an issue for him. This knowledge is implicit for him and therefore learning to program in a new language is just a matter of learning the syntax of that new language. Alan on the other hand exhibits a naive understanding as he appears not to be aware of other important facets of programming. Thus, he explains what learning to program is by taking into account only the pragmatic elements of his experience. This actively illustrates that the learning outcome of each category is not a fixed attribute that can either guarantee success or failure, but a mapping of students' understanding. As was mentioned in the previous section, the desired learning outcome is for the students to acquire the knowledge encapsulated in each category of understanding during the course.

Category 2: Learning and understanding the programming constructs

Students in this category are focused on the process of learning and understanding the constructs involved in programming in general and therefore this category follows directly from the first one. The understanding expressed in Category 1 is presumed, however the syntactical details of the constructs have receded into the background of the experience. The dominant feature of this category is understanding the essence of the programming

constructs. Neil expresses this conception strongly:

> **Neil$_3$:** [...] you need to know the syntax of the language that you are programming in and the concepts like iterations and conditions. But all these you kind of need to know how to use them, like, and when to use them... The syntax of these things is also important but you need to understand the concepts first not in a single language, like, because they kind of exist for most of the programming languages.

Neil is very clearly distinguishing between the syntactical details of the language and the programming concepts such as iterations, conditions, etc. Stephan also talks about the techniques you need to learn when programming. When asked what it takes for someone to learn how to program, he responded:

> **Stephan$_3$:** It takes a good deal of work actually.
>
> **Interviewer:** Work on what?
>
> **Stephan$_3$:** On studying concepts... programming concepts and the languages.
>
> **Interviewer:** When you say concepts, what exactly do you mean?
>
> **Stephan$_3$:** Well, basically, techniques, the basic understanding of how to make a computer do something and then the language is something that you learn in order to translate that to something that the machine can understand, but you have to understand first how the concepts work so you can use them when you need to the right way. You know...

Both students very clearly and plainly express an understanding of learning to program that deals with learning and understanding the programming constructs.

Category 3: Learning to write programs

In this category the focus moves from the syntax and the programming constructs to the actual writing of programs. Thus learning to program is experienced more as utilising the syntax and the features of the language to write programs that solve the problem at hand. Although the focus is clearly on the programs; the nature of the programs is not specified. Anthony exhibits this conception in the following answer:

> **Interviewer:** What do you think it takes to learn how to program?
>
> **Anthony₃:** It involves setting out your ideas after thinking how to, like, put these ideas into the proper syntax and you need to know the commands in whatever language you are working in and then writing out the program. That's about it really.

Anthony's conceptions include the understanding presented in Categories 1 and 2, as he talks about the *proper syntax* and *the commands*, however the emphasis is on learning to program as the activity of writing programs.

Sean points out the importance of learning from other programming examples:

> **Sean₃:** Ehm... first of all the basics of how the language works and what is the grammar that it uses... and then just various different examples, just to get used to how it is being used really. That is the only way to learn it really properly by writing and writing... small programs at the start and then bigger as you get to know more. You need lots of experience.

Sean's point, of learning from programming examples, is not stressed in Anthony's response, however they both focus on the fact that learning to program is about learning to write programs. Mark summarises this conception by saying:

> **Mark₃:** You need to be able to know the language you want to program in, but that is not enough, you need to be able to write the programs right, that's programming.

The construction of programs is central in this category. The views presented in Category 1 and 2 are also present but they seem to have moved to the background. All the passages included in this section refer to the syntax of the language as something one needs to learn when learning to program, but according to the students this is not all that is needed when learning how to program. Therefore, there is a clear distinction between this category and the ones presented before, since this one is more pragmatic and unveils a richer learning experience, even if it is not complete.

Category 4: Learning a way of thinking

In this category the perception has progressed from language and programs to learning a new "way of thinking". Many students have expressed that when programming one needs

to be in a different frame of mind. Hence, in order to be able to program one needs to learn to think in this new way. This frame of mind has been vaguely described as logical thinking. Patrick explains this in the following way:

> **Interviewer:** What do you believe is required, or what does it take to learn how to program?
>
> **Patrick$_3$:** You definitely need the sense of logic, that is the basic thing because everything else then is based in your ability to logically think of a solution, even with some of the other programs that we are doing you need the logic to be able to see it. Basically, you need decent mathematics and a basic sense of logic and the practice, of course. Pay attention and then it clicks, I guess.

Patrick here links the process of learning how to program with learning mathematics. This may have been influenced by the fact that the students in the labs and tutorials were given programming examples and problems based on mathematics. One thing is clear though; this category reveals that programming requires the learner to be in a different frame of mind where logic is central. Patrick mentions something else as well in his response, this "click" moment that many students have talked about in our informal conversations during the course. This moment is described as something that one cannot control and when it happens, programming starts to make sense and the learner is able to program as if all the knowledge has fallen into the right place.

Category 5: Learning to "Problem Solve"

This category derives from the previous one in that the concept of logical thinking is now expressed as a condition for problem-solving. This more complete conception involves learning to problem solve as part of the experience of learning to program. Being able to see the solution to a problem is a critical feature of this category. The importance of problem-solving is highlighted by the teaching staff from the very beginning of the course and the students have been taught to perform case analysis when attempting to solve a problem. This involves finding all the different cases that may arise in the solution to a problem and the conditions that affect them. Usually, students are encouraged to write down the conditions in English and use this as a guide when writing the code for their

solution. This has possibly affected the number of students who belong to this category. For example, Eamonn describes this in the following quotation:

> **Eamonn$_3$:** [...] Ehm logic, how to think logically and... breaking the whole thing down. Like in the tutorial the demonstrator told me that you need to break the whole thing down and then you work out what you need to have and how you go about it (case analysis), and then the whole thing made a lot more sense. It is just ... you only understand by being shown things... the book is good so then you only have to learn it (the syntax) off.
>
> **Interviewer:** So after the demonstrator showed you that then you could do it?
>
> **Eamonn$_3$:** Yeah I learn better when someone shows me first. Like not that I can do it perfect now but after he showed me it's just clicked, and I felt I could do that. Like I am not sure that I understand everything but I can pretty much use them if you know what I mean. It sort of makes more sense.
>
> **Interviewer:** So you say that one thing you need to know when learning to program is how to do case analysis. Right?
>
> **Eamonn$_3$:** Yeah... Let's just say... I am actually coming from the... actual point that Java code doesn't matter that much, like I haven't learnt it. Like I can do it with a book along side I am not confused anymore. But the important thing is to think in this way so you see the solution... Do you know what I mean?

Unlike Eamonn, Brian does not explain how he learnt to solve programming problems; rather he focuses on the characteristics that a good solution should have.

> **Brian$_3$:** I think it's more trying not to look at it as one, it is not just one problem, but more as to how you are going to deal with the problem. There is a lot of ways probably... an infinite number of ways that you can do any one problem. It's just trying to do it the smartest and quickest way really not the quickest way, the best, the more efficient way.
>
> **Interviewer:** So learning to program is about being able to see the more efficient solution to a problem?
>
> **Brian$_3$:** Yeah.

Declan points out the difference between problem-solving in programming and problem-solving in other situations:

Declan$_3$: Hmm logic and just kind of... hem... it has to be clear in your mind what you have to do, if you know what I mean. Like, there is always going to be certain cases when you have to solve a problem and you just have to know what rules you have to follow and what kind of things you have to do to solve it. [...] [B]ecause you can solve problems without having to think so much about it, where in computer programming solving a problem is more complicated because the instructions are so basic you have to go down and solve the problem into its very basic elements, like, I think a lot of the problem and its solution. Because when you are thinking of a problem solution in your mind you take a lot of things for granted but it is not this way in programming, because you have to think more basic.

The focus in this category is on problem-solving, and the students have described how one can learn this technique. This category reveals a more developed conception of learning to program that has extended beyond the previous categories to include the problem as well as the ways one can approach its solution. The logical and systematic way of thinking described in Category 4 is now used to enable problem-solving.

Category 6: Acquiring a new skill

The final category conceptualises learning to program as learning a new skill that affects the way one thinks in real-life. This conception assumes understanding of the language and the programming constructs while it also draws on elements from Categories 4 and 5. However the way that thinking logically enables problem-solving is now viewed from a different perspective, that extends beyond the course to the real-world. Students experience learning to program as the acquisition of a new skill that can be useful in other areas of their studies, such as learning other programming languages, or even in different areas, like mathematics. Colin voices this understanding in the following:

Colin$_3$: Just practice really. The more familiar you get with how it works then you build a more solid base so I mean I would say that I have a pretty fair idea of how Java works.

Interviewer: Have you learnt anything else to reach that point in programming ability that you have now, except for the syntax that you've already mentioned?

Colin$_3$: Well I didn't have to carry out anything very difficult so far but, yeah, you learn how to break down things to more manageable pieces which would be applicable to any programming language and any mathematical operations or anything really.

In a different interview session when students were asked what they thought was the most important thing they learnt during the course Ken and Cormac said the following:

Ken$_4$: [...] sort of you work out how to analyse problems and you realise that maybe there are ways of doing something, that you haven't thought of before and you can apply that to other things like in 1BA3[1].

He then continues by giving a mathematical example where the logical thinking he learnt in the programming class has helped him solve a problem he had in his mathematics class. Ken has explained how his new skills helped him in other courses while Cormac moves even further:

Cormac$_4$: Yeah I did (learn) definitely logic and problem-solving etc. and it helps in many other ways like cleaning the flat (laughs)

Interviewer: How was that?

Cormac$_4$: (giggles) I mean the first time we tried to clean it everyone was trying to do the same thing, but the next time we said, you do this and then you do the other, and everything happened very fast and very structured.

Cormac argues that learning to program has affected the way he behaves and accomplishes tasks in real-life situations, like when he is cleaning his flat. This conception removes the experience of learning to program from the strict boundaries of the course and university, and makes it part of the learner's everyday world.

5.2.1 Structure and Meaning of Students' Experience of Learning to Program

The qualitatively different ways that students experience learning to program when they are first introduced to it are presented in Table 5.3. The six categories that have been

[1] 1BA3 is the code name for the assembly course in the first year Computer Science degree program.

identified within the study's population are clearly distinct and inclusive, in the sense that each one assumes the conception(s) formulated in the preceding categories. Thus, the earlier categories express a relatively basic understanding and as we progress through the categories the experience becomes richer and the view broadens and matures. The focus of the understanding and the structure of the students' experience is presented in Table 5.4.

The referential aspect of the first three categories, *learning the syntax of the language*, *learning and understanding the programming constructs* and *learning to write programs*, reveals an understanding that is more strongly oriented towards the technicalities of the process of learning to program. However the internal horizon and the focus of each category is different.

The experience in the first category is solely limited to the student learning the syntactical details of the underlying programming language. Here the variation is brought about by the different syntax the programming expressions have and the practices one uses to learn them. In Category 2, the conception has evolved to a more abstract understanding of programming constructs that is not confined to the taught programming language (Java) as these exist in other languages as well. The variation here is introduced by the different programming constructs like conditions, iterations, object and so on. The experience broadens even further in the third category where the focus is on the way all of the previous categories enable the creation of programs. This category is more pragmatic when compared to the previous one and the variation here comes from the different problems and the different concepts that are required to be used to arrive at the end goal which is the program itself. The existence of these particular conceptions is not surprising as they fit, more or less, within the structure and development of the course. In particular, the study of programming focuses on each of the foundation elements (column 2, Table 5.4) present within the first three categories, at different stages of the course presentation. However, since the interviews that discuss this theme were held during the latter stages of the course it was hoped that the students would have moved from this limited conception to more complex experiences, such as the ones described in the next three categories.

In Categories 3 and 4, *learning a new way of thinking* and *learning to problem solve*, it

Table 5.4: Understanding of learning to program; the focus of the conceptions.

	Referential aspect	**Internal horizon (foundation)**	**Focus**
1. Learning the syntax of the language	Learning the syntactical details of the language.	Code, programming instructions, programming language.	On the syntactical details of the language.
2. Learning and understanding the programming constructs	Understanding the programming constructs that extend beyond the current programming language.	Programming constructs, programming languages.	The programming constructs and when to use them.
3. Learning to write programs	Using all the gained knowledge to create programs.	Programs, user, code, programming constructs.	The complete solution to a problem in the form of a program.
4. Learning a way of thinking	Learning to think in a certain way required in programming.	Logical thinking, mathematics.	A "new way of thinking" that is required when programming.
5. Learning to "Problem Solve"	Learning to think in a logical way that conditions problem-solving.	Problem-solving techniques, algorithms, logical thinking.	Using this way of thinking to solve the problem at hand, though various techniques.
6. Learning a new skill	Learning a new skill that affects the way one thinks in real-life.	Logical, structured way of thinking.	A skill that reaches outside the boundaries of the course or even programming itself.

is observed that the referential aspect encompasses a view that is not intrinsic to the course content or structure. In Category 4, this way of thinking is expressed as logical thinking. The variation in the focal awareness of this category is that students have realised that there is something more to learning to program and that it is a systematic or logical way of thinking. Hence, the students felt that this is what one should learn, and concentrate on, when learning to program. However, the conception described in Category 4 is not complete, since this way of thinking is not clearly understood and the students refer to a "click" moment, where all prior knowledge falls in place. In the penultimate category the conception has become clear: learning to program is experienced as learning to solve problems using programming constructs and techniques. The conception has moved from the detail of the programming language to a more abstract understanding that is centred on the actual problem. The variation here is brought about by the fact that the language is seen as the means for solving the problem, and the different techniques one can employ to devise an algorithm that will solve the problem. Although this view was not explicitly part of the course, it was encouraged and fostered throughout the course, especially when programming tasks were assigned to the students.

In the final category, learning to program is viewed as acquiring a new skill. This conception presupposes the foundations discussed in the previous categories and concentrates instead on something external to the components of programming. This complex conception views the process of learning to program as learning a new skill, that is not only required when programming and solving problems but that changes the way a person thinks and acts in life. This structured and logical way of thinking, which was initially outlined in Category 5, is now put in the context of the learner's everyday life. Those in this category have discerned the skills that are inherent in learning to program and are now using them in other courses or activities, such as solving mathematical problems, learning other programming languages or everyday activities that require a structured way of thinking.

5.2.2 Discussion on Learning to Program

In the two sections that follow we discuss the outcome space identified for the phenomenon of learning to program; in relation to other phenomenographic studies on *learning to program* and the experience of *learning* respectively. By comparing and discussing the similarities and differences of the derived outcome spaces we provide an insight into the relevance of the findings and consequently draw a comprehensive picture of the results within the context of the field.

5.2.2.1 Studies on Learning to Program

The experience of learning to program among novice programmers is a popular phenomenon that has been investigated in a number of studies, namely (Booth, 1992; Bruce et al., 2004; Eckerdal, 2006) and (Stamouli and Huggard, 2006). These four studies have been conducted in isolation and with different student cohorts and degree programs (Computer Engineering, Information Technology, Aquatic Marine Engineering, and Computer Science respectively). Despite this difference in the educational settings, we can observe a number of similarities in their results as they all investigate the same phenomenon: the experience of novice college students when learning to program. In order to illustrate the similarities and differences among the four phenomenographic studies diagrammatically we have constructed Figure 5.2. This shows how the categories of the different studies are related and how they span the learning to program experience continuum.

The outcome space of the four studies can be logically separated into three sections: the categories that focus on the pragmatic elements of the experience, the conceptual categories and finally the meta-categories that transcend the experience of learning to program itself. This separation is denoted by dashed lines in Figure 5.2. In some studies the categories are more condensed than others and thus a single category from one study may summarise a number of conceptions that appear in more detail in another. An example of this is Eckerdal's first category of description, which synopsises the meaning of the first three categories of our study and two categories from Bruce's and Booth's studies. These differences can be attributed to the different settings of each study as well as to the background and course of each student cohort involved. In our study students were very

Figure 5.2: An effective comparison of four phenomenographic studies on learning to program and the relationships between them.

Study:	**Stamouli**	**Eckerdal**	**Bruce**	**Booth**
Course:	Computer Science	Aquatic and Environ-mental Engineering	IT	Engineering
Novice experience of learning to program			1.Following.	
	1. Learning the syntax of the language.		2. Coding.	1.Learning a programming language.
	2. Learning and understanding the programming constructs.	1.Understand the programming language and use it to write programs.	3. Understanding and integration of concepts.	
	3. Learning to write programs.			2. Learning to write programs in a programming language.
	4. Learning a new way of thinking.	2. Learning a new way of thinking.		
		3. Learning is to gain understanding of how programs appear in everyday life.		
	5. Learning to problem solve.	4. Learning a way of thinking that enables problem solving.	4. Problem solving.	3. Learning to solve problems in the form c programs.
			5. Participating/ enculturing.	4. Becoming part of the programming community.
	6. Acquiring an new skill.	5. Learning a new skill.		

specific on the different pragmatic elements of their experience of learning to program, and this resulted in exposing many more facets of the experience. Hence the richness of the interview data determines the detail and number of categories found in different studies.

In other cases some categories appear in only one study. An example of this is Bruce's first category *"following"* where learning to program is experienced as getting through the unit and getting a good grade on it (Bruce et al., 2004). The same argument used above in relation to the richness of the data could be applied here also. However, the specific nature of the category in this study is rather different. It could be said that this category illustrates the *motivation for trying to learn* how to program rather than students' experience of *what learning to program means.* In Berglund and Eckerdal's study on the motives that students have when taking a Computer Science course, a very similar motive is identified under academic achievement (Berglund and Eckerdal, 2006). Thus the researcher's views of what constitutes the phenomenon under investigation also affects the results. There is only one case where the same experience is explicitly encapsulated in a similar category in all studies and that is learning to problem solve.

We argue that the reason for the differences in the way the categories are expressed and appear in the experience of the same phenomenon is due to the difference between the student cohorts in terms of background, course content, environment, programming language and, in some, cases the researcher's view of the phenomenon itself. As Booth explains in her thesis when she talks about the reliability of the results of a phenomenographic study,

> *"The reliability of a phenomenographic study has much in common with the reliability of a journey of exploration. Such a study is in many respects a process of discovery [...] The results are in the end a description of the territory in terms of what has been seen and experienced. It would not be expected that a second explorer, even charged with the same task, would tackle the journey in the same way, and might therefore arrive at different description."* (Booth, 1992, p. 67)

She continues by saying that it would however be reasonable to expect the second researcher to arrive at very similar results with the same set of data. The four studies described above have investigated the same phenomenon in very different environments and while the results

differ in some aspects they show marked similarities in others.

5.2.2.2 Studies on Learning

Apart from specific qualitative studies on *learning to program* there has been much research interest over the last three decades on the experience of *learning* in general. These studies include (Säljö, 1982) and (Marton et al., 1993), where longitudinal and very thorough phenomenographic studies have been conducted on learning among students on a Social Science foundation degree course in Britain. However it is evident that such studies are profoundly influenced by the particular educational context in which they are conducted; hence the results found in this study are related to Marshall et al. which was conducted in an Engineering and Science context. The qualitative conceptions that were identified in their study are presented in Table 5.5, (Marshall et al., 1999). The authors' original notation (A, B, C, etc.) and terminology is retained in the table and throughout the discussion that follows.

Table 5.5: Marshal *et al.*, on students' conceptions of learning in an engineering context.

Conception	Description
A	Learning as memorising definitions, equations and procedures.
B	Learning as applying equations and procedures: learning is experienced as the ability to apply some knowledge.
C	Learning as making sense of physical concepts and procedures: learning is experienced as active construction of meaning, an 'activity within the mind'.
D	Learning as seeing phenomena in the world in a new way: learning is experienced in terms of using the understanding of the concepts in the learning material to see phenomena in the physical world in a new way.
E	Learning as a change in a person: by interacting with the learning material - or with peers - a student may develop new ways of seeing phenomena in the world and this leads to change as a person.

The progression of the experience in the categories found in Marshall et al. shows a shift from a passive or pragmatic perception (A, B) to the construction of meaning for the learning material and real-world phenomena (C, D). This in turn leads to the

conception (E) where the person views learning as a life changing experience. Looking at these conceptions for *learning* and the ones found in this study for *learning to program* it is clear that there are many similarities in the qualitative meaning. These are illustrated in Table 5.6. The first three categories of description in our study show similarities to Categories A and B of Marshal et al. in that they are concerned with the pragmatic elements of the experience. Category 4 is more in tune with C as learning is described as an activity through the mind. In 5 the focus has moved to the problem itself and the actions one should follow to model this problem through programming. Conception D shares some elements with Category 5, in respect to the modelling of real-world phenomena or programming problems. The final category found in our study's outcome space shares a great similarity to conception E as acquiring a new skill constitutes a change in how a person would react to the world phenomena.

Table 5.6: The categories of description found for learning to program in relation to the Marshall et al. conception of learning.

Category of description on learning to program	Related conception from Marshall et al.
1. Learning the syntax of the language	A
2. Learning and understanding the programming constructs	A, B
3. Learning to write programs	B
4. Learning a new way of thinking	C
5. Learning to problem solve	D
6. Acquiring a new skill	E

In these two sections we have discussed how the conceptions for learning to program,

as they were seen in the study's population, relate to other phenomenographic studies that investigate the same theme. Although some of the details and focus of the conceptions change among these studies, the spectrum of the experience and understanding is completely captured. The categories of description identified in our study also showed important similarities to a more general study of learning presented in (Marshall et al., 1999), underlining the depth and reliability of our findings.

5.3 Understanding of Correctness

An understanding of correctness is an important aspect of the experience of learning to program, since it affects the approach a learner adopts. This theme was taken up in the last interview session, which took place after the students had received feedback on almost all their assignments and laboratory work and at a point where their understanding of programming was reaching a more mature stage. The questions that were used to investigate their understanding were: *"When do you believe a program is correct?"*; and *"What is your idea of a correct program?"*.

Four qualitatively different ways of experiencing the correctness of a program have been identified: the first two categories focus more on the problem and the code correctness of the program while the last two are more developed, since they incorporate non-functional elements that aim to assist the actual user of the program. The categories are summarised in Table 5.7 and these are further analysed in the sections that follow.

Category 1: Syntactical correctness

In this category a program is experienced to be correct when it is free of any syntactical errors, in other words when it compiles and runs. The student focus in this category is solely on the code. Neither the problem requirements nor the human aspect of programming are involved in this perception. The experience is narrowed down to the relationship between the code and the programmer, as Liam explains in the following:

Interviewer: When do you think a program is correct?

Liam$_4$: When it has no bugs.

Table 5.7: Categories of description for *understanding of correctness*.

Category Label	**Category Description**
1. Syntactical correctness	A program is perceived to be correct when it is syntactically right, that is when it compiles and runs without any errors.
2. Functional correctness	Apart from being syntactically correct the program needs to fulfil the requirements of the problem specification.
3. Design correctness	In addition to the above, the program should be correctly structured in order to enable extendability.
4. I/O validation and performance correctness	As above, and also the program should cater for invalid input and it should also be optimised in terms of code length and how fast it executes.

> **Interviewer:** Anything else?
>
> **Liam$_4$:** No, not really, as long as it runs, it's right.

Alan, in the next quote, elaborates a bit more by saying that a program should use all the methods and classes that he has previously defined. However the focus of correctness remains the syntactical correctness of the program itself.

> **Alan$_4$:** When it works and the application runs and calls all the methods in the class and everything is fine and no errors.
>
> **Interviewer:** So when the syntax is right then it is correct?
>
> **Alan$_4$:** Yeah. I find that doing it in paper first is easier for me.

The understanding of when a program is correct in this category is restricted to the code and the language that is used. The student is satisfied with the correctness of the program when it runs, independent of the functionality of the program. Fortunately, not many students within the participating population shared this view.

Category 2: Functional correctness

This category of description expresses an understanding of program correctness where the problem requirements are the focus. The understanding that was expressed in the previous

category is still present; however the central point of the experience has expanded beyond the code to include the problem definition and the requirements of the application.

As Patrick explains in the following statement, program correctness is about satisfying the requirements of the problem:

> **Interviewer:** When do you think a program is correct?
>
> **Patrick$_4$:** I suppose when it satisfies all the things in the question and when it compiles successfully or when the red lines disappear from Eclipse [laughs]. I suppose it is right when it does what you want it to do.

Another opinion is expressed by Stephan in his final interview:

> **Stephan$_4$:** Well I suppose that the way I look at correctness is a bit different than what the lecturer believes. He thinks that a program is correct only if it follows the object-oriented way while I think that a program is correct when it does what it has to do. Also the source code has to look correct and readable.

The lecturer has been emphasising the importance of following object-oriented design principles when developing an application. However Stephan, who has prior programming experience with procedural languages, cannot see the point in object-oriented design as long as the necessary functionality is implemented; design is not considered to be part of a program's correctness here. Students in this category experience program correctness as being related to the code being error free but the primary focus is on the problem definition rather than on the code *per se*.

Category 3: Design correctness

In this case learners experience program design as the principal criterion for correctness; affecting the extensibility and readability of the program. Therefore, apart from all the other properties that should be present in a solution, the right structure of the program is key. Eamonn explains this as follows:

> **Interviewer:** I mean let's take for example your poker program, which were your criteria of correctness?

> **Eamonn_4:** I felt it was correct when it ran, did what it was supposed to do and it was structured properly. [...] I feel design is part of correctness. It is easier and better when it is correctly structured, because people can understand it... and because then you can go back and extend and reuse what you had there. So appropriate design is really important and it adds to the solution.

Brian explains how his perception of when a program is correct has changed after the exam.

> **Brian_4:** I used to just say when it does what is supposed to do but it was... Karl, when we had that sort of half exam just before Easter and it [his program] was right I mean what he wrote was right but it was missing something small and for that he got I think 10 out of 40 or something, so it is kind of like... it kind of has to be functional as well it has to be open and object-oriented and it has to be sort of modular [he writes down again] because he had it all in a big blob of code. So it is right if it is like... I don't like to think that... if the final result is right so the whole thing is right. It is how you got there, how you designed the thing to get there as opposed to correctness as such.

Brian says that he previously believed that program correctness was all about the functional elements of the program, but as his friend had marks deducted from his exam paper because his design was inappropriate, he changed his mind. Brian's focus in his answer is clearly on design and this view is unlike that expressed in the previous categories. In the last sentence of his response Brian makes a distinction between the end-product correctness and the process that one follows to achieve that. Thus, from his point of view what is important is the techniques used to develop a solution and that these are what constitutes the program's correctness. Declan is thinking along the same lines in his response:

> **Declan_4:** Well, if it works firstly and then the way the code is written and structured because sometimes code can look terrible, I mean really horrible, like if you got like i+= 1 I don't like it because it is very hard to read. Just, generally on how easy the code is to read and so on...

Although all of the students quoted above experience design to be an important part of correctness, their motivation is very different. Eamonn stresses the importance of design

because that makes the solution easier to extend, Brian concentrates on achieving good grades and separates syntactical and design correctness, while Declan points out that the right structure improves readability and therefore makes the program correct. These students share the same perception of program correctness, however they approach it from very diverse and different angles.

Category 4: I/O validation and performance correctness

In this category the perspective is broadened even more to include non-functional requirements as part of the experience:

> **Anthony$_4$:** When it fulfils the functions that it is supposed to do... without any side effects, it might be able to fulfil all the tasks but it should be responding in cases where the user enters something invalid.
>
> **Interviewer:** You mean checking for the validity of the user input?
>
> **Anthony$_4$:** Yeah.

Anthony here emphasises the importance of input validation and he experiences this as being part of a correct program. Thus, apart from the functionality that should be implemented, the learner considers non-functional elements such as error checking to be necessary when a solution is developed. Even though the focus is still on the problem requirements, there is more to correctness than just the basic functionality. Colin expresses this more strongly:

> **Colin$_4$:** A program is correct when it does what you want it to do first of all, so you give it the values you want it to use and then it just works. It works also when it is user proof so if you use the wrong values then you cannot crash it, you cannot pass values that would make it not work. You have to be able to respond properly when you don't give exactly what it wants. It should be able to distinguish among what is valid and what is not. It should also be what is the word... optimised. It has to be as short as possible to do it and it should also take up less memory and it should run faster.
>
> **Interviewer:** Did you come up with this by yourself, or you read it somewhere?
>
> **Colin$_4$:** Oh, by myself, from my experience.

Collin's understanding of correctness takes into account the behaviour that a program should have when it is executed. From his point of view the responses a program provides to the user should be meaningful. Hence, in order for a program to be correct, input and output validation is necessary. Colin also emphasises other non-functional properties such as code length and memory efficiency. Kevin emphasises the importance of providing clear guidelines regarding the user input and program usage:

> **Kevin$_4$:** When you run it and it does everything you kind of ask from it to do, say it looks for user errors, so if you ask for yes or no and you expect the user to type "y" or "n" then if the user types something else this would not crash the program, like, it will deal with that correctly. So sort of input and error checking is important as well.

Karl thinks along the same lines in his response:

> **Karl$_4$:** When it does what it is supposed to do without errors I suppose hmm... Caters for any error that might occur any problems, like yeah, problems that might arise from the user doing something that he is not supposed to rather than crashing and it is supposed to do whatever you expect it to do rather than something that you either don't need or don't want. It is correct when it solves the problem and prevents other problems from occurring.

All the above quotes reveal an understanding of correctness that incorporates the previous categories but focuses mostly on I/O validation and other non-functional properties such as optimisation and efficiency. The focus of the students' experience has expanded to involve the user as well, since the importance of validating input and output in this category aims to assist the user in using and acquiring a better understanding of the application.

5.3.1 Structure and Meaning of Students' Understanding of Program Correctness

Four distinct categories of description were identified among the students with regard to their views of program correctness. The students were asked what constituted program correctness within the context of the course rather than in general. Thus the qualitative categories of description were formulated with this in mind. The focus of the conceptions found for this theme are presented in Table 5.8. The internal horizon of the first two,

syntactical and functional correctness, reveal a conception that is focused on the more tangible foundations of the theme.

Category 1, *"A program is perceived to be correct when it is syntactically right, that is when it compiles without any errors"*, describes a conception where the central focus is the programming language and the syntax of the solution. The variation here is brought about by the different syntactical mistakes and structure of different programs. Students that share this viewpoint have difficulty understanding programming and in many cases remain puzzled when they do not achieve the grades they expect.

Table 5.8: Understanding of program correctness; the focus of the conceptions.

	Referential aspect	**Internal horizon (foundation)**	**Focus**
1. Syntactical correctness	The program is correct when it compiles.	Code, syntax, programming language.	On the syntax of the solution.
2. Functional correctness	The program is correct when it fulfils the requirements of the problem.	Syntax, problem specification, output.	On the completion of the functionality.
3. Design correctness	The program is correct when it is correctly structured.	Problem specification, object-orientation, design.	On object-oriented design and extendability of the solution.
4. I/O validation and performance	The program is correct when it caters for unexpected behaviour from the user.	Code, user, input, output, design, non-functional requirements.	On the interaction of the program with the user and robustness of the program.

Functional correctness was a popular category among this study's population. The fulfilment of the problem requirements is the focal point of this conception. The foundation of this conception is primarily the functionality that is presented in the problem specification, while the variation for this understanding comes from differences in the problem-solving activities. Even though this is a straightforward and very logical conception, students that experience program correctness in this way often fail to achieve their potential in a course such as the one under investigation. When learning object-oriented programming, an essential goal of the course is for students to learn to write and think according to the object-oriented paradigm. Not taking object-orientation into account results in incorrect or incomplete solutions. From the analysis of the interview data, it appears that students with previous experience in procedural programming languages failed to see the importance of the object-oriented paradigm.

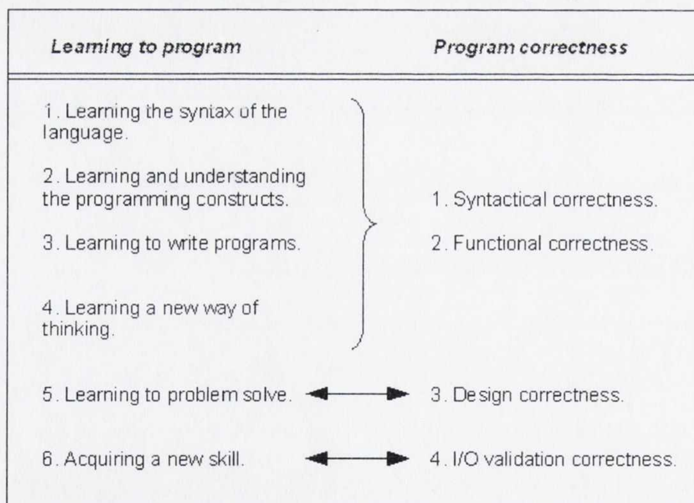
The next category, *design correctness*, presupposes the understanding expressed in the previous two conceptions and instead focuses on the design of a program. As illustrated in the foundations of this conception, students that share this view do not merely try to fulfil the requirements of a given problem, but try to follow object-oriented techniques and develop an extensible, and more reliable, solution. This conception reveals a richer understanding of what constitutes a correct program. This is clearly differentiated from the previous category and has been placed in a more realistic paradigm where programs can be reused or further extended. The variation here stems from the novelty inherent in the object-oriented paradigm that mandates an original view of a programming solution.

Finally, the category labelled I/O validation and performance correctness focuses mostly on the interaction between the program and the user. The foundation of the learner's conception has moved beyond the tangible elements of correctness and is now concerned with non-functional aspects of the program. The actual user is central to this conception and this enables the students to experience a correct program as something that solves real-world problems and therefore should be designed to interact appropriately.

5.3.2 Learning to Program and its Relationship With Program Correctness

When observing the outcome space for the constructs of learning to program and program correctness at the level of individual students, we find that the way students experience learning to program is related to their perception of what constitutes a correct program. As illustrated in Figure 5.3, the conceptions of the individual students whose experience of what it means to learn to program falls into the first four categories of description, experience program correctness as either syntactical or functional. A one-to-one relationship was not established between the two themes: from the nine students that fell into the first four categories of the first theme, all, except one, viewed program correctness as syntactical or functional. Thus, the way one experiences learning to program influences how one understands the outcome of that process, i.e. the program itself. In the first four categories of the learning to program theme, the experience is confined to the programming language and the programs. Similarly, for the program correctness theme, the first two categories focus on the language and the problem at hand, rather than taking the bigger picture into account.

Figure 5.3: Learning to program in relation to program correctness.



For Categories 5 and 6, (Learning to Problem Solve and Acquiring a new skill) a one-to-one relationship is observed with design correctness and I/O validation and performance

correctness respectively. The distribution of the seven students that belong to the aforementioned categories, is that three experienced program correctness as design correctness, while four experienced programming as I/O validation and performance correctness. Thus, when the focus of learning to program is on the structured way of thinking that enables problem-solving, the primary criterion for program correctness is the object-oriented structure and extendability of the solution. Finally, when learning to program is conceptualised as the process of acquiring a new skill that can be used in real-life; the understanding of program correctness is focused on the interaction between the user and the program, taking into account non-functional elements such as optimisation of the code and performance of the program as a whole. Hence, the results suggest that there is a clear linear relation between the two themes.

Although, the study's sample population is not sufficiently large to draw final conclusions on the relationships observed, it is sufficiently representative to indicate the existence of this trend. Therefore, the results suggest that students develop a general view about learning programming and the programming constructs, and that this then influences their experience throughout the course and, possibly, their undergraduate career.

5.3.3 Program Correctness in the Literature

The notion of correctness is an elusive one as it greatly depends on one's background, experience and standards. Nevertheless numerous research studies have been conducted on the habits and norms novice programmers employ to test the correctness in their programs. Specifically Scott et al. found that students do not test their systems to the same level that would be expected from industry. This is because they do not have the required knowledge to do so and, furthermore they do not value testing as highly as the industry would (Scott et al., 2004). Other researchers such as (Fleury, 1993), have found that college students' understanding of programming differs from that of experts and, as was presented in the previous section, this was found to extend to their standards of correctness. Although many studies have looked at the errors, either syntactical or logical, that novice programmers make when learning to program e.g. (Spohrer and Soloway, 1986; Barr et al., 1999; Hristova et al., 2003), very few have looked at what students perceive to be a correct

program, which makes it hard to compare our results in great detail.

One recent study did investigate how students define correctness. It was conducted among high school and college students through the use of questionnaires (Ben-David Kolikant, 2005). Her conclusions regarding the definitions of correctness are the following:

- A correct program is a working program.
- A working program exhibits reasonable I/O for many legal inputs.
- A reasonable output should be correct, but also in cases of incorrect output the program should display something that is expected.

The progression of the reasoning that the students have followed in this set of results is somewhat similar to the progression of understanding in our categories of description. However these results cannot be compared in any great depth and detail to the ones presented in Table 5.7, although they do illustrate that similar perceptions appear within a different educational context.

5.4 Summary

This chapter presented the conceptions novice programmers have of the theoretical components under consideration. These were the *nature of programming*, *learning to program* and *program correctness*. The phenomenographic outcome space for each of the components has been analysed with supporting excerpts from the interview sessions used to present the categories of description in relation to the individuals. The meaning and structure of the categories for each theme have also been discussed, illustrating the overall picture of the components. The relationship of each theoretical construct to other research projects in the area of Computer Science education has been discussed, demonstrating the similarities of our findings to those from other educational contexts.

Chapter 6

The Object-Oriented Components of Programming

In this chapter we discuss our findings regarding the object-oriented components of the study. The themes that are analysed deal with students' understanding of the following constructs: object, class, attribute, method and constructor. These are undoubtedly key constructs in the object-oriented paradigm. In most programming modules students are introduced to these constructs at the beginning of the course. Thus it necessary for them to gain an understanding of these in the early stages of their studies. This holds true for the course where we conducted this study since the adopted teaching approach was *objects first.* Due to the fact that these concepts are very closely inter-related, they were investigated together at two separate interview sessions, namely in sessions 1 and 4. The former was at the end of the first term while the latter took place at the end of the third term when the students had been using objects and classes extensively in their programs for a significant amount of time. Although it was expected that a significant variation would be observed between students' conceptions in the two interviews sessions, it was evident that the observed shifts in their understanding were minimal. As discussed in Section 6.6, these shifts may be classified as spontaneous and triggered intra/inter-contextual shifts as defined by (Pong, 1999) and (Berglund, 2005).

It was observed during the interviews that students often tended to explain and describe the object-oriented constructs in relation to one another. Thus, in some of the excerpts used

in the following analysis their responses may capture their conceptions for both objects and classes, or attributes and methods. Another issue that arose during the interviews on these themes was that students preferred to discuss them by giving specific examples. This can be attributed to the abstract nature of these concepts. Where students felt they needed a reference point to use when describing their understanding often they referred to previous exercises they had completed in tutorials or as assignments.

The object-oriented components included in this study are analysed from the phenomenographic perspective in the rest of this chapter. A general discussion of the results for each construct is provided, and the chapter concludes with an overview of the students' shifts in understanding.

6.1 Understanding Object

The fundamental idea behind object-oriented programming is that programs are comprised of collections of interacting objects. Hence, when students first become familiar with this programming paradigm, they are required to think in terms of objects and classes and to use them extensively in the development of programs. There are many definitions of what an object is in the literature, and the students in this study were introduced to many of these during their programming course. However the definition that we feel captures the essence of them all comes from Conway, who states (Conway, 1999) that *"[a]n object is anything that provides a way to locate, access, modify, and secure data. [...] But in the more general sense, anything that provides access to data may be thought to be an object"*.

For some this may appear to be simple to understand, however students seem to have very diverse experiences of what constitutes an object and what it *really* is. Due to the abstract nature of the concept, a number of different questions were used to investigate the students' conceptions of the construct. These were: *"What do you think an object is?"*, *"How do you understand the concept of objects?"* and *"What is the difference between an object and a class?"*. In addition to the data obtained from the two interview sessions, students were also given a problem and asked to decide how many different objects they would include in their solution (see Appendix B, problems 1 and 2). The data from this written exercise served as a staring point for further discussions and clarifications.

Table 6.1: Categories of description for *understanding of objects.*

Category Label	**Category Description**
1. Object as code	Object is experienced as a piece of code.
2. Object seen as a programming construct	**a.** Object is experienced as a programming construct that is designed to hold values and group behaviours. **b.** Object is considered to be an active entity in the program that contributes to its design.
3. Object is seen as a model of real-world phenomena	As above and, in addition, object is understood to be a representation of real-world phenomena.

Table 6.1 summarises the categories of description of the construct of object that have been found and also shows the subcategories identified for the second category. In this second category of description an understanding is voiced that objects are experienced as programming constructs of the language. The difference in the role and purpose of the object in a program forms the basis for the creation of the two subcategories.

Based on the data gathered, the understanding outlined in the first category is not included in the other categories. However, a logical inclusiveness is presumed with respect to the latter categories due to the simplicity of the former.

Category 1: Object as code

In this case the learner experiences an object as a piece of code. The focus of the conception is limited to the code as a set of instructions. Sean says:

> **Sean$_4$:** [...] an object would refer to those different attributes and then you put values to these different values. It's just code..

Stephan explains how he understands an object by giving a programming example and more specifically by writing down a code example in the following:

> **Interviewer:** What do you think is an object?
>
> **Stephan$_4$:** Like if you have a Car class (he writes down how a class is defined) then an object would be `Car mazda = new Car()` and then you call the constructor. So it is part of the program I guess...

While Stephan explains his understanding of what an object is by using the appropriate Java syntax for creating one, Karl explains what an object and a class are in terms of code length in the following:

> **Karl $_1$:** [...] When you create a class you have a big file with all the information about that class, when you create an object you do it with a single line. Right? [...] [S]o an object is a single item which is representing the class and contains all the characteristics of a class. An object is the way of using the class within the program, it's a line of code.

This conception was not widespread among the population. Nonetheless, it is a distinct way of experiencing the concept of an object, which is highly focused around the programming language and the specific syntactical details used to define an object.

Category 2: Object seen as a programming construct

This category of description expresses an understanding of objects as a programming construct. There were two subcategories identified that belong within the same framework of the category, but which expose the different roles of the object.

a. Object as a construct

This subcategory is strongly related to Category 1 since a programming construct is part of the code, however here the understanding is better developed. Object is experienced in a coherent way, as a construct that can hold multiple values and has a role in the program; as Alan explains in his answer:

> **Interviewer:** So you say class is the template, then what is an object?
>
> **Alan$_1$:** The object just holds the values.

In the third session, while completing a written exercise, Alan said that objects are used for holding values for every attribute in its respective class. Liam expresses that understanding of objects as programming entities in a clearer way in the following:

> **Interviewer:** What is an object then?

> **Liam $_1$:** It is a particular instance of a class, but you take the abstract definition of the class and you put the values in, and you reference the object and you act on that object, like with Integers and stuff.

Liam associates the way objects are used with the way `Integer`, a built-in construct in Java, is used in a program. Neil describes an object in terms of the information it holds and clearly expresses an understanding of object as a programming construct in the following.

> **Neil $_1$:** An object is an entity, that pretty much is the dictionary definition of an object it has attributes to it and information about certain parts of it. You can have several objects that are the same type of object but they have different features... it's a construct in Java.

The understanding presented in this subcategory is more developed in the sense that objects are not just seen as arbitrary lines of code, but are rather experienced as coherent programming constructs that serve a purpose within the program.

b. Object as an entity in a program

Here an object is seen as an entity that is important in the design of the program since it provides structure to the program. Patrick explains this plainly in the following:

> **Interviewer:** And then an object is...
>
> **Patrick $_1$:** Is an entity with the program that has values for these attributes and plays an important role in the overall program.

and again in his fourth interview, he stresses the point of having objects for handling functionality in the program.

> **Interviewer:** How do you approach a problem, do you first look for classes or are you looking at the functionality required?
>
> **Patrick $_4$:** The first thing I do is like find the classes, I look at the most basics and then what I need to create. Like in terms of objects you are not going to be getting anything more basic than the object like. So I do the objects first because the lecturer is very particular about how you identify and define objects and stuff, having structure

within the programs. So I thought like you have a Card object and then you think about the attributes and stuff and then you have another class for handling the Cards, it's like... it seems that is the best way of doing it structurally. You first have the Card and then you have all the other classes that go with it I guess and then I kind of work on the methods from there.

Finally, Mark says that an object can be anything that has properties and functionality and comprises a complete entity.

> **Mark $_4$** : Hmm anything... a thing that has properties and generally methods that can be applied to that thing as well and to its properties I suppose everything that can be described as a complete thing.

All the above excerpts denote an understanding in which an object is viewed as a complete entity that contributes to the structure of the program. When compared to Subcategory 2a we can clearly observe the same foundation for the understanding, although the role of the object is altered. In the first subcategory the role of the object is to hold attributes and values, while in second one an object is mainly seen as an active entity that contributes to the design of the program. Thus, the two approaches of understanding result in the same category but have different starting points.

Category 3: Object as a real-world phenomenon

This final category draws on the previous one in the sense that these entities are now associated with real wold phenomena, like physical objects. Students who share this understanding have also demonstrated the understanding presented in Category 2, so these last two categories are inclusive. Brian, when asked in reference to a problem if "Red" would be a class or a value for an attribute of the class `TrafficLights`, responded with the following:

> **Brian $_1$:** Well Red it is more of a description, while traffic light is a physical object. So classes are usually physical objects of the real-world. So door could be a class and your blue door [points to my office door] would be an object, and then open would be a value of the attribute status.

He makes a similar point in his fourth interview:

> **Brian $_4$:** An object is just a... like you would have the class and then you can have many objects, it's kind of hard... because you can't talk about objects without referring to the class as well and the other way around. If you have your class... it's basically your rules for the object so the object is just a thing that fits within this rules, like if you think about it as a physical example of the class.

Eamonn's understanding incorporated the views presented in the previous category but moved beyond them to associate the conception of object with real-world phenomena.

> **Eamonn $_4$:** Object is everything you can represent as a complete thing. A person is an object, a chair is an object, everything in this room can be an object in a program.

In his fourth interview Karl expresses a different understanding to that in his first interview, in that it extends beyond the physical description of objects:

> **Interviewer:** What is an object?
>
> **Karl $_4$:** Could be anything really .. it doesn't necessarily have to be a physical one it could be a concept, a list for example.

The conception presented in this category encapsulates an understanding in which an object is viewed either as a physical entity or a phenomenon of the real-world. This illustrates a more advanced understanding since it puts the object into a wider perspective, opening the horizon for a broader, more efficient usage of the construct.

6.1.1 Structure and Meaning of Students' Understanding of Object

The analysis of the students' understanding of objects revealed three distinct categories. The meaning encapsulated within the categories of description and their focus is summarised in Table 6.2. Like the structure and focus of the categories analysed in the previous chapter, the students' understanding of object moves from a level of relatively poor understanding such as that presented in Category 1, to the more complex and multidimensional conceptions summarised in Categories 2 and 3. This is also evident when observing the internal horizon of the categories, where the foundation of the conceptions move from

the obvious (e.g. code and syntax) to the mores fundamental elements of object-oriented modelling and real-world objects.

Looking at each category separately, when an object is experienced as a piece of code, the students' understanding assumes a relatively poor and naive format. The focal point of this first conception is on the actual line of code that is used to create and manipulate the object. The variation for discerning this understanding is brought out by the different ways objects appear in specific programming examples. Therefore, the focal awareness in understanding an object as a piece of code stems from the ways that the various objects appear and are used in programs.

In the first subcategory of the second category, object is experienced as something more coherent than just lines of code. Here object is perceived as a programming construct, that can hold multiple values for its attributes. The foundations of this conception are still close to the programming language, but they are more in tune with the object orientated paradigm of the language. The dimension of variation here is brought out by the way other programming constructs are viewed when compared to programmer-defined objects. For example, `Integer` is a programming construct and a Java built-in defined class. The variation between such programming constructs and custom-defined objects that can store different sets of attributes bring out the variation necessary for this understanding.

In the second subcategory, object is viewed as an active entity in the program. Here the foundations of the conception remain the same, although the role of the object is experienced in terms of the design and structure of the program. The focal point of this understanding is that object is a complete entity that acts as a medium for utilising the necessary functionality for the program to achieve its goal. The basis of the category remains the view of object as a construct, but the scope differs. The variation in the understanding is brought about by the way object actively shapes the program and the functionality within it. Thus the different ways object is used and the values it takes brings about the variation in the focal awareness.

In the final category, objects are experienced as representations of real-world phenomena. This category encapsulates a richer understanding since it includes the conceptions of the previous categories while developing the notions of objects further by relating them

Table 6.2: Understanding of objects; the focus of the conceptions.

	Referential aspect	Internal horizon (foundation)	Focus
1. Object as code	Object is understood as part of the code.	Code, syntax, programming language.	On the actual instructions used to create and manipulate an object.
2. Object as a programming construct	**a.** Object is experienced as another programming construct. **b.** Object is considered to be an active entity that contributes to the structure of the program.	Programming construct, object-orientation, programming language.	On the values that the attributes of different objects hold. On the object as a coherent entity that acts as medium for utilising functionality necessary for the program/algorithm.
3. Object as a real-world phenomenon	Object is experienced as a model of real-world phenomena.	Problem specification, design, real-world objects, object-oriented modelling.	On modelling objects from real-world objects and phenomena.

to real-world phenomena and objects. The foundations of this understanding include the problem specification, since it is from there that students can recognise and determine the necessary objects for their solution. Real-world objects and phenomena that objects are modelled on, are also in the foundation of this conception. The dimension of variation here is the objects that are found in real-life and can be modelled in programs. The variations in these constitute the focal awareness.

6.1.2 Discussion of the Concept of Object

The analysis of the three categories and two subcategories of description have been presented. These were derived from interviews conducted with the students both at the start and the end of their acquaintance with objects as part of the course. From the point of view of the course and instruction outcomes, the understanding summarised in all three categories is both desirable and necessary for the students to acquire: since an object is a *piece of code*, and it is used to *hold values*, while at the same time it is a *construct* that can represent *real-world phenomena*. Problems may arise in cases where a student's understanding reaches that presented in Category 1 and does not progress from there. Very few students articulated only the conceptions of Category 1 during the last interview, however the snapshot nature of this study cannot tell us if, or when, their conceptions matured at a later stage.

In the literature, only one study has investigated students' understanding of object and class (Eckerdal et al., 2005). Eckerdal conducted a phenomenographic analysis on the two themes of object and class drawing on interview data from students who were enrolled on an Aquatic and Environmental Engineering course. It should be noted that in the final analysis that is presented in (Eckerdal et al., 2005) and (Eckerdal, 2006) the outcome spaces for the two constructs are combined and their implications are discussed in parallel. However, in order to be able to draw parallels with the findings of our study they will be presented separately here and in Section 6.2.2 below.

Three categories of description were identified for object among the population in Eckerdal's study (Eckerdal, 2006):

1. Object is experienced as a piece of code.

2. Object is experienced as something that is active in the program.

3. Object is experienced as a model of some real-world phenomenon.

The categories bear a great similarity to the ones identified among the Computer Science population of this study. The first category in Eckerdal's study summarises the same experience as the first identified in this study and presented in Table 6.1. The second category varies significantly. In (Eckerdal et al., 2005), this category is described as a result of program execution and the task at the object in run-time. In our study this is conceptualised in Subcategory 2b, while Subcategory 2a (object experienced as a programming construct) was not identified within Eckerdal's outcome space. The third category in both studies captures the same conception where object is experienced as a model of a real-world phenomenon.

The two studies were conducted independently, in isolation from one another, at different points in time, in different contexts and with different sample populations. Nonetheless the results show many similarities and few differences. The differences in the studies may be attributed to the background and course of the student population. The fact that more detail was observed in Subcategories 2a and 2b of our study may be attributed to the technical background of the course in general and the prior programming familiarity that most Computer Science students have. The similarities between the two outcome spaces highlight a very important attribute for the results of this study as a whole; the fact that they are *generalisable.* Considering the diversity of the two courses and the settings of the studies, the similarity of the results leads us to argue that the findings can be generalised to other groups of students that are taught object-oriented programming.

6.2 Understanding of Class

Like objects, classes are a fundamental building block of object-oriented programming. Given the definition of a problem students need to able to recognise which elements can be represented as classes and also construct classes in a way that simplifies the solution by providing structure and meaning to the program. Due to the close relationship between object and class, students often tended to describe one using the other as a reference or part

of their definition. We discuss classes in two interview sessions, 1 and 4, at the beginning of the course and at the end. Data from session 2 where students were required to solve programming problem out loud (see Appendix B) was also used in the analysis. The questions used to investigate students conceptions of class were: *"How do you understand the concept of a class?"*; *"When you have a problem, how do you pick what is going to be a class?"*; *"How would you describe what a class is to someone that didn't know about programming?"*; and *"What is the difference between an object and a class?"*.

Table 6.3: Categories of description for *understanding of class.*

Category Label	Category Description
1. Class as code	Classes are experienced as a piece of code.
2. Class provides structure to the program	Classes are experienced as entities that contribute to the structure of the program.
3. Class is a template for objects	Classes are understood as templates that model the attributes and methods of objects.
4. Class is a model of real-world phenomena	As above, and also classes are understood as types for objects that can be found in the real world.

Four qualitatively different ways of experiencing classes have been identified among this study's population. The categories follow a similar pattern to the one observed in the analysis of the construct of object. Not all categories have been found to be inclusive. Based on the data, only Categories 3 and 4 were found to be fully inclusive, however an underlying logical inclusiveness is assumed. The categories are summarised in Table 6.3 and are further analysed in the sections that follow.

Category 1: Class as code

In this first category of description, the construct of class is experienced as a piece of code. Often students in this category describe a class in terms of the attributes and methods of which it is composed; as Cormac says in his response:

> **Interviewer:** So how would you define a class?
>
> **Cormac$_1$:** A grouping of attributes... I don't know.

Anthony is more specific in his response, since he describes class as a block of code in the following excerpt.

> **Anthony$_1$:** A class is... It is very hard to describe like... It is basically a block of code that describes what the object is made from. It is a description of the object. Classes are basic building blocks, while objects are instances of these.

Sean expresses this conception in an even clearer way when he specifically relates concept of class with the syntax of Java.

> **Interviewer:** What is your understanding of a class?
>
> **Sean$_4$:** It is a type in Java you can have various methods in a class just that it is a type in Java.
>
> **Interviewer:** So then the class is an actual type?
>
> **Sean$_4$:** No it defines a type more or less, it is not a type itself. Well say for a example that there isn't any `RationalNumber` type in Java so you use the class to define that type, so in a sense you create that type using the class.
>
> **Interviewer:** What did you have in mind when you were describing this to me?
>
> **Sean$_4$:** Hmm I thought of code mostly.

Students that share this conception describe their views from different angles but the focus of the conception is clearly limited to code and Java. The experience of class in this category is consistent with the description of object in the first category dealing with objects.

Category 2: Class provides structure to the program

Category 2 describes an understanding in which class is viewed as a construct necessary for constructing and structuring a program solution. A class is also described as something that plays an important role in the problem, and therefore should form part of the solution. Declan illustrated this very clearly when discussing an assignment that required the construction of a dating program:

> **Interviewer:** So in your dating program you had a class Person. How did you identify that Person should be a class?

> **Declan$_1$:** I am not sure really I suppose it depends on the program you want to write... Aa... yes the program that you want to write so I am going back to the dating (program) one, you want to use the program for the person and so wherever you want to use the program on, person is part of the program... and therefore I will use it as part of the solution, so I have a class. [he writes down the syntax of the class] [...] Ohm so it (person) would be inside such a program and would be necessary to know his information. So when something is important and can be broken down to its attributes it becomes a class I suppose. Hmm so for everything you need to know more information about, can become a class. That's the best I can give you for definition! [laughs].

Eamonn stresses the fact that a class is a collection of attributes and methods but it has to represent something that is coherent and has a meaning in the solution of the problem.

> **Interviewer:** Okay so what is your understanding of the concept of a class?
>
> **Eamonn$_4$:** Classes?
>
> **Interviewer:** Yeah.
>
> **Eamonn$_4$:** They represent some objects that you can use in your normal programs.
>
> **Interviewer:** So given a problem how do you identify the classes?
>
> **Eamonn$_4$:** You just decide what you are going to need to solve the problem and then decide the attributes the problem might posses and then you write a class with the methods and attributes you need to solve the problem, like your variables.
>
> **Interviewer:** So you say that you use a class as a collection of methods and attributes then?
>
> **Eamonn$_4$:** Well it should be something coherent as well like a class should be something to represent something and that would hopefully give you the means to solve the problem as well.

Eamonn illustrates an inclusive understanding of Category 1 but his understanding is more developed and better aligned with the current category. The last excerpt comes for Stephan, where he described how he decided on the classes to use in his solution.

> **Stephan$_1$:** So in a given a problem ... I decide on what are the really essential entities needed and then represent them as classes, and maybe after that ... maybe some complementary classes to make the design more clear.

Category 3: Class as a template for objects

In this category of description a class is understood as serving as a template for the object. Classes are viewed as descriptions of objects and are used to create multiple instances of them in a program. Patrick expresses this in the following way:

> **Interviewer:** How do you understand the concept of a class?
>
> **Patrick $_1$:** Usually I guess a class is a template for creating an object and an object is a particular entity within a program that basically I guess it is an entity that plays a role within your algorithm, your overall program. A class is more general. It is a template for repeatedly building objects. So a class is a generalisation of an object itself.

Patrick stresses the fact that objects are important components inside programs and algorithms, while the understanding that a class serves as a mould for creating new objects is clear from this response. Alan expresses the same understanding, focusing more on the values that the individual attributes would take when creating different instances of a class:

> **Alan$_1$:** I would describe a class as a general template for creating lots and lots of different types of the same object... No wait... it's the same type of object but different attributes each time. Like you would have a class for a `Person` and for each person you'd have to define and create a new one with their eye colour, hair colour, height, age etc. you can just contain all the list of possible variables you would have for what it is that you're creating.

Later in his fourth interview when the topic of class was revisited he added to this conception, saying that a class is like a blueprint for an object, and once you have defined them you can use them to act upon the methods that the class specifies.

> **Alan$_4$:** A class is a generic type, a generic way of creating many types of the same class. A class would... let's say rectangle would like... let you create many different types of rectangle... sorry many instances of the class rectangle. And then it would let you create these objects and call these methods from within them so that you make them do what you want them to do.

Finally Brian uses the same terminology of "blueprint" to describe his understanding of class and in his response it appears that he has developed a hierarchy of how the concepts of object and class are related to each other.

> **Brain$_1$:** A class is just like a blueprint for objects. A class is basically a list for things ... whatever you want to describe, you do it with a class. The object comes from the class so you can have whatever object you want once you define the class. Also, the class is the higher order from the object.

In this category, the concept of class is experienced as a template for the object. As such, it is related to the programming language and the concept of object, but also to the overall program. The excerpts from the interviews mostly discuss the technical issues of how a class is used to create multiple instances for objects.

Category 4: Class a model of real-world phenomena

In this final category a class is perceived as something that models real-world entities and phenomena so they can be used and represented accordingly in a program. This is clearly expressed in Declan's response in the following:

> **Declan$_4$:** A class is hmm .. something that represents an object because it is a way of representing something into a computer, because you can't describe what a person is made up... to a kind of... you can't really describe it any other way. So when you need to represent something from the real word then you kind of have to use a class to describe it.

Declan expresses the understanding that was summarised in Category 3, but the overall meaning and focus of his response illustrates the fact that his experience of the construct class goes beyond that and extends to the real-world. Mark, in the following excerpt from his fourth interview, initially appears to perceive class as a template for objects, however later when he is asked what it signifies when something is a class, he reveals an understanding that relates classes to real-world objects and phenomena.

> **Interviewer:** What is your understanding of a class?

Mark$_4$: Hmm it is basically a blueprint of creating lots of different instances of class... no lots of different objects from it hmm it just gives you the basic information you need to create an instance of a class and then it is just a blueprint.

Interviewer: Do you usually approach a solution to a problem by identifying the classes first or by looking at the functionality?

Mark$_4$: I would start by thinking first of the classes and then the rest of the things.

Interviewer: And what signifies when something is a class?

Mark$_4$: Hmm it depends on the problem and how they are on the real-world as well. Like for the poker I had (a class) `Card` and `Hand` and then the program, because cards are real objects and then the hand is something different as well, so I thought this was the way I should represent it.

The conception of class that was presented in this category proclaims an understanding that is more in tune with the real-world, which means that it goes beyond the boundaries of the program and the programming language to include the metaphor of real-world modelling in the programs. This understanding is inclusive of Category 3, however the focus is on the real-world entities that can be described through the use of classes.

6.2.1 Structure and Meaning of Students' Understanding of Class

Four categories of description have been observed for the students' understanding of the concept of class. The categories follow a similar pattern to the ones identified for object, but there is an extra category as the constructs may be interrelated but at the same time they are quite different as well. Table 6.4 shows the structure, foundations and focus of the conceptions found.

In Category 1, class is understood as a piece of code that describes the behaviours and characteristics of the object. This conception is not wrong and cannot be described as a misconception of class, but it is a partial view of the theme closely related to the code and the syntax of the language. In this category the need for user defined classes is caused by the fact that some types are not built-in to the programming language. Therefore, a class is perceived as a piece of code that defines a type for a construct (like `Person`, or `RationalNumber`) that is not already defined within Java but is required for the program.

The dimension of variation is the textual code representation of a class and the variations in the focal awareness derive from the different classes there may be in a program.

In Category 2 the conception of class becomes more coherent, as a class is now perceived as a complete construct that is used in the program to provide structure and design. A class in this category is considered to represent the important elements of the problem definition and is used as part of the solution to it. The foundations of the conception follow the same lines, as shown in Table 6.4. The variation in the focal awareness is brought about by the different programs and the alternate ways that classes are used in them.

Category 3 captures how a class is experienced as a template for creating multiple objects. The focus of this conception is on modelling the behaviours and characteristics of an object in a construct that allows for the creation of multiple instances of it. This conception brings about an understanding that describes the fundamentals of object-oriented programming, since the cardinal relationship between object and class is clear, while their contribution within the program is also emphasised. The dimension of variation in this category is that objects are modelled through the use of classes and the different values of these constitute the focal awareness.

Finally, the last category of description of student understanding of class is the same as the last one identified for object. It focuses on modelling real-world objects and phenomena through the use of classes. This conception illustrates a richer and more complete understanding as it includes the object-oriented view of classes that was presented in the previous category, but it also includes the real-world metaphor. As for Category 3 in the construct of object, the dimension of variation comes from the variety of real-world objects and phenomena that can be modelled through the use of classes and the requirements that programming problems have. Variations in the values of these constitute the focal awareness for the understanding in this category.

6.2.2 Discussion of the Concept of Class

In a formal definition of class given by Conway (Conway, 1999), it is stated that "*A class is a formal specification of the attributes of a particular kind of object and the methods that may be called to access those attributes. In other words, a class is a blueprint for a*

Table 6.4: The focus of different understandings of the concept of class.

	Referential aspect	Internal horizon (foundation)	Focus
1. Class as code	Classes are understood as a piece of code.	Code, syntax, programming language.	On the actual piece of code, the attributes and methods that constitute the class.
2. Class provides structure to the program	Classes are experienced as a construct that provides structure to the program.	Programming construct, programs.	On the class as a construct for providing meaningful structure to a program.
3. Class as a template for objects	Classes are considered to act as templates for creating multiple objects.	Problem specification, object-orientation, object, programs, design.	On the class as a construct for modelling the behaviours and characteristics of an object.
4. Class as a model for real-world phenomena	Classes are experienced as a means for modelling real-world objects and phenomena.	Problem specification, types, design, real-world objects, object-oriented modelling.	On modelling real-world objects and phenomena.

given object". The understanding presented by this definition can be partially found in the categories of description identified among the population of the study. However, the full meaning of the definition is only fully captured in Categories 3 and 4, which represent both a richer conception and the desirable level of understanding that students should achieve by the end of the course.

As was mentioned in Section 6.1.2, (Eckerdal, 2006) has researched Aquatic Engineering students' understanding of the concept of class. The phenomenographic analysis of her study yielded the following three categories of description:

1. Class is experienced as an entity in the program, contributing to the structure of the code.

2. Class is experienced as a description of properties and behaviour of the object.

3. Class is experienced as a description of properties and behaviour of the object, as a model of some real-world phenomenon.

Comparing these categories with the ones found in our study, we can again observe many similarities along with a few differences. The first and second categories identified among the Computer Science students of this study are captured by Eckerdal's first category of description. While Categories 2 and 3 from Eckerdal's study differ only in wording from Category 3 *(class is a template for objects)* and Category 4 *(class as a model for real-world phenomena)* in our study. Once again, this shows that the investigation of the same phenomenon in different settings yields similar results. Even though some of the categories are somewhat different, the spectrum of the experience for the constructs is encompassed in both sets of results. The differences that are observed is something that should be expected, given the variance in the course setting and the researchers that carried out the study.

6.3 Understanding of Attribute

Attributes are the principal elements of a class and its corresponding objects. Attributes define the characteristics and properties of a class by outlining its composition. A common

problem that was observed within the first year students of this study related to the choice of aspects that should be represented as attributes in a class. Many students, who were strongly influenced by prior experience with procedural programming languages, tended to create attributes that did not represent characteristics of the object but that were merely variables, such as counters, that could be used by the main program. Like objects and classes, students' understanding of attributes was discussed in the first and last interview session and also during the second session when students were solving exercises out loud. Students did not appear to fully grasp the purpose of an attribute in the first interview session, however, in the later interviews they were well able to express their understanding of this construct. The questions that were used to investigate the theme were: *"What is the role of attributes in a class?"*; *"How do you choose the attributes of a class?"*; *"What do you think attributes are?"*. The data yielded three distinct categories of students' understanding of the construct of attribute as presented in Table 6.5.

Table 6.5: Categories of description for *understanding of attributes.*

Category Label	**Category Description**
1. An attribute is a container	Attributes are understood merely as containers of specific type that hold values.
2. Attributes determine the functionality	As above, but also attributes determine the functionality of the class.
3. An attribute is a characteristic of the object	As above, and that attributes represent the characteristics of the object that are important in the design of the class and the problem.

The categories were inclusive and the development of the students' understanding from one to another is evident. The categories are analysed in further detail in the following subsections.

Category 1: An attribute is a container

In this category, students experience attributes as containers that hold values. The focus is on the type of these variables, as Liam explains in his fourth interview:

Interviewer: What is the role of the attributes in a class?

Liam$_4$: To hold values, depends what you want to do with them.

Interviewer: Can you elaborate a bit more?

Liam$_4$: Yeah, well any kind you can define your own types of values and at the end of the day it is all broken down to numbers and characters that are numbers as well and essentially you combine those to make more complicated things.

Liam here does not make a distinction between attributes and common variables, rather he emphasises the low-level characteristics of characters as primitive data types. Neil following in the same line of thought describes attributes as follows:

Neil$_4$: The object needs to have certain information, in order to store certain information we have the attributes that have certain types and so on. You can have as many as you need to use in the object.

In this category the conception of attributes has not been differentiated from that of variables, which is not incorrect since attributes are variables, but their purpose and properties serve a specific purpose for the class and the object. The understanding of attributes in this category is not complete as the students' focus is primarily on the practical aspects of the concept, and does not consider attributes as part of the model of the class.

Category 2: Attributes determine the functionality

In this category attributes are seen as determinant factors of the functionality of the class. Most textbooks (Cahill, 2001; Deitel and Deitel, 2005) associate methods with the functionality of the class. However, here students perceive attributes as the instigators of the functionality and behaviour of a class. Eamonn expresses this very clearly in the following excerpt:

Interviewer: So what is the role of attributes in a class?

Eamonn$_4$: They define how a class would respond...

Interviewer: Wouldn't that be the methods?

Eamonn$_4$: Well yeah if the attributes are not there in the first place then you use the attributes to invoke everything and that kind of defines what you can do.

Here, Eamonn expresses an understanding in which the relationship between attributes and methods is very close, since the former influences the latter. This will become clearer when we investigate students' perceptions of methods in Section 6.4. In Category 2, students experience *"methods as something that manipulates the attributes of a class"*. Brian, after a long discussion about an assignment, says the following about attributes:

> **Interviewer:** Are attributes specific to the class you are trying to create or are they just general?
>
> **Brian$_4$:** It depends on what you are trying to do with the class itself. Like with the class Chair you can have colour, material, whatever. If you have the method *where is the chair?* That would mean that in your class attributes you should have (an attribute for) location or something... like I am still going back to the methods because if you need it, you use it, if you don't need it you don't use it...
>
> **Interviewer:** So you say that your attributes sort of derive from the methods and functionality you want to have in this class?
>
> **Brian$_4$:** Yeah that's it!

Brian explains his understanding with an example saying that depending on the functionality one wants to include in a class, one should provide the attributes that would allow for that to be developed. This is a distinct understanding that incorporates the elements presented in Category 1, but it captures a more elaborate and broader view of the theme that involves both the methods and the problem definition in its foundations.

Category 3: Attribute as a characteristic of object

This category subsumes the understanding of the previous two categories, however a new angle in the perception is introduced. The attributes of a class are experienced as containers that hold values for the different objects and are influenced by the functionality the class requires. However, they are also seen as characteristics of the object that the class is describing, drawing on the physical properties of the type in the way that it is represented in the real world. Stephan says this in the following:

> **Interviewer:** What is the role of attributes in a class?
>
> **Stephan$_4$:** They are just describing the characteristics of an object.

Tim, responding to the same question, is a bit more detailed in his response.

> **Tim_4:** Hmm... they define what the class is, the values that the class can get.
>
> **Interviewer:** In your assignment how did you decide that your class `Card` would have those two attributes `suit` and `value`?
>
> **Tim_4:** Because they actually describe the actual card so every card has these two attributes... values of whatever.
>
> **Interviewer:** So the attributes describe the object then?
>
> **Tim_4:** Yeah.

Patrick illustrates an understanding which incorporates that presented in Category 1 but makes a further distinction between how things are viewed from the code and the conceptual point of view:

> **Patrick_4:** I suppose they are used to describe the object, essentially they are containers from an object point of view they are the objects characteristics I suppose.

Finally, Mark explains how the attributes can be derived from the actual characteristics of the object when it is transformed into an entity in the code.

> **Mark_4:** Well it depends on what you are trying to describe, you usually choose your attributes based on physical properties or, for a person, it would be name and stuff, so general characteristics I suppose.

The interview excerpts included in this section show an understanding in which attributes are experienced both in relation to the problem and the code, but also as characteristics of actual objects in the real world. This understanding illustrates a more mature view which contains aspects of the technical perspective of the construct while also encapsulating the metaphor of the attributes' purpose.

6.3.1 Structure and Meaning of Students' Understanding of Attributes

The structure of the categories and the meaning of the students' understanding of attributes is presented in Table 6.6. The referential aspect of the categories that represent the direct object of learning, in this case what attributes are understood to be, shows a gradual shift

from the programming language (Category 1) to a wider perspective that involves the characteristics of real-world objects (Category 3).

Thus, when attributes are experienced as containers, i.e. as simple variables, the emphasis is on the type that an attribute may have and the way this is expressed through the programming language. Therefore, this first category illustrates an understanding of attributes that is completely detached from the actual purpose of the construct in relation to the class or the object. This is not to say that attributes are not "containers", or variables, but there is a major difference between the purpose of attributes and "regular variables" that may be used as counters in a loop for instance. Here there is no differentiation between the two, and therefore the students' understanding is incomplete. The variation, or even the lack of variation that encourages this understanding comes from the lack of differentiation between regular variables and class attributes.

In Category 2 a rather unexpected understanding is presented. Although methods are the constructs that define the functionality that a class implements, the students here perceived attributes as being part of the definition of functionality. However this is not incorrect since the attributes that a class should include depend on the purpose and, therefore, on the functionality of the class. In the foundations of this understanding (second column, Table 6.6) the class and object are included because attributes are intrinsic to them. However the emphasis is on the fact that the functionality required by the problem instigates the elements that will be included as attributes. The dimension of variation here is brought about by the programs that the students work on. Thus the variation for this understanding stems from the different attributes the classes require.

Finally, the last category subsumes the understandings of the previous categories and extends it by introducing a new dimension of understanding where an attribute is experienced as a characteristic that the object has in the real world. In the literature, attributes in object-oriented environments are also referred to as object attributes, illustrating the direct association of an attribute with the respective object (Conway, 1999). The understanding of attributes here is more complete, and "richer", because it approaches the issue of deciding an appropriate attributes for a class from a different perspective: the class as a type, the real-world object and the problem requirements. The dimensions of variation for

Table 6.6: The focus of different understandings of the concept of attribute

	Referential aspect	Internal horizon (foundation)	Focus
1. Containers	Attributes are understood merely as containers for storing values.	Code, syntax, containers (variables).	On the code and variables.
2. Functionality	Attributes are experienced as instigators for the functionality of a class.	Class, object, functionality.	On the functionality that the class should implement.
3. Characteristics	Attributes are understood as characteristics of the object that the class is modelling.	Class, real-life objects, problem specification.	On the characteristics of the real objects and the problem specification.

this complex conception are brought about by differentiation in the problems and classes, but also by the fact that attributes are presented in direct relation to characteristics of real-world objects.

All three conceptions together, as experienced by the students, present a complete picture of the desired understanding one should achieve during a programming course about the construct of attribute. However, when a student's understanding is restricted to the first two categories the chances of developing misconceptions, or of generating programming mistakes in the design phase, are much greater. As was mentioned previously, many students had problems when deciding what to include in the attributes of a class. As a result many programming solutions included counters and other unrelated variables as attributes of the class because these would be needed later in the main program or for some method. Therefore, while the first two categories of description do not represent a misunderstanding of the construct, such limited understanding may lead to problems. As the third category subsumes the conceptions of the first two categories it is the one that should be encouraged through the use of appropriate examples and even explicitly in the introduction of the construct of attribute during the instruction.

6.4 Understanding What a Method is

Method is the most common construct used to provide the functionality required in an object-oriented program. Every Java application begins with the execution of its main method and this typically creates some objects and then invokes methods on those objects (Cahill, 2001). Thus, methods in the object-oriented paradigm are not merely routines that are used repeatedly in a program as in procedural languages, but rather they encapsulate the functionality and behaviour of an object. When designing a class, the functionality and behaviour of the class should be captured by methods. It is reasonable to say that an understanding of the construct of method is essential in understanding how object-oriented programs work.

The course notes guided students to ask the following when deciding on the methods of a class: *"[...] The next step, in common with the design of any class, is to decide what methods the class should be providing. To put it in another way, what do we want to be*

able to ask instances of the class to do for us?" (Cahill, 2001). However, from observations and discussions from interview session 2 where students were solving exercises out loud, it was evident that many students place functionality inside classes that is often outside of the scope and purpose of the class.

The categories of description that were identified for the understanding of method are presented in Table 6.7. The questions that were asked to uncover the understanding of methods during the interviews were: *"How do you understand methods"*; *"What is the role of methods in a class?"*; and *"How do you decide on which methods to include in class?"*. It should be noted that methods were discussed along with attributes, and so these were mostly follow-up questions from the previous discussion on attributes.

Table 6.7: Categories of description for *understanding of methods*.

Category Label	**Category Description**
1. A method is just code	A method is experienced as a piece code that has a specific structure, although its purpose is not clear in this understanding.
2. A method is used to manipulate objects	Methods are experienced as above, but now the purpose of a method is clear. It aims at manipulating objects through their attributes while providing ways of using objects to achieve something.
3. A method defines the functionality of a class	As above, but also a method is now seen through the perspective of the problem specification and in terms of the functionality it encapsulates. Thus, methods are seen as the problem-solving part of a class that enable the representative object to perform actions and solve the problem.

Like the categories found for attributes, the qualitative ways of experiencing methods here are also inclusive. Observing the categories, the change in focus from Categories 1 to 3 is evident. The richness in the conceptions is further analysed in the sections below.

Category 1: A method is just code

The first category summarises an understanding that is directed towards the programming language and its syntax. When students who share this conception explain what methods are, they tend to include syntactical details, like Kevin in the following excerpt:

> **Interviewer:** What do the methods represent in a class?
>
> **Kevin$_4$:** The methods are basically you have... it's the method name and you'd have the return type to see what it returns and... and you have the methods whatever way you need them and you can basically call these methods from you main class with the method's name.
>
> **Interviewer:** So when you were describing that to me, what was the mental image you had?
>
> **Kevin$_4$:** I was thinking of code not specific code but an outline of how these things... how methods are constructed in general.

Although Kevin was asked what methods represent, his response was more in terms of the syntactical elements that compose a method and how it is used in a program. The focus of his understanding is clearly on the syntactical details and the actual code. This becomes more evident in his answer to the follow-up question; when he says that when he thinks about methods it brings to mind the syntax of the Java code. Tim states this very briefly when he says that:

> **Tim$_4$:** Methods are commands to the object for the class to use.

He continues in a detailed description of the method `ChangeSuit` he had developed for his poker assignment.

Category 2: A method is used to manipulate objects

This was a common way of understanding methods in the population of this study. The understanding of the previous category is subsumed by this one, however the purpose of the method is clearer and is targeted at the manipulation of the corresponding object. This conception is very closely related to the object attributes, since a method is experienced as a medium for accessing the values of the attributes. Karl voices this in the following:

> **Karl_4:** They [methods] provide ways of extracting these values of the attributes from the objects and they basically provide ways of seeing the object for what it is. Because in reality you could see the object `Card` and see all its attributes... the methods provide a sort of way of accessing those attributes.

Patrick thinks along the same lines in his response, but he views methods as "things" that allow one to manipulate and interact with the objects in general and not only for retrieving and managing their attributes.

> **Interviewer:** So what are the methods?
>
> **Patrick_4:** It is just a way... ways of I am not sure I can define it as the notes [laughs].
>
> **Interviewer:** Well I wouldn't want you to do that anyway, I want to see what is your understanding of a method.
>
> **Patrick_4:** It is just that... I know what I am talking about just expressing it properly is a bit problematic [laughs]. Methods are for manipulating objects. I mean there is no point in having objects in a program if you can't do anything with them. So the methods provide you with ways of interacting with the object basically.

Brian's understanding of methods is clearly directed towards object manipulation. He summarises it in the following:

> **Brian_4:** A method is like an action or a series of things you can do to... no its... its kind of hard to think about that like... methods are methods it is something that you use to do something with your object. It is something that is inherent to the object something that you can manipulate it with.

The quotations from the students above illustrate an understanding of methods that is directly associated with the object and its attributes. It presupposes the understanding presented in the previous category, but its foundations are still limited to the object and its attributes. Students' understanding in this category is highly influenced by the `Set` and `Get` methods that are used to set and get the values of the attributes in an object (Cahill, 2001). This understanding, although not fully covering the whole spectrum of an attribute, progresses from the previous category because methods are now seen in relation to objects and serve a well defined purpose in the design of a class.

Category 3: A method defines the functionality of a class

The final qualitative category introduces the problem requirements as another aspect of the students' experience of a method. Students understand methods as a medium for implementing the functionality required by the problem, but this is tied to the object-oriented design. A clear example of this understanding articulated by Alan who illustrates the inclusiveness of the previous categories in his understanding.

> **Alan$_4$:** Methods use the attributes to give the answer that you want it to give you. A method uses the attributes to calculate whatever you want calculated... whatever the exercise wants you to calculate.

Eamonn stresses the point that a method is the place in the program where one implements the "problem-solving" part of the solution but at the same time a method should represent an action that is inherent to that specific class. Thus the functionality that is implemented in a class should be related to the characteristics and purpose of the class itself. This is illustrated in the following excerpt:

> **Eamonn$_4$:** Methods are ways of solving problems they are used to solve the problem, and you can invoke them from new objects.
>
> **Interviewer:** You have me confused here... how would you decide that a method would be in a specific class and not in another?
>
> **Eamonn$_4$:** Because it performs a task for a class to solve a problem for that class. So the methods that are included in a class are related to the class itself and the operations that this class can do...

The understanding presented in the above quotations is qualitatively "richer" than in the previous categories, since they incorporate in their foundation both the object-oriented design concerns and the requirements of the problem at hand. When designing a solution and deciding on where to put the functions that the program should perform, students who experience methods as *"implementations of the functionality"* mainly focus on the inherent behaviour of the class as a type, but they are also aware of the desired output that the program should have.

6.4.1 Structure and Meaning of Students' Understanding of Methods

There are clear similarities between the ways attributes and methods are experienced by the students, both in terms of the foundations and focus and in the ways the understanding is developing from one category to another. Table 6.8 presents the structure and focus of the conceptions on methods.

In Category 1, a method is experienced merely as a piece of code. The concept of a method is described and experienced through its syntactical components, such as the return type, the arguments and so on. The focus of the understanding is on the programming language and the technical expressions that are used in the program to invoke methods on the object. The foundations of this understanding are restricted to the syntax of the programming language ignoring the relationship of the construct to the class and the object. Since the students develop a variety of methods for any given program, the variation in the awareness of this understanding comes from the particular differences and similarities these may have. The changes in the return types and arguments in different methods constitute the focal awareness of this understanding.

In Category 2 the experience broadens so that its foundations now include the object for which the methods are designed. In this category, methods are seen as part of the object while they act also as media for accessing and using objects. There is greater emphasis here on the object and, more specifically, on its attributes when methods are used to manipulate it. The concept here is placed in a richer context since methods are now seen in relation to, and as part of, the objects. The variation in awareness is brought about by object/attribute interactions through the use of methods. The difference within these methods constitutes the focal awareness.

In the final category, a method is experienced as the construct that provides the functionality that a specific class requires. Therefore, a method is understood both in relation to the functionality that is engendered by the problem and also to the relevance that it has with respect to the class as a type and its real-world behaviour. The richness and completeness of the conception demonstrated by this category presupposes a relatively mature understanding of object-orientation. Among the foundations of this experience is problem-solving: methods are now seen as elements where the problem-solving part of the

Table 6.8: The focus of different understandings of the concept of method.

	Referential aspect	Internal horizon (foundation)	Focus
1. A method is just code	A method is experienced through its syntax.	Code, syntax, programming language.	On the actual piece of code and on the commands that are used to invoke methods though objects.
2. A method is used to manipulate objects	A method is experienced as a medium for manipulating the attributes of the object.	Code, attributes, objects.	On the interaction of objects and their attributes.
3. A method defines the functionality of a class	A method is experienced as a construct where one can implement the functionality that is inherent to the class.	Problem requirements, problem-solving, object orientation, design, class/object.	On the both the functionality and the design of the class.

program is implemented. The dimension of variation is brought about by the problems and the functionality of the classes that the students are called to implement. The variations in the program's design and functionality constitute the focal point of the awareness in this understanding.

6.4.2 Functions and Methods

In the object-oriented paradigm methods implement commands that instances of the class are capable of carrying out. As such, methods have been introduced to the students of this course as constructs that model the behaviour exhibited by the instances of a class (Cahill, 2001). In the final interview of the series, students were asked to voice their understanding of a method. The students were guided to discuss their ideas within the context of object-orientation and the course that they were currently following. Analysis yielded a set of categories that capture the understanding of a method within such a setting.

When looking at different programming paradigms, such as functional programming, the experience of the equivalent theme to a method, which would be a function, is quite diverse. Although the constructs of method and function are somewhat synonymous[1] in object-oriented programming, this is not the case in functional programming. In the latter a function takes the form of a mathematical equation since the paradigm and associated languages provide very few built-in programming constructs when compared to the high level abstraction mechanisms that Java and other object-oriented programming languages provide. One of the first phenomenographic studies on programming was conducted by Booth in Sweden in 1992. One of the themes investigated in this study was the construct of a function in Standard ML, a functional programming language. Students' conceptions of the construct of function were summarised in three distinct categories of description (Booth, 1992):

- The function as a *static* relationship, in which a function involves a rule. The focus is not on the rule itself, but rather on the expression of the rule or on the effect of the rule. The rule is a black box.

[1]In the literature we find that "[m]ethods that return a value are often called *functions* and methods that return no value are often called *procedures*" (Cahill, 2001).

- The function as a *dynamic* relationship, in which a rule is again involved but the rule itself is the focus. Now the black box has been opened and it is being examined.

- The function as a *dual static/dynamic* relationship, in which a rule is involved, that can be viewed as dynamic or static.

The primary focus of the conceptions of function found in Booth's work is on the rule that a function needs to carry out. The categories progress from a static view to a more complicated one that literally combines the dynamic and static conception. Although functions in Booths' study and methods in this study constitute different phenomena, the progression from one category to another presents a number of similarities. In our first category, *"a method is experienced as a piece of code"*, the experience is a static one as the focus is on the syntax itself. Similarly in Booth's first category the focus is on the expression of the rule rather than on the rule itself. In the second category a method is perceived as something manipulating and interacting with the attributes of the object. Thus the focus is on the effects of the method, similar to the function in the second category of the Booth's study. Finally, the third category in both studies encapsulates a complete understanding where all prior categories are subsumed. Therefore, even if the two constructs are perceived differently within object-oriented and functional programming, the foundations of the categories follow the same development trend.

6.5 Students' Understanding of a Constructor

The concept of a constructor is of a more technical nature than the object-oriented concepts that have been discussed up to this point. However, it is an important element in the construction of a class and in the object paradigm in general. A constructor is generally defined as a "special" method that is used only to initialise the instance variables of a new object (Cahill, 2001). Theoretically, constructors are not difficult for the students to understand and implement. However it was observed in the classroom that many students are not fully aware of the constructor's purpose and, as such, they just include it because *"it's something that has to be there"*.

Students' understandings of constructors were discussed in the second interview session

where they were prompted to create a fully functional class. The questions that were asked during the interviews were: *"What is a constructor?"*; *"Where and when do you use it?"*; and *"What is its purpose?"*. Analysis yielded three qualitatively different ways that students understand constructors, as presented in Table 6.9.

Table 6.9: Categories of description for *understanding of constructors.*

Category Label	**Category Description**
1. A constructor is used for creating new objects.	A constructor is experienced as "something" that creates objects.
2. A constructor is used to initialise object attributes.	As above, but now the internal functionality of a constructor, which is to initialise the objects' attributes, is also emphasised.
3. A constructor is a specialised method.	As above, but constructors are now seen for what they are: methods that have a specific use and specialised purpose.

The categories for understanding of constructors were found to be inclusive. Thus, in the first one the concept is experienced as "something" that a class has to have and that is used to create objects. Students in this category did not seem to have a very clear understanding of how constructors work and even their understanding of a constructor's purpose is somewhat vague. In the second category, the purpose of a constructor is more concrete; while in the final category the students see that constructors are essentially methods, although used in a specific way.

Category 1: A constructor is used for creating new objects

The first category captures an understanding that is targeted at objects. The students see a constructor as something that is used for creating objects, however they do not necessarily exhibit a knowledge of *what* a constructor is. Brian in his second interview session said.

Interviewer: Do you know what a constructor is?

Brian$_2$: Yes.

Interviewer: Can you explain it to me?

Brian$_2$: Okay for example in the class `Student` that we have here the constructor would make an object of the class Student...

Interviewer: Right, so what is a constructor?

Brian$_2$: It is what makes the objects...

Interviewer: Hmm so where do you use it?

Brian$_2$: It is used in `main` to create objects...

Brian very clearly says that constructors are for creating objects, despite the interviewer's attempts to encourage him to elaborate he seems to be viewing them in this single dimension. Mark and Karl exhibit the same level of understanding when they responded with the following:

Mark$_2$: I don't know how to explain it... Hmm it is the initial method that you need to create an object I suppose...

and

Karl$_2$: It is used in the main program when I suppose you want to reference an object you need to create the object first, so you call the constructor to make the object.

Both excerpts are focused on the use of a constructor that is directly related to the object itself. Although this understanding is not a misconception, it illustrates a partial understanding of the concept. When completing the exercises during the interview sessions students were asked to point out the exact line where the constructor was called. Some students were unable to do so. This illustrates that they were not aware of the actual details and purpose of the constructor. However the same students were able to create objects and the class constructors when asked to do so.

Category 2: Initialising an objects' attributes

A common way of understanding a constructor is that it is used to initialise the attributes of an object and therefore to create a new instance of it. The functionality and internal operations of constructors are apparent in this conception and so it constitutes a more complete understanding than in the previous category. Answering the question "what is a constructor?", Eamonn says:

Eamonn$_2$: It is where you can pass the values for the object. So you have [class] `Student` with name, age and address. To assign values to these variables you use the constructor. So you initialise the object this way.

Interviewer: And when would you use it?

Eamonn$_2$: You use it in `main` when you create the object.

Eamonn uses a previously discussed example about the `Student` class to explain his understanding of the constructor. Declan, in the following, gets a bit confused initially but then states clearly his conception of constructors to be initialising the object attributes:

Interviewer: Could you explain to me what a constructor is?

Declan$_2$: So that is when hmm... when you are creating that object and stating what attributes make that object and what their types are...

Interviewer: So you say a constructor is where you state the object or class attributes?

Declan$_2$: Huh well yeah. It is sort of where you actually define where the attributes that compose the actual object are. So you initialise the object by initialising its values.

Colin exhibits the understanding described in this category in the following:

Colin$_2$: Whenever you create a new object you use that to get all the parameters for the attributes. So you put all your attributes as parameters in the constructors and then when you have your new object with the specific values that you want.

The understanding in this category is focused both on the objects and their attributes, while the purpose of a constructor is concretely stated in all the above excerpts.

Category 3: A constructor is understood as a specialised method

In this final category, a constructor is not merely described as "something" but it is now understood in its essence as a specialised method. Neil says this in the following:

Neil$_2$: I would say it is a method in a class that takes in parameters and assigns them to the variables of the class and creates the new object of that class.

Liam adds to this by saying that you can run methods inside a constructor, demonstrating an advanced understanding:

> **Liam$_2$:** It is a method first of all, and it is used to initialise all the attributes of the object and you can also run some other methods in it as well... and by using it you initialise an object and you instantiate it obviously.

Liam is an experienced programmer and it was expected that he would give a rather detailed and technical explanation of the construct. A final quote comes from Colin. He summarises the conceptions that were presented in the previous categories while clearly stating his understanding of constructors as methods.

> **Colin$_2$:** Yeah it is something that hmm. It is just an attribute sorry a method to assign values to the attributes when a new object is created of the specific class.

6.5.1 Structure and Meaning of Students' Understanding of Constructor

As the theme of constructors is more technical than the others discussed in this chapter, the progression of awareness between the categories is different to that of objects and classes. As can be observed in the referential aspect of the categories, (see the first column in Table 6.10), the awareness changes not so much in *"what is understood"*, but rather in *"how much is understood"*. Although none of the categories capture any misunderstandings in the students' conceptions, it is apparent that the first two categories illustrate a partial perception of the constructor relative to that expressed in the third category.

Looking at the categories individually, in Category 1 the main focus is on the outcome that the use of a constructor has. Students here describe the constructor as "something" that creates an object. This exhibits a poor understanding of the internal operations of the construct. The variation in awareness that encourages this understanding is brought about by the presentation of the constructor as a necessary element for creating an object. Thus in the construction of classes, the repeated declaration of attributes that is followed by the creation of the constructor constitutes the dimension of variation in this category.

In Category 2, a constructor is experienced as the medium for initialising the attributes of the class when creating an object. Thus, at the foundations of this understanding is the class; since constructors are now seen to be actively related to the class and its attributes.

The understanding in this category is qualitatively different and richer when compared to the previous one because now the *how* aspect is also emphasised alongside the purpose of the construct. The variation in awareness in this category is brought out by the relation of the attributes and the respective object, while different examples of these constitute the dimension of variation in this awareness.

Table 6.10: The focus of the understandings of the concept of constructor.

	Referential aspect	**Internal horizon (foundation)**	**Focus**
1. A constructor is used for creating new objects	A constructor is experienced in relation to object creation.	Objects.	On the outcome (object).
2. A constructor is used to initialise object attributes	A constructor is experienced as a means of initialising the objects' attributes.	Class, object, attributes.	On the internal operations (attributes).
3. A constructor is a specialised method	A constructor is experienced in its essence as a specialised method that is used for instantiating object attributes.	Class, object, attributes, method.	On the essence of the concept (method).

In the final category, a constructor is experienced for both its purpose and how it achieves it, but now the conception broadens to include what it is in its essence. The awareness has progressed so that students can now abstract and see that a constructor is fundamentally a method that is used in a different way from other methods. Students

that share this understanding perceive constructors outside their strict boundaries of attribute initialisation and become more creative with it by invoking other methods within it. This category illustrates a view of constructors that is more complete and mature. The dimension of variation here is brought out by relating constructors to other methods. The more elaborate and complicated the constructor is, the more attention it will require to understand it; resulting in students paying attention to it and understanding it as a specialised method.

6.6 Shifts in Understanding of Object-Oriented Components

As mentioned throughout the analysis of the object-oriented concepts presented above; all the themes, except for constructor, were discussed with the students during two separate interview sessions, namely session 1 and 4 (see Appendix B). As expected, a number of shifts were observed in students' understanding of these constructs. As shown in Figure 6.1, the majority of the shifts in their conceptions occurred in relation to objects and classes, while only limited instances of shifts were identified in attributes and methods. This can be ascribed to the relative complexity of the constructs of objects and classes, when compared to the more technical themes of methods and attributes.

The categories of description for the individual concepts represent the conceptions of the sample population of the study, but also show the division in the *continuum of understanding* of the phenomena under investigation. The fact that the categories are set in hierarchies does not imply that some categories are "better" than others, in the sense that the existence of an understanding guarantees a "better" learning outcome. Taking the stance of (Bruce et al., 2004) *"[...] successful students may adopt any of the different learning approaches associated with each category at different stages of their study. It would appear that the problems are likely to occur when students don't move beyond the learning experience of Categories 1 or 2"*; we agree that the students go through different stages when learning and becoming aware of a concept. Therefore, students are expected to progress and shift between the categories, since that is what constitutes learning.

In Figure 6.1 there are cases where the shift in the students awareness has occurred "backwards" to qualitatively lower understanding. This was not an unexpected event,

since the categories are inclusive in most cases. Therefore, when the students exhibited an understanding captured by a more developed category, they were aware of the conceptions in the former categories, but the emphasis of the replies in the later interviews had a narrower focus. Based on the previous argument by Bruce et al. the desired outcome is for the students to go through the different learning approaches of each category during their studies, therefore any shift, either forward (to a "richer" category) or backwards (to a "poorer" understanding) is a positive event, since it illustrates how the awareness of the student develops during the learning process.

Figure 6.1: Shifts in understanding of the object-oriented concepts

Student	Concept	Initial category	Shift	Type of shift
Liam	Objects	Cat. 2: Programming construct	Cat. 4: Real-world phenomena	Intra-contextual
Stephan	Objects	Cat. 3: Entity in a program	Cat. 1: Code	Inter-contextual
Karl	Objects	Cat. 1: Code	Cat. 4: Real-world phenomena	Inter-contextual
Eamonn	Classes	Cat. 4: Real-world phenomena	Cat. 2: Structure to the program	Inter-contextual
Declan	Classes	Cat. 2: Structure to the program	Cat. 4: Real-world phenomenon	Inter-contextual
Anthony	Classes	Cat. 1: Code	Cat. 3: Template for an object	Intra-contextual
Sean	Classes	Cat. 3: Template for an object	Cat. 1: Code	Intra-contextual
Alan	Attributes	Cat. 3: Containers	Cat. 1: Characte-ristics	Inter-contextual
Neil	Attributes	Cat. 3: Characte-ristics	Cat. 1: Containers	Inter-contextual
Neil	Methods	Cat. 3: Functionality	Cat. 2: Manipula-tion of Objects	Inter-contextual

In summary, the range of understanding in the form of the categories of description remains stable, but the student's perceptions do not. As discussed by Pong (1999) in the dynamics of awareness, there are two types of conceptual shifts that can be observed during an interview session. The *inter-contextual* that occur due to the introduction of

a new subject in the discussion, thus when the context changes, and the *intra-contextual* shifts that occur within the same context usually spontaneously or when triggered during a conversation (Pong, 1999). In this study there were two interview sessions where these concepts were discussed, hence there is also an element of development in awareness during the year. The context and experience the students had in the time between interview sessions allowed for the majority of shifts to be inter-contextual. However, a number of intra-contextual shifts were observed during the same interview sessions as well, as in Figure 6.1. To illustrate this further two representative student cases are analysed in detail in the following subsection.

6.6.1 Case Studies on Shifts in Understanding

In the first case study Liam demonstrates an intra-contextual shift in his understanding of objects within the same interview session. Earlier in this chapter (Section 6.1) when Liam was asked what an object was he associated objects with `Integers` in his response, saying that they are all programming constructs that are created to hold values and can be later referenced. His understanding exhibits the clear properties of the second category of description for the concept of objects. However, after that statement the discussion continues with a tutorial example they had completed the day before the interview:

> **Interviewer:** How does this relates to the objects you had in the bank system?
>
> **Liam$_1$:** Well... in the bank system I had `accounts` and `customers`, because these were important objects for the problem, and with the bank you need customers and because you have many of them and because they all have different attributes you make them classes. Then you have to decide what are the appropriate methods and attributes, because the account object should be able to increase and decrease the balance because you can't have these kind of methods in the customer it doesn't make sense... A customer can do other things...

When Liam is asked to elaborate on his response in relation to the bank problem, he talks both about classes and objects and he relates objects and their methods to the behaviours that real-world entities have. He specifically says that you would need another class for "account" because "it doesn't make sense" to have methods in the customer class that would

increase/decrease the balance in the object of the customer, since it is not related to the conceptual view of a real customer. Thus, Liam voiced another facet of his understanding where the object is seen as a representation of a real-world phenomenon that has a specified role in the program. In this case, the shift was triggered by the interviewer asking the student to elaborate on his initial response causing a spontaneous intra-contextual shift in the student's awareness.

Another example of a shift, this time inter-contextual, is exhibited by Eamonn's understanding of classes. When the concept of class was discussed in his first interview, Eamonn was a bit hesitant in his response as to what a class is:

> **Eamonn$_1$:** Yeah well classes, they are a little bit difficult to get your mind around it, [...] you certainly have to make yourself think in terms of objects and classes... which attributes go with each class... it's just something that you don't normally think about.

Later in the discussion he was asked about the differences between an object and a class. At this point he expressed an understanding that is more in tune with the fourth category of description for class, where the focus is on how these real-world phenomena are translated into models in the program.

> **Interviewer:** Okay so what is the difference between an object and a class?
>
> **Eamonn$_1$:** A class would represent all the objects, where as... Say class `person` would represent all people, all the people as they are in the world, so they would have the basic attributes in that class, all people generally have hair colour, eye colour and stuff. An object like describes an individual. Like an individual person or an individual chair I suppose instead of the general picture.

Eamonn exhibits a more abstract understanding in the above since he illustrates, with the example of a `Person` class, how people in the real world can be modelled through the use of a class in a program. His fourth interview took place at the end of the academic year and in it the focus of his awareness has changed. As presented in Section 6.2, he viewed a class as the means to achieve structure in the program and as a way to help solve the problem at hand. However, his previous conception has not been eliminated since he states that *"[a class] should be something coherent as well, like a class should represent*

something and that would hopefully give you the means to solve the problem". Thus, the understanding of classes as real-world phenomena is still present, although it has receded into the background bringing the focus to the problem-solving activity where a class is experienced as a construct that provides structure. The shift in the understanding in this case is inter-contextual since the context of the students' experience has changed.

6.7 Summary

In this chapter we have investigated how first year Computer Science students experience five programming constructs that lie in the heart of the object-oriented paradigm. Some of the constructs, namely object and class, are doubtlessly more important in the context of object-oriented programming than some of the other constructs that have been discussed such as constructors. However, the mapping of students' understanding of all these constructs formulates a complete picture of students' experience and understanding of object-orientation.

Holmboe (Holmboe, 1999) outlines a cognitive framework for different types of knowledge in object-orientation. In Table 6.11 we can see how the categories of description for the five constructs that were analysed in this chapter are associated with Holmboes' framework. The four levels, or types, of knowledge described by Holmboe:

- **Initial Hunch:** An initial hunch is described as folk wisdom. Students of this type lean towards operational understanding by focusing on the process of programming.
- **Practical Knowledge - User-base:** The concept of object-orientation is perceived in terms of the processes and operations performed by the programmer. There is still a strong focus on practical applications and how things work. The common factor for all practical knowledge is that it is based on skills with little understanding of why it works.
- **Theoretical Knowledge - Definitions:** This type could be described as a person with an initial hunch who does not obtain personal experience but his understanding is based on theoretical definitions and the formal aspects of object-orientation.

- **Holistic Understanding - Relational:** A person with holistic knowledge relates the implementation and design of a computer program to the real world being simulated.

Although Holmboe talks about cognitive types of knowledge from a constructivist perspective, it is evident that the types that he identified relate to the progression that appears in the categories of description of the object-oriented constructs that we have analysed in this chapter.

Starting with the construct of object, we argue that the understanding encapsulated in Category 1 "object as code" is an *initial hunch* type of knowledge as the conception in this category leans towards the operational processes. The first subcategory of the second category, where objects are experienced as constructs used to hold values, is classified as *practical knowledge*, since the focus is on the practical implications of how the construct works. The second subcategory is a *theoretical type of knowledge* in that the view of object as an active entity denotes a familiarity with the formal aspect of object-orientation. The final category for the concept of object clearly demonstrates *an holistic understanding* since the perception links the formal programming construct to the subjective content of reality.

The categories for the constructs of class and object showed many similarities in their analysis (see Sections 6.1.1 and 6.2.1). In particular, Categories 1 and 4 for class were almost identical to Categories 1 and 3 for object. Therefore, Categories 1 and 3 for object, and Categories 1 and 4 for class can be classified as *initial hunch* and *holistic understanding* respectively in Holmboes' framework (Holmboe, 1999). Category 2, where a class is experienced as a construct that provides structure to the program falls under the *theoretical type of knowledge* although it involves elements from the *practical type of knowledge* of the framework. The same holds true for Category 3 as the conception has a strong theoretical influence.

For the constructs of attribute and method, Category 1 in both outcome spaces presents a *practical type of knowledge* rather than an initial hunch. This is because in both cases the understanding stems from the processes and operations that the constructs can perform. Category 2 for the two constructs demonstrates *theoretical knowledge* as the understanding appears more abstract. The third category for both method and attribute pertain to

emulation of real objects' characteristics and behaviour. Thus they are characterised as an holistic type of understanding.

The final construct analysed in this chapter is constructor. The first category of description for constructors appears to constitute more of an *initial hunch*; students in this category were unable to discuss many of the operational details of the construct. The second category, where the constructor was viewed as something that is used to initialise the attributes of the object, demonstrates a *practical type of knowledge* because the focus was on the operational details of how the construct worked internally. The final category of description for constructors encapsulates both a practical type of knowledge but it also shows an advanced *theoretical understanding* of what a constructor actually does.

Table 6.11: Object-oriented framework of knowledge and its relationship to the object-oriented constructs.

	Initial Hunch	**Practical Knowledge**	**Theoretical knowledge**	**Holistic Understanding**
Object	Category 1: object as code.	Category 2,a: object as a programming construct that holds values.	Category 2,b: object as an active entity in the program.	Category 3: object seen as a model of real-world phenomena.
Class	Category1: class as code.		Category 2: class provides structure to the program. Category 3: class is a template for objects.	Category 4: class is a model of real-world phenomena.
Attribute		Category 1: an attribute is a container.	Category 2: attributes determine the functionality of a class.	Category 3: an attribute is a characteristic of the object.
Method		Category 1: a method is just code.	Category 2: a method is used to manipulate objects.	Category 3: a method defines the functionality of a class.
Constructor	Category 1: a constructor is used for creating new objects.	Category 2: a constructor is used to initialise object attributes.	Category 3: a constructor is a specialised method.	

Chapter 7

General Programming Components

In this chapter we extend our analysis to four concepts which are central to both object-oriented programming and to programming in general. These concepts are: *algorithms*, *arrays*, *iterations* and *selection*. Since these are neither specific to Java nor strictly to object-oriented programming we refer to them as *general programming components* in order to distinguish them from the *theoretical* and *object-oriented components* which have been analysed in the previous two chapters. The four general programing constructs that we investigate in this chapter were selected both because of their importance according to the relevant literature (see for example (Gries, 2002; Howe et al., 2004; Walker, 1998; Koffman and Wolz, 2001)), and also because our pilot study showed that educators think that some students find them challenging.

The construction of algorithms lies at the heart of programming; indeed it is fundamental to the process of problem-solving. It encourages a more abstract way of thinking that we have previously identified as critical to the development of *programming thinking* (see chapter 5). Most students who enter a programming course have a clear conception of what an algorithm is, but they have not necessarily linked this to the concept of computer programming. For example, for some, an algorithm is a more general, and even elusive, construct because when students solve a programming problem they are not necessarily aware that they are constructing an algorithm. Similarly an educator may be teaching algorithm construction without specifically mentioning algorithms, but rather by illustrating different ways to solve a problem and to reach a solution.

On the other hand, arrays, iterations and selection are some of the more coherent constructs that are taught in most introductory programming courses. Students are not required to have developed a prior understanding of these, and the educator teaches them deliberately and explicitly through examples and problem-solving.

It is imperative that an insight into how students understand these constructs is obtained since they are fundamental to programming and programming thinking. As Dijkstra pointed out about programming constructs (Dijkstra, 1982): *"The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities."* Therefore, looking at how these are understood by novices would provide educators with a powerful teaching guide. Unfortunately, most of the literature that discusses the programming constructs under consideration here concentrates on the ways experts believe these should be understood and on the best ways of teaching them in introductory and advanced courses. In contrast, this study investigates how these general programming constructs are understood by students taking a typical introduction to programming course. It is argued that no meaningful change in the way programming constructs are taught may be made without first comprehending the students' experience (Berglund, 2004).

In the remainder of this chapter the concepts of algorithms, arrays, iterations (loops) and selection are analysed through the phenomenographic perspective within the Java paradigm on an introductory programming course. The critical aspects of the derived outcome spaces for these constructs are then discussed; while the relevance of the results with respect to the literature is also addressed.

7.1 On the Understanding of Algorithms

The notion of algorithm is central to the programming paradigm, since it encapsulates the most basic way of thinking required to develop a program. One of the most essential things that students have to realise when learning to program is that a computer cannot understand and interpret complex commands as a human can. Computers can only interpret a limited set of simple commands that should be presented in a meaningful sequence and are generic enough to work for any case scenario. One of the most common textbooks used for introductory Java courses gives the following definition for algorithms (Deitel and

Deitel, 2005): "Any computer problem can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of

1. the *actions* to execute and

2. the *order* in which these actions execute

is called an *algorithm*."

An algorithm is a general concept that can be meaningful not only in the programming paradigm but also in real-life scenarios. Therefore, most students have a prior understanding for what an algorithm is. However in this study they were asked to discuss their understanding of algorithms in the context of computer programming.

Algorithms were introduced in the very first weeks of the programming course considered in this study. Students were initially asked to reflect on the complexity of instructions that a computer can understand relative to those understood by a human being. In an active, physical example they were asked to instruct one of their classmates to reach the door from where he was standing given a finite set of instructions. This exercise illustrated the need for clearly sequenced instructions; while the fact that their initial solution would work only from a specific starting point, introduced the need for conditional commands and repetition of some parts of the algorithm. For almost three weeks the students were given a series of exercises in which they had to develop algorithms. Initially they produced instructions in English but they gradually progressed to the Java syntax. Students' understanding of an algorithm was discussed in the first interview session. This was close to when they had been formally introduced in lectures, but after one week of practical classes on the topic.

It is important to note that when discussing their understanding of what *learning to program* means to them, many students paralleled it with learning to write algorithms. A comprehensive example of that notion comes from Anthony in his first interview where he said *"Programming is writing an algorithm for solving a problem. Programming is getting the computer to understand what this algorithm is going to do."* According to Anthony learning to write algorithms is equivalent to learning to program. This illustrates the importance that the students' understanding of algorithms holds in this study. As

Table 7.1: Categories of description for the *understanding of algorithms*.

Category Label	Category Description
1. Algorithms as problem-solving	Algorithms are seen as methods that include a series of commands designed to solve the problem at hand.
2. Algorithms as a means of reducing complexity in programs	As above, but now algorithms are seen as a feature that reduces the complexity inherent to the problem specification.
3. Algorithms outside the programming paradigm	Algorithms are seen as a procedure that represents a step-by-step solution that can be applied to both the physical world and any programming paradigm.

was mentioned previously, it is perceived that algorithms encapsulate the *different way of thinking* that is necessary when learning to program.

The students were asked to answer the following questions: *"What is an algorithm?"*, *"How would you define it?"* and *"Why do you think it is important when learning to program?"*. The three conceptions found within the student cohort are summarised in Table 7.1. These will be elaborated on, and discussed further, in the rest of this section.

Category 1: Algorithms seen as problem-solving components

In the first category, algorithms are understood to be the equivalent of methods. In many cases algorithms are referred to as functions, but more often students tend to describe them as a series of commands designed to solve the problem at hand. This conception is the most pragmatic one since it is grounded in the programming paradigm, and involves the tangible elements of algorithms. For example, Patrick says *"I guess it is a series of commands that solves a problem."* This conception of algorithms is very close to Categories 2 and 3 on the nature of methods in Section 6.4.

On the other hand, Anthony highlights the *generic* characteristic of an algorithm, which is to solve the problem for any possible case.

> **Anthony$_1$:** An algorithm is a set of instructions that would solve a problem but it would work for any given situation. The instructions are specific to the problem you

> are trying to solve, but it should give you the right answer for whatever input you give it.

Ken shares this understanding when he says the following:

> **Ken$_1$:** An algorithm is a collection of commands I guess. It is supposed to be a bullet proof way to solve a problem. Obviously, the commands that are available are limited so you have to work with that, it's like a function really...

The next two excerpts from Declan and Mark emphasise that algorithms are meant to be a procedure that can be carried out by a computer.

> **Interviewer:** How would you define what an algorithm is?
>
> **Declan$_1$:** As a set of instructions, to explain to the computer how are you going to solve a problem.

and when Mark was asked to describe what an algorithm is, his response was:

> **Mark$_1$:** It is a series of instructions to tell the computer what ... to do something.

By observing the individual responses of the students, it can be seen that while different, they share points of similarity. All the students that share this conception began describing their understanding of an algorithm by saying that it is "a set of instructions." This shows that their understanding of algorithms is grounded in the pragmatic elements of an algorithm which is its components, the sequence of instructions. At the same time each student emphasised another property of an algorithm which is its *result.* Ultimately, if one needs to devise an algorithm, there is an underlying problem that requires a solution, and the students incorporate this into their understanding. Hence students view algorithms in terms of their result, which is to solve the problem. Although implicitly all the students were talking about algorithms that a computer can understand, only Declan and Mark made that explicit. However their responses express an understanding of algorithms that has its basis in the problem-solving aspect of algorithms.

Category 2: Algorithms as a means of reducing complexity in programs

In this second category algorithms are identified as the means by which a reduction of complexity is achieved in a program. Unlike the previous category an algorithm is not viewed as something that ultimately solves a problem by itself, but rather it is perceived as something that can be incorporated into a program or a method to reduce the complexity of the program. Alan explains this in the following:

> **Alan$_1$:** An algorithm is a list of commands, very basic, to form some complex function. Yeah it is just a series of commands and you have to make it like with many small steps and simple ones to make it do something complex. It has a structure and usually contains loop with a while command and you figure out if you need counters and so on. [...] There is a logical structure and you make it do whatever it is what you have to do, but not all at once you can have many that are connected to do that.

Cormac also takes up this aspect of a logical sequence as a property of algorithms in his response:

> **Cormac$_1$:** It is a list of instructions but a logical list of instructions. It has to be in a certain order the set of commands controlling what is going on and saying what is going to happen. You have to be able to differentiate between the different situations.
>
> **Interviewer:** So you say it is a sequence of logical steps that has conditions and commands with a beginning and an end.
>
> **Cormac$_1$:** Yeah.

Both students mention the use of logical sequencing as a means of reducing complexity and achieving the end result, which is to solve a problem through the use of algorithms in a program. They both talk about possible components of an algorithm such as conditions and iterations, although an algorithm is not viewed as an isolated procedure that provides a solution but rather as a component within a program.

Other students exhibit a broader understanding of algorithms based on the reasoning that if an algorithm is an implementation of a part of a problem, then every construct that helps in solving the problem and reducing the problem complexity can be perceived as an algorithm. Brian, when asked to identify the most important concepts that were introduced in the course up to that point, says the following:

Brian$_1$: I would probably say just the algorithms... Because then you have methods but methods are algorithms in a sense because it is the basis for problem-solving and the basis for everything. Just the rest of the things, like classes and object they are just programming ways of thinking and structuring things. Like a method is called a method but it is actually doing something like the algorithm does, an implementation. A class is called a class but.. I don't know how you define that and anything but it is terminology more than anything, algorithms are the actual problem-solving part of it. You have the problem and then you break it down and then for each part you create an algorithm. So then you combine all these it is not as complicated as it seemed before.

The supporting excerpts in the analysis of this category clearly demonstrate a view of algorithms as procedures for handling the complexity of a problem. Unlike Category 1, the focus has shifted from the *result* that an algorithm may have to the *purpose* that it has in a program. Thus algorithms are viewed as a means for reducing complexity while at the same time they are perceived to be sufficiently abstract so as to encapsulate the characteristics of other programming constructs like methods and classes.

Category 3: Algorithms seen outside the programming paradigm

The conception in this final category of description was voiced by a small number of students, however it is important to include it since it opens up an angle that was not apparent before and, at the same time, is inclusive of the previous conceptions. Students in this category view algorithms both as a programming construct but also as something that can model a solution outside the programming paradigm. The understanding here is more abstract and somewhat detached from programming. Stephan, an experienced programmer, is able to talk about algorithms in a way that illustrates a deep understanding of the abstract process without loosing track of the concept of algorithm as a procedure that can be applied in other paradigms.

Max$_1$: An algorithm is actually a sequence of actions... well yeah a sequence of actions designed in such a way that you will get something to carry out an action. It doesn't hold only for computers, it is the same for people. Like, I can give an example... the lecturer has shown us how to construct an algorithm to get a person

> from one place in the class to walk to the wall. So it was just a sequence of steps on a smaller scale like very simple to demonstrate basic understanding. Something that when all of these simple commands are combined you can perform a bigger more complex action.

Liam, another student with previous programming experience and general aptitude in programming, in a very laconic statement summarises this understanding in the following:

> **$\mathbf{Liam_1}$:** A series of instruction to do one particular task, either physical or programming.

A final expression of this understanding comes from Neil:

> **$\mathbf{Neil_1}$:** A series of commands that... That's it pretty much... to do something whatever it is required to do. It can be done by both a person or a computer, it is more general.

All the students who voiced an understanding of algorithms that moves beyond the programming paradigm were high achieving students that had programed before in at least one programming language. This is not to say that only students who had previous programming experience can understand the nature of algorithms in this way. However it may be that this experience allowed them to view algorithms in a more abstract way, both within the programming paradigm and in a more general setting. We argue that this understanding is a richer one since it encapsulates the metaphor of a real-world step-by-step solution together with how this can become an algorithm for a computer program.

7.1.1 Identifying the Variations in Understanding of Algorithms

To identify critical aspects of the understanding of algorithms, we need to look at the structural aspect of the categories. In Table 7.1, the first category presents an understanding where algorithms are compared to methods. They are experienced as a single component that solves the entire problem. The focus of this understanding of algorithms, the structural aspect, is that students parallelise and compare problem-solving components and methods. In the foreground of the understanding is the notion of the computer as something that will take an algorithm and use it to solve a problem. This is understood to be the ultimate goal of an algorithm. For students to be able to discern the understanding

described in Category 1, the relationship between the problem and the algorithm needs to be apparent. The different types of algorithms that are developed for solving different types of problems constitute the values among this dimension.

In Category 2, algorithms are experienced as a construct that reduces complexity in a program. Unlike the previous category, an algorithm is now understood in relation to a deconstructed problem from which many algorithms can be developed, which are later combined to solve the problem. The focus of this understanding is on problem deconstruction, while the solution of each component of the problem is experienced as an algorithm. It is only later that these algorithms are combined to create a program. Thus, the focus here is on the purpose of an algorithm. This is to reduce the inherent complexity of the problem, whilst it is the combination of several algorithms which constitutes a solution of the problem. In order for the students to discern this understanding of algorithms, they have to become familiar with the merging of several sub-algorithms to create a complete solution. By experiencing problem deconstruction and program construction through the combination of different algorithmic parts, a student can become aware of algorithms as a means for reducing complexity. The different types of problems and the combination of algorithms to derive a solution constitute the values among this dimension of variation.

In the last category given in Table 7.1, the structural aspect is the algorithm as a set of instructions that uphold certain properties and that can be interpreted by either a computer or a person. In this case the student can see how creating algorithms relates to things that happen in daily life. For example, a student who experiences algorithms in this manner, would view giving instruction to a tourist on how to get to the local post office, as requiring the same type of thought process as creating an algorithm that counts the occurrence of a specific letter in a sentence in a program. Although this level of understanding is expressed in a less technical way, it illustrates a higher level of understanding where the knowledge has matured and can be transferred to other practises. In order to discern such an understanding, a student needs to be aware of the modelling attributes of a computer algorithm that replicate those seen in real-life situations. Table 7.2 provides a summary of the critical aspects that were identified for the understanding of algorithm.

It should be noted that deciding on the hierarchy of the categories of description that

Table 7.2: Aspects of variation in the understanding of algorithms.

Referential Aspect	**Structural Aspect**	**Dimensions of Variation**
1. Algorithms are understood as problem-solving components (methods).	The focus is on algorithms as a set of instructions that solve the programming problem at hand.	Variation of problem specification and the algorithms that can be developed for them.
2. Algorithms seen as a means of reducing complexity in programs.	Focus is on problem decomposition and the development of sub-algorithms that, when combined, constitute a complete program.	Variation is brought about by the various combinations of algorithms in a program and by the varying complexity of the problem.
3. Algorithms seen outside the programming paradigm.	The focus is on the reality aspect of algorithms.	Variation is brought about by the transformation of daily life problems into computer programs.

were identified among the students was not an easy task for this theme. For many people mathematical notions lie at the heart of an algorithm. Therefore it may appear at first inappropriate to place the last category where algorithms are also experienced outside the programming paradigm at the top of the hierarchy. This decision was not taken lightly but rather the data corroborated the hierarchical structure, based on the inclusiveness of understanding discerned in the categories.

7.1.2 Pre-college Students' Understanding of Algorithms

A recent study performed with pre-college students in Israel evaluated their conceptions of what an algorithm is through the solutions they devised to algorithmic problems (Heberman et al., 2005). The research was build on the premises that *"an algorithm is a solution of an algorithmic problem; it must correctly fulfil the pre-condition/post-condition (I/O relationship) specified in the problem; and it should be concise and efficient"* (Heberman et al., 2005). The study was conducted with 97 pre-college students who studied Computer Science following their high school syllabus presented in (Gal-Ezer and Harel, 1999). The methodology used to gather the data was a mixture of quantitative and qualitative techniques, where the students were initially asked to solve a puzzle and then develop an algorithm for it and after that a number of students were interviewed about this process.

The results indicated that the students of the study demonstrated a *cognitive obstacle* (as defined in (Tall, 1989)). Nevertheless, the conceptions that they identified among the population were as follows:

- ***An algorithm is the exclusive solution of an algorithmic problem.*** Students consider the algorithm as a "stand-alone" product of the problem-solving process.
- ***An algorithm "serves as a proof of its own correctness".*** Students relate to an algorithm as a mechanism - a means that proves the correctness of a suggested solution. For example, a student that adopted a proof-by-examples approach said: "the process that is emulated by the algorithm proves and demonstrates its correctness, because I can 'run' it (mentally or on the computer) for a large number of inputs and demonstrate that for each input, the correct output is obtained".

- ***Algorithms embody processes; they are not I/O objects.*** Even when they identify simple I/O relationships, students usually avoid the use of arithmetic expressions/functions for displaying a final declarative result of the problem's analysis.
- ***(too) Short algorithms are not acceptable.*** Students hold the misconception that "a short algorithm is not satisfactory" if it neither represents the story described in the problem, nor reflects the problem's analysis process. Hence, students might not consider short concise algorithms as satisfactory solutions, even though they correctly yield the desired I/O relationship. (Heberman et al., 2005)

Although the above study investigates the students' conceptions of algorithms, this phenomenon differs from that explored in our programming course. The way algorithms were introduced to the high school students was through a formal mathematical definition of the concept. In our study algorithms were introduced in a more abstract way and were primarily presented as (Cahill, 2001) *"A procedure for solving a problem for all possible cases that can be interpreted by a computer."*

Even though data were captured and analysed in the two studies following different philosophies, we can observe a number of similarities and differences among the two sets of results. The conception of pre-college students that "an algorithm is the exclusive solution of an algorithmic problem" corresponds to Category 1 of the outcome space where "algorithms are understood as problem-solving components."

The second conception in Heberman et al. was that algorithms are perceived to serve as proofs of their own correctness. This is seen within Categories 1 and 2 of this study but it is never in focus or explicitly voiced. For example, in Section 7.1, Anthony says *"An algorithm is a set of instructions that would solve a problem but it would work for any given situation. The instructions are specific to the problem you are trying to solve, but it should give you the right answer for whatever input you give it."* Later in the interview he says *"Like you have certain cases that he [the teaching assistant] gives us and you have to make sure it works for all for them, then your algorithm really solves the problem."*

The last two conceptions found in that study: "algorithms embody process" and "too short algorithms are not acceptable", are not in the same context as the phenomenographic outcome space presented in section 7.1, and therefore they could not be compared or

combined further. Nonetheless, it is interesting to note that the two studies, although being so different with regards to their population and even the phenomenon in focus, shared some significant commonalities.

Another study where students' understanding of algorithms was in focus is (Perrenet et al., 2005). This was a quantitative project among mature university students in the Netherlands, which was later cross-validated with qualitative methods in (Perrenet and Kaasenbrood, 2006). Few points of commonality are observed compared to the categories found among this study's population. This may be due to the different analytical methods that were employed in the two projects, since in (Perrenet and Kaasenbrood, 2006) the conceptions were predefined based on the experts' understanding and not on the students' understanding. The most probable explanation for the lack of correlation in the results would be that although the theme was seemingly the same (students' understanding of algorithms), the difference in the manner and depth that it was taught in the two courses constituted an altogether different phenomenon.

7.2 Arrays

An array is a programming construct that exists in most programming languages. It is one of the simplest data structures and is introduced in the majority of the introductory programming courses. An array provides an easy way to store and manipulate multiple objects and primitive data types. Hence, arrays are essential to the solution of certain types of problems. In this course, arrays were introduced in the second week of the second term. Both single-dimension arrays and 2-dimensional arrays were taught to, and used by, the students on this course. They were used in various contexts from data sorting to matrix multiplications. The majority of the exercises that were given to students during the course can be found in (Cahill, 2001, p. 369).

The concept of arrays was discussed during the fourth interview session. The main reason for discussing it at this later stage in the year and not just after it was introduced in the class, was the disparity of understanding among the students. Based on observations of the students during the lectures, it was clear that many students' lack of understanding and knowledge of arrays actually prevented them from using them in their solution. Therefore,

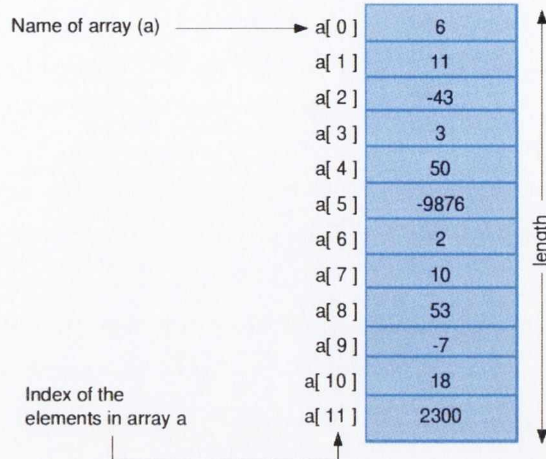
it was decided to allow the students some time to get to grips with arrays in both their theoretical and technical aspects before investigating their understanding. By the time of the fourth interview the students had had the opportunity to work with both single-dimension and 2-dimensional arrays. This allowed for the collection of a very rich set of data. Due to the technical nature of arrays, the students were asked to solve a problem (see Appendix B) to verify that they could apply their knowledge. Students were initially asked to explain their understanding of arrays verbally. Prior to the presentation and analysis of the results obtained, the following subsection presents a short introduction to arrays using the same examples as in the course text.

7.2.1 Arrays in Java

In their introduction to arrays students are presented with the problem of creating a program that would maintain student records for the school. After identifying the types of classes that would be required in a program, such as `Student`, the students were then asked to think of a solution that would manage all these objects. The issue of managing and storing multiple `Student` objects then became apparent. This suggests that such a program might need to use hundreds of different variables to hold *references* to these objects. In the presentation of this problem the educator demonstrated the need for a construct that would be able to hold, and manage, collections of objects in a program. He then continued by saying (Cahill, 2001)*"What we need is a way of representing a collection of entities, like the collection of students, that doesn't require us to know in advance how many entities, will be in the collection, doesn't require us to declare a separate variable for each entity, and provides a convenient way for us to access the individual entities when necessary. One way of managing such a collection in Java is known as an **array**."*

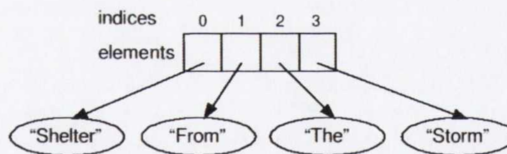
Thus an array is an *indexed* collection of values. In Java an array can contain values of any type, both primitive types such as integers and reference types such as instances of objects. To refer to a particular element in an array, one needs to specify the name of the reference to the array and the index number in the array. Figure 7.1 shows the logical representation of an integer array. The length of the array is 12 and the index of the first element in any array is always 0.

Figure 7.1: Logical Representation of an integer array.



Another example of an array, adapted from (Cahill, 2001), is shown in Figure 7.2. In this case, the array contains values of a reference type (object) String, the array stores *references to the corresponding objects.*

Figure 7.2: Example of an array of objects.



In Java there is no single "array" type. Instead, Java provides the means to introduce new types whose values are arrays of some existing type. These new types are called *array types.* Thus an array is actually a reference to an array object in memory, meaning that individual arrays are objects of the corresponding array type.

The syntax of arrays was then introduced along with various examples on their use. The above provides a short introduction to the theoretical basics underlying arrays in Java and also provides examples of the material used to present this construct to the class.

Table 7.3: Categories of description for *students' understanding of arrays.*

Category Label	Category Description
1. Arrays as containers	Arrays are experienced as "something" that can contain a number of objects, or rather as collections of objects.
2. Arrays as tools for dynamic object access	As above, but now the dynamic access characteristic is the main focus of the conception.
3. Arrays as collections of pointers	As above, but now the focus is on the internal operation of the arrays. This understanding illustrates a "deep" understanding of the memory allocation of pointers and how these are used through the arrays.

7.2.2 Students' Understanding of Arrays

Three ways of understanding the nature of arrays have been articulated by the group of students in this study. They are summarised in Table 7.3. In the most basic conception arrays are not really perceived as a programming construct with properties but rather are viewed as a collection of objects, or "something" that holds objects. This conception has been labelled *arrays as containers* and is similar to the category identified for attributes (Section 6.3), with the difference now being that arrays can hold multiple values or objects.

In the second category arrays are seen as constructs and the emphasis is on the dynamic access that can be achieved through their use. In the final category the focus moves to the internal and memory representation of arrays. This last category expresses a deeper understanding.

Category 1: Arrays as containers

In this category arrays are experienced as collections of objects. The emphasis is on the objects themselves rather than on the array as a programming construct. Eamonn expresses this clearly in his response:

> **Eamonn$_4$:** They [arrays] are basically groups of objects of one particular type.

In the above, Eamonn sees any group of objects as an array and therefore it is recognised as a multiple container. Patrick is a bit more hesitant in his response, however he seems to be reaching the same conclusions as Eamonn.

> **Patrick$_4$:** Array is a ... a way of... it contains multiple types... I am sorry... I am very bad at describing these things. I guess it is a *multiple container* for containing many different types of stuff, objects or primitive types but they should all have the same type like. It can hold one plus things, [...] it is a way of holding one or more than one objects or things but they must all be of the same type of objects.

When the student was later asked when it is required or preferable to use arrays in programming, he responded:

> **Patrick$_4$:** Well I guess it depends, I mean it seemed to be the right thing to do in that poker program because you're going to have... I suppose if you want to like have multiple objects in the same place together like the cards in the poker program!

As in the first category found for the programming construct of *attributes* (Section 6.3) the type and nature of the objects that would be stored in an array form part of Patrick's conception, although the simplicity and superficial nature of his description of the construct illustrates a rather naive understanding. A final example of this category comes from Colin who says:

> **Colin$_4$:** A single array is a set of objects of the same type which are held in its... I suppose you can think of it as a container that has something in there, like many things, and they would all be the same type but they would all have different properties hm... not different properties but different values and the same properties.

Colin's description of an array of objects echos the multiple containers that Patrick described. Many students voiced a similar level of understanding but as their responses did not differ significantly from the ones above they have not been included here. All of the excerpts mainly focus on the properties of the objects that can be stored in an array rather than on arrays themselves. Hence arrays are experienced in this category as mere containers of objects.

Category 2: Arrays as tools for dynamic object access

In the second category arrays are experienced as tools that allow dynamic access to their contents. The understanding in this category has moved beyond that of the contents of an array to a more inclusive view of an array as a construct. Many students who voiced this conception tended to describe arrays as a box or a list, as if they were describing a mental picture they had formulated. Alan illustrates this in a very clear way in the following:

> **Interviewer:** Could you explain to me what do you think is the nature of an array?
>
> **Alan$_4$:** An array is like a box of whatever it is you're doing an array of, for example I would say an array of `ints`. So you would have a box that would be n times long that would be specified by you and each one of those would contain a number `int` inside it, and they could be accessed at anytime as long as you specify which box you want it to go to and if you specify an array of length 7 because of the way java is the first one is 0 and the last one is 6. Because this is the way it is specified within it.
>
> **Interviewer:** So while you were talking about arrays what did you have in your mind?
>
> **Alan$_4$:** Actually the diagram. Well it is just an abstract way of looking at it, to help you understand it. Whereas all that it is really is the code and it has a list of numbers that it is storing it away and this is all that it's doing. So I kind of thought of the diagram and I also thought of the code as well.

The focus has clearly changed when compared to the responses in the previous category. The emphasis here is on the arrays themselves rather than on the items that they contain, however the underlying conception of arrays as a group of objects is still present but has moved into the background. This becomes apparent when the student is asked when he feels it is necessary to use arrays.

> **Alan$_4$:** Arrays are simple because you can create them easily... My card program... my poker program as I said earlier I didn't know about arrays before but if I had I would have used them straight away because my class then would have used two attributes and an array of type `int` and an array of type `char` that would be very fast and it would be easier to sort them into numerical order it would basically make it a lot easier. So I would say any program that has lots and lots of the same type of variable should use arrays.

Although contained within a discussion of his poker program, the understanding of arrays as a grouping of objects is present in the above excerpt. This differs from the previous category as here arrays are experienced as constructs and more specifically as high level tools.

> **Neil$_4$:** They are a group of primitive data types but they can also be objects as well. They basically store all information you need to have, and it is an easy way of storing information because you can access them later easily because they have the indexing bit. It is a very easy way of managing data. It is like the link lists in C++ but easier, much easier! [laughs] [...] Yeah so basically it is a list of object all of the same type and each object had an index and each array has a fixed amount of objects it can hold and they are very handy for doing things.

Neil compares arrays with linked lists in C++, signifying an understanding of an array as a programming construct that provides easy access to its contents. Another visual representation of arrays comes from Cormac who says:

> **Cormac$_4$:** It describes like... it has all the boxes and you store an object in each of the boxes and then you can access those boxes in a way you can call up on one of these boxes, and then you have a set of objects in an array, that you can access whenever you want, and you don't have to go though them one by one you can access which ever you want. It was really helpful when I could visualise it and see how it can be represented.

In all the above excerpts, arrays have been described very vividly, as if describing a picture. The emphasis was on the benefits of using an array which was, according to the students' experience, the ease of access to the stored objects. This differs from Category 1 as an array is experienced here as a programming construct with associated properties and capabilities.

Category 3: Arrays as collections of pointers

In neither of the two previous categories has an array been experienced as a collection of pointers; rather the "boxes" of an array were used to store objects (and not pointers to them). This third category encapsulates a view that demonstrates an advanced understanding of how arrays operate as programming constructs in memory. Liam summarises this in the following:

Liam$_4$: It is a sequence... so your array would be like a pointer to some memory location where there are integers or anything else each successive memory location would be an integer and then you reference them.

[...]

Interviewer: Why do we need them, what do they add in the Java context?

Liam$_4$: You can write more general things you don't need a specific name for each object you don't need to know that specific name for each object you can just reference it with a number and then you have it.

The description given by Liam is again diagrammatic, although his response manifests an advanced understanding of the inner workings of arrays. This is reinforced when he talks about the flexibility of arrays in the context of Java, where he says that it is possible to have objects without specific names that are accessible by reference through the array. This is also expressed by Anthony:

Anthony$_4$: I have a very good understanding of arrays. An array is like a group of pointers that would point to a certain object or it could be an array of primitive types as well, but it points to references of the objects, so you can store any number of objects in there and then you can access them through the indexes that hold these pointers inside.

Mark also articulates this understanding in the following excerpt:

Mark$_4$: It is a list of references to lots of different object, I mean if it is an array of objects, it could be an array of primitive data types then it is a list of those things. It is just a list really... a list of pointers to whatever.

The above excerpts demonstrate an advanced understanding that brings out the actual nature of arrays as data structures that refer to objects rather than physically storing them. The understanding that was presented in the previous two categories is still present, although in a somewhat different framework. There are still objects in an array, although now there are references to these objects and there is still the option for dynamic access to the data in the array, but this is now done through their references.

Table 7.4: The distribution of understanding of the nature of arrays.

Nature of arrays	Number of students
Arrays as containers	5
Arrays as tools for dynamic object access	8
Arrays as collection of pointers	3
Total	16

7.2.3 Critical Aspects of The Students' Understanding of Arrays

Three different conceptions have been found in the understanding of arrays within the population of this study. From the categories it can be seen that the understanding develops and becomes more mature, as the understanding within the individual categories in this theme is inclusive. When considering the understanding that is encapsulated in the first category, one could say that this view of arrays constitutes a misunderstanding since arrays are much more than containers for objects. However, one cannot say that this response is wrong either, as it demonstrates a partial and maybe even naive understanding of the construct. In the first category, the focus is on the contents of the array rather than the construct of the array itself. Arrays here are seen purely in the context of their purpose, which is to hold collections of objects, and no further attention has been given to the construct itself. In order for the students to reach this understanding, they have to experience a variation in the types of arrays they need to create for given problems. A significant number of students shared this understanding. The distribution of the students among the categories can be seen in Table 7.4.

The understanding is expanded in Category 2 where the students are more focused on the benefits of using arrays, which are now experienced as constructs and not merely as collections of objects. The variation in the awareness in this category is brought about by using arrays in situations where dynamic access to the data within the arrays is required. The majority of the students shared this understanding, suggesting that the hands-on experience of using arrays in problem-solving encourages this understanding.

Very few students experience arrays as collections of pointers and those that did understand them this way were the most experienced programmers among the population. This category of understanding has as its focus the internal mechanism of how arrays are repre-

sented in memory. Thus arrays are understood as a continuous block of memory where the values of the objects are stored while the array itself contains pointers to these, as illustrated in Figure 7.2. Since the categories are inclusive, this last category proposes a "richer" understanding of arrays. In order for the students to discern this understanding they have to first understand how the construct of an arrays works in theory, since mere use of the construct will not necessarily bring the students awareness to that aspect of arrays. More experienced programmers, who have worked with other programming languages, such as C++, are already aware of other complex data structures that utilise pointers. Therefore, this understanding comes to them more easily.

7.2.4 Is an Array an Object?

Even though the construct of an array was explicitly presented to the students as an object during lectures, it became apparent through observation that students had not really grasped this idea. Due to the practical nature of the course many students do not attend lectures, but they rather go to the laboratory and tutorial sessions only. Therefore, for many students it proved to be quite challenging to understand that an array is an object of a built-in class in Java. For this reason the students were specifically asked if they thought arrays were objects or not in the last interview session of the study. It appeared that many of the students had not thought about this before and they were only coming to the realisation that an array is an object during the interview, like Brian in the following:

> **Interviewer:** So do you think that an array of integers, let's say, is an object of a class?
>
> **Brian$_4$:** Hmm.. I guess it kind of would be actually hmm I haven't thought of that before really.
>
> **Interviewer:** Are arrays part of the primitive data types?
>
> **Brian$_4$:** No, I don't think so. Because you have methods that only refer to arrays like... and primitive data types don't have methods like that, and in order to use methods you need objects through which you use them and in order to have an object you should have a class so array is a class!

Through this deductive process Brian realises that arrays are objects. Others are more hesitant and do not clearly see the connection, like Patrick who said:

> **Patrick$_4$:** It is a primitive type isn't it? Like integers and so on?... Oh well maybe it is not... No it isn't. It is an object but it is an object that is inherent in it so it is part of Java in general... I am not sure really.

Many students felt that an array is not a primitive data type, but they could not comprehend what it really is. A significant number of responses were similar to Declan's response where he said:

> **Declan$_4$:** I wouldn't say that they are objects... but they definitely aren't primitive data types... because it's more complex than primitive data types, you can do things with it... but you don't write a class for it... it must be something else... I think.

Only three of the students were aware prior to the interview that arrays were objects, and these were the students that viewed arrays as collections of pointers (Category 3 shown in Table 7.1). This is certainly due to their more in-depth knowledge of the language constructs and their experience with other programming languages.

Thus, by the end of the course the majority of the students were not aware that arrays are objects, even if it was written in their lecture notes and presented to them during the course. This could be due to the fact that the syntactical notation of arrays is very different from what students have previously seen when using and developing objects (Howe et al., 2004). The emphasis in this course, and in other introductory courses, is on using and creating objects while learning their syntax. Therefore, even if the students were using the keyword *new* which is associated with object creation, they did not realise that they were creating an array object because they were more focused on the problem-solving aspect of programming. Irrespective of this, most students were able to solve problems that involved using arrays with ease. This means that students do not need to know everything about a construct in order to be able to use it efficiently. However, the awareness of an array as an object allows for a deeper understanding of the construct since the use of *new* and other methods associated with arrays are not merely a syntactical issue but have a specific meaning. Ventura et al. (Ventura et al., 2004) argue that arrays are far too complicated for

Figure 7.3: Logical premises for `while` loops in Java.

```
while (<Boolean expression>)
{

    <statement sequence>

}
```

introductory students and should not be introduced as the first data structure. However, their results are inconclusive and do not show that introducing arrays later in the course after teaching HashMaps had any significant impact on students' understanding (Ventura et al., 2004). This will be further discussed in Chapter 8, where the implications of the results for educators are addressed.

7.3 Students' Views on Iterations

Repetition in programming is a key feature that even the most basic problems require for their solution. Most programming languages provide some sort of repetition constructs and, even if they vary among languages, their purpose is the same: for the program to be able to execute the same sequence of statements a variable number of times. The concept of iteration was taken up in this study as one of the general programming constructs since it is recognised to be an essential part of programming. Iterations are not as multifaceted as the previously discussed themes and could even be considered a simple thing to understand. For most of this study's sample population this was the case, although our analysis showed that complete and deep understanding of iterations is not always reached by all.

In Java there are three different iteration statements, `while` loops, `do-while` loops and `for` loops. All three operate under the same logical premises. All three were taught and used during the course. The most widely used were `while` loops and these were introduced first. The students were then introduced to `for` loops, which were primarily used for looping through arrays. In this study, iterations (loops) were investigated at the conceptual level, while the students' knowledge of the syntactical details and of how individual loops work was not in focus, since these would not constitute an appropriate phenomenon for a phenomenographic analysis. In the following sections the conceptions

Table 7.5: Categories of description for *understanding the nature of iterations.*

Category Label	Category Description
1. Iterations as a static mechanism of repetition	A loop is experienced as a mechanism that allows a block of code to be executed a predetermined number of times.
2. Iterations as a dynamic mechanism of repetition	As above but now the condition that determines the number of iterations can also be dynamic or based on an event.

or understanding of loops that were identified among the sample population are further discussed.

7.3.1 Students' Understanding of the Concept of Loops

Two conceptions were found among the students in this study. Due to the rather simple nature of the concept of loops the variation within the conceptions was limited. Iterations were widely understood as the repetition of a block of code. The difference between the two categories is mainly focused on the nature of the iteration, meaning dynamic or static as shown in Table 7.5. The variation comes from the understanding of the termination condition of the loop. In the first category loops are understood as counter-controlled repetition constructs. Thus, the condition of the loop is always a counter that is either incremented or decremented a predetermined (static) number of times. In the second category counters are not the only way to determine the number of iterations, since the condition can be a run-time or user determined event.

Category 1: Loop as a static mechanism of repetition

In this conception the students experience loops as a mechanism that can repeat a block of code a finite number of times. Thus to create a loop students feel that they need to know beforehand the number of times the loop is going to execute. This suggests that the condition for the termination of the loop is always understood to be a counter of some sort. Alan expresses this when he explains the concept of loops.

Alan$_3$: A loop tells it to do something, tells the computer to do certain commands a

certain amount of times. You can tell it to do it 100 times or two times and it will keep doing it until it is told to stop. It starts off by checking the condition say a counter that counts up to 50. So it checks the condition to see if it is 50 then it is not and then goes on and does it and then you add one to the counter and it will do it until the counter reaches the specific value which is 50.

Interviewer: Yes. Could you have a different condition other than a counter in the loop?

Alan$_3$: What do you mean?

Interviewer: Could there be a case where you didn't know beforehand how many times the loop is going to execute?

Alan$_3$: No... definitely no, unless there was an error and you got an infinite loop...

Alan expresses an understanding of loops where the repetition is static, meaning that the number of iterations has to be known in advance, so no events can interfere with this process. Although he does say "it will keep doing it until it is told to stop" which would suggest that he understands that a run-time event could determine the number of iterations, he later explains that this would not be possible. Like Alan, Declan was explicitly asked about the nature of the terminating condition in the following:

Declan$_3$: So you have a condition and you say until that condition is met you do an operation repeatedly. So you say while number c is less that 6 so we have c = 0 and then we say while c is less than 6 add one to c, so every time we go in the loop we check if c is less than 6 and for as long as c is less than 6 we continue adding ones to it until it reaches 6.

Interviewer: So you use this only with counter conditions?

Declan$_3$: Em... yeah I mean you can use it for different scenarios but you usually yeah you increment numbers really. Like whenever you want to do an operation multiple times then you use loops really. But I only use as conditions like counters so I increment or decrement the counters.

Interviewer: Can you use a loop without knowing beforehand how many times it is going to iterate?

Declan$_3$: Hm... I am not sure... maybe.

Declan's explanation shows how he uses loops with a counter, however he seems unsure whether using the construct without a *definite* repetition would be possible. Other students shared this static understanding of loops, but to avoid repetition only one more interview excerpt will be included in this section. This comes from Tim who said:

> **Tim₃:** Okay the loop is used to repeat a number of instructions until a given condition is fulfilled. So say while `i` is less than 10 repeat this number of instructions for 10 times so each time you increment `i` so you repeat the same action 10 times. So you have a condition that you manipulate to repeat a specific action or a set of instructions.

All the above excerpts demonstrate an understanding where loops can only be counter-controlled. During the course students used loops primarily with counters that were incremented or decremented, however observational data showed that all these students also used loops that were controlled by user events. So is this conception of loops wrong? The answer is no. Even if these specific students have used loops with Boolean or event generated terminating conditions, they were not aware that they were doing it at the time. By following either the lecturer's or the demonstrators instructions there were able to achieve dynamic behaviour in loops but apparently they were not aware or they did not fully understand this at the time. Moreover, this conception would be further encouraged when the students started using `for` loops, since in these loops the counter control is part of the condition. However the students mainly used `while` loops that allow for event terminating conditions as well. Since the students primarily used loops with counters as their termination condition they have developed an understanding where only this aspect is in focus. Therefore this conception is not incorrect but rather it is incomplete.

Category 2: Loops as a dynamic mechanism of repetition

In the second category a loop is experienced as a dynamic mechanism for repeating a block of code, however its termination condition is not only controlled by counter conditions, but rather it is understood as a Boolean (sentinel) condition that can also be event generated. This is expressed by Patrick in his response:

> **Patrick₃:** A loop is used when you want to do something continuously, and carry out a process until one condition is fulfilled. Like for example if `i` was a number to

reach 10, so I will keep adding ones until it reaches that value. So basically it saves you from writing the same thing 10 times. Like it is an efficient way and the syntax is easy. Like for example if you want to do something for 100 times this is better to do it with a loop. Like it doesn't work only with counters it works with `Boolean` values as well. Like do something until this value is true, like stop if you find the letter in an array, or do it until the user presses Q or something. Basically it is an efficient way of doing something repeatedly it saves you a lot of time and space.

He then continues by writing down some of the examples he previously mentioned. What is obvious in the expressed understanding is that the conception of Category 1 is presupposed in this category. However the termination condition for loops is experienced as sentinel-controlled and therefore it can be any run-time event *"do it until the user presses Q or something"* or a generated event *"like stop if you find the letter in an array."* Cormac is more abstract in his response but he express the dynamic repetition view of this category in the following:

> **Cormac$_3$:** It is just a mechanism to avoid repeating yourself like repeating loads and loads of lines of code. And it is not always that you know how many times something can be repeated, it just makes it possible to do something many times. Like you don't know how many people come in first year so if you say while there are still people coming do something.

A final quote come from Stephan who, while talking about the loop's terminating condition, comes to the realisation that counter expressions are also Boolean.

> **Stephan$_3$:** Basically we have some Boolean expression oh.. well in some cases it is not, with the counters... Hmm but yes it is a Boolean expression because it translates to true or false! oh yeah it is! So if it is true then the block of statements enclosed in the brackets it is executed as many times... as long as the Boolean expression is true. Repeating a block of statements without explicitly repeating them physically if you know what I mean. And you can have also infinity loops and loops that you don't know how many times they are going to be executed beforehand.

All of the excerpts above demonstrate an understanding where a loop is a programming construct that allows for repeating a block of code *"without explicitly repeating them physically"* as Stephan says. The difference in this category when compared with the previous

one is that students are aware that no matter what the terminating condition is, it is always a Boolean expression and therefore it can be generalised to be anything, either a predetermined number or a generated run-time event. In this category the understanding is more complete, since loops are fully understood through all their facets. It has to be said that the majority of the population in this study shared this conception.

7.3.2 Critical Aspect of The Students' Understanding of Loops

Loops represent an important control structure, without which students are often incapable of solving simple programming problems that occur frequently in applications. Therefore, a concrete understanding of this basic construct in programming is imperative when learning any programming language. Within the student population of this study it was found that all of them were able to express an understanding of the principal purpose of a loop which is to *"repeat an action or a sequence of actions while some condition holds true"* (Koffman and Wolz, 2001). Although this intrinsic understanding was encapsulated in both categories of understanding, the two categories are clearly distinct. The difference between the two emanated from the nature of the repetition, i.e. static and dynamic. In the first category, students experience loops as static in nature where the repetition is counter-controlled, meaning that a control-variable determines the number of times a set of statements will execute. Counter-controlled repetition is often called *definite repetition*, since the number of repetitions is known before the loop begins executing (Deitel and Deitel, 2005). Thus in this initial category students experience of loops is single-dimensional, focusing only on the definite repetition properties of a loop. Even if these students have actually used loops that were not counter-controlled during the tutorials and their assignments, when specifically asked whether dynamic behaviour could be achieved in loops their response was negative. This indicates that, even in cases where they are developing loops that are not counter-controlled, they are not fully aware of what they are doing, and thus their understanding of the nature of loops is restricted. This understanding is discerned due to the lack of variation in instruction, since most of the loops students are required to use are static, and therefore using control variables only encouraged this view of loops.

In the second category, loops are not only experienced as above, but the understanding

is broadened since the dynamic nature of the construct is brought to the foreground of the conception. Thus, it is understood that the terminating condition for a loop can be any event that can be represented as a Boolean condition. This means that it is not necessary to know beforehand the number of times the code is to be repeated. In the literature this type of repetition is referred to as *sentinel-controlled* repetition, where the sentinel value translates into a Boolean statement and can be any event, like the end of a file or the user pressing a specific key while the program was executing. This sentinel-controlled repetition is also called *indefinite repetition* since the number of repetitions is not known before the loop begins to execute (Deitel and Deitel, 2005). The understanding in this category is more complete since loops are understood in their full, static and dynamic, nature. The dimension of variation in this experience comes from the different types of loops that the students use when solving problems. Thus the variation between problems that require the development of definite and indefinite repetitions constitutes the values in this dimension of variation.

7.3.3 Loop Invariants

There are a number of research projects that investigate how loops should be taught, and at which stage of an introductory programming module (see (Dale, 2005; Walker, 1998; Arnow, 1994; Roberts, 1995)). None of these look at the understanding of loops as a construct from the students' perspective, therefore the results found in this study cannot be compared or related to any in the literature. However in (Walker, 1998), Walker reports that a large number of his first year student sample *"had a great trouble describing ideas behind a loop even after much study and practice"*. This contradicts the findings presented above, as the majority of the students in this study had no difficulty in describing the ideas behind loops and indeed presented a number of examples during the interview session. This was also confirmed by observational data from the laboratory and tutorial sessions where students were often asked to explain line by line their programs and the reasoning behind their solutions. This disparity may be attributed to the different academic background and courses of the students, but it is also an indication of how varied one class can be from another.

Walker (Walker, 1998), amongst others (Nguyen and Xue, 2004), suggests that *loop invariants* should be part of the curriculum and the way that loops should be taught to novices. A loop invariant is a statement that must always be true before, during and after each iteration of the loop, and it is used to help prove that a loop is written correctly. An example of a `for` loop invariant that is used in a sorting function is (Koffman and Wolz, 2001):

```
/* invariant:
* The elements in x[0] through x[fill-1] are in
* their proper place and fill <x.length is true
*/
```

If an educator chooses to use loop invariants when teaching repetition constructs, they would typically engage the students in self-reporting their understanding of the operational premises of the loop conditions. This helps the students develop a vocabulary for expressing their understanding better. However, understanding the loop conditions and expressing them did not appear to be challenging for the group of students in this particular study. In addition, loop invariants tend to be used for counter-controlled loops, since it is easier to express these types of conditions mathematically. Presenting further counter-controlled invariant examples would augment the gap between the students' understanding of the static and dynamic nature of loops, and this is surely not a desired outcome.

Doubtlessly, there are many benefits to using loop invariants when introducing the concept of loops. The main one being that you prevent students from blindly writing something down without thinking, as the process of writing the invariants would require them to think through the design of their solution. It may even help the students whose experience is limited to the static nature of loops, but only if used for dynamic loops with sentinel, as well as counter, values. Therefore, loop invariants may be a powerful technique for reasoning about real programs and complicated loops, but the educator should be cautious in introducing loop invariants for both definite and indefinite loops.

7.4 How Students Understand Selection

Selection in most programming languages, including Java, is primarily achieved through the use of `if` statements. An `if` statement always contains a Boolean expression, and it may have either one consequent or more alternatives. An `if statement,` with two alternatives determines which of two alternate statements will be executed; this is also called an `if-else` statement (Koffman and Wolz, 2001). Students' understanding of selection was investigated in the third interview session where students were asked to discuss their understanding of `if` statements and were also given an exercise (see Appendix B) to solve out loud. At the time of the interview the students were very familiar with the construct and were able to use multiple conditions that were connected with the Boolean AND (&&) and OR operators (||).

A thorough analysis of the interview data yielded a unified understanding of `if` statements, thus no variations in students' understanding were identified. A representative response to the questions *"How do you understand if statements?"* comes from Stephan in the following:

> **Stephan3:** Basically you have the if which is a keyword, then you have the Boolean expression and then if this Boolean expression is true then the sequence which is in the brackets under the statement is executed. Then you can have an else-if which will have another condition and if the condition is not true then the else sequence will be executed and so on and so forth. At the end you have the final else which means that if none of the conditions is true then the last code that is under this final else will be executed. [He sketches his response while talking].
>
> **Interviewer:** What would you say is the purpose of an if statement in programming?
>
> **Stephan3:** Well you need to make decisions, like if you have found that letter in the array print it or something, and so you use them [if statements] all the time... I can't even think of any program I've written that I didn't use ifs!

All the responses followed a similar pattern to Stephan's. The obvious question to ask here is why there was no variation observed in the student responses.

After revisiting the analysis of the data on four occasions and discussing them with other colleagues, it was evident that the outcome space for selection statements was composed

of only one unified category, which was summarised by Stephan above. The reason for this is that the notion of selection or decision making is so closely related to our everyday life that its full and deep understanding comes *naturally* when learning how to use it in programming. All of the students could effortlessly express their understanding. Thus it would appear that some phenomena, such as this one, are so conventional and inherent to our everyday way of thinking and operating, that they yield no variation, or at least they did not bring forth any variation among this study's cohort.

Despite the fact that the students' voiced understanding of `if` statements was homogeneous, when attempting the simple selection problem (see Appendix B) during the interview session they adopted very different strategies to its solution. Some combined all the statements into one Boolean expression connected with logical *and* and *or* operators, while others used nested `if` and `else-if` statements. All of the students managed to solve the given problem, but the variety of approaches adopted suggests that the variation in this phenomenon is hidden in the problem-solving strategies and not the principal understanding of the concept. This however falls outside of the scope of this thesis.

Chapter 8

Different Perspectives on the Study and its Implications

In Chapters 5, 6 and 7 we presented and discussed the phenomenographic results obtained for the twelve themes that are the focus of this thesis. In this chapter we draw on these findings to synthesise the bigger picture of how students experience object-oriented programming by discussing them as a whole and also by exploring the effects of the learning environment on students' experience.

In order to reveal the relationships between the individual phenomena and the whole we first consider them within their initial categorisations of theoretical, object-oriented and general programming components. This provides a means by which we can observe the relationships between the categories of description of the investigated phenomena and study their interactions. Based on these we draw some general conclusions and discuss the implications of the results, both as regards the learning experience and the implications for educators.

Students' experience of the overall educational environment is also explored in this chapter. At the final interview session students were explicitly asked to comment on the elements of the environment which affected their learning experience. Although the data on which this discussion is based were gathered through the interview process, a phenomenographic analysis was not performed as these were considered as comments and specific suggestions on the course in general.

Furthermore, it should be emphasised that this chapter provides an important bridge between the outcomes of this research and teaching practise. Evidently the main body of research in this study was focused on identifying and analysing students' understandings. However drawing from this phenomenographic analysis, and through the use of variation theory, a number of suggestions on teaching approach together with some general classroom interventions were identified. These have the potential of improving teaching and thus learning of programming. Even though a thorough evaluation of the suggestions and interventions proposed is outside the scope of this study, we present a chain of reasoning that supports them based on our findings. These suggestions are further linked to existing studies based on well-practised educational theories in order to provide a more complete and coherent synthesis of the bigger picture that this chapter aims to present. The opportunity to rigorously examine their effectiveness is left to future research projects.

8.1 Relationships Between the Theoretical Components and Implications For Teaching

In Figure 8.1 the categories of description for the three components that constitute the theoretical facet of the students' experience of programming are summarised together with the relationships between them. As was discussed in the phenomenographic analysis of the themes in Chapter 5, each individual understanding of the phenomena represents an important aspect of the experience, since all are needed to reach a deep level of understanding of the theoretical facets of programming.

The first theoretical theme to be analysed was the nature of programming. Four qualitatively different ways of experiencing this were found within the study's population. In the first two categories programming is experienced as the act of coding and as a means of manipulating the hardware, thus the focus is on the pragmatic and tangible elements of programming. These two understandings do not capture the essence of the nature of programming, which is better represented in the last two categories in Figure 8.1.

In relation to the experience of learning to program, six conceptions were voiced by the students. The first four present a partial understanding that is strongly linked to the

output and results of the act of programming such as the syntax, programming constructs and the programs themselves. The last two Categories, 5 and 6, illustrate more abstract understandings that move beyond the obvious to more holistic and complex conceptions.

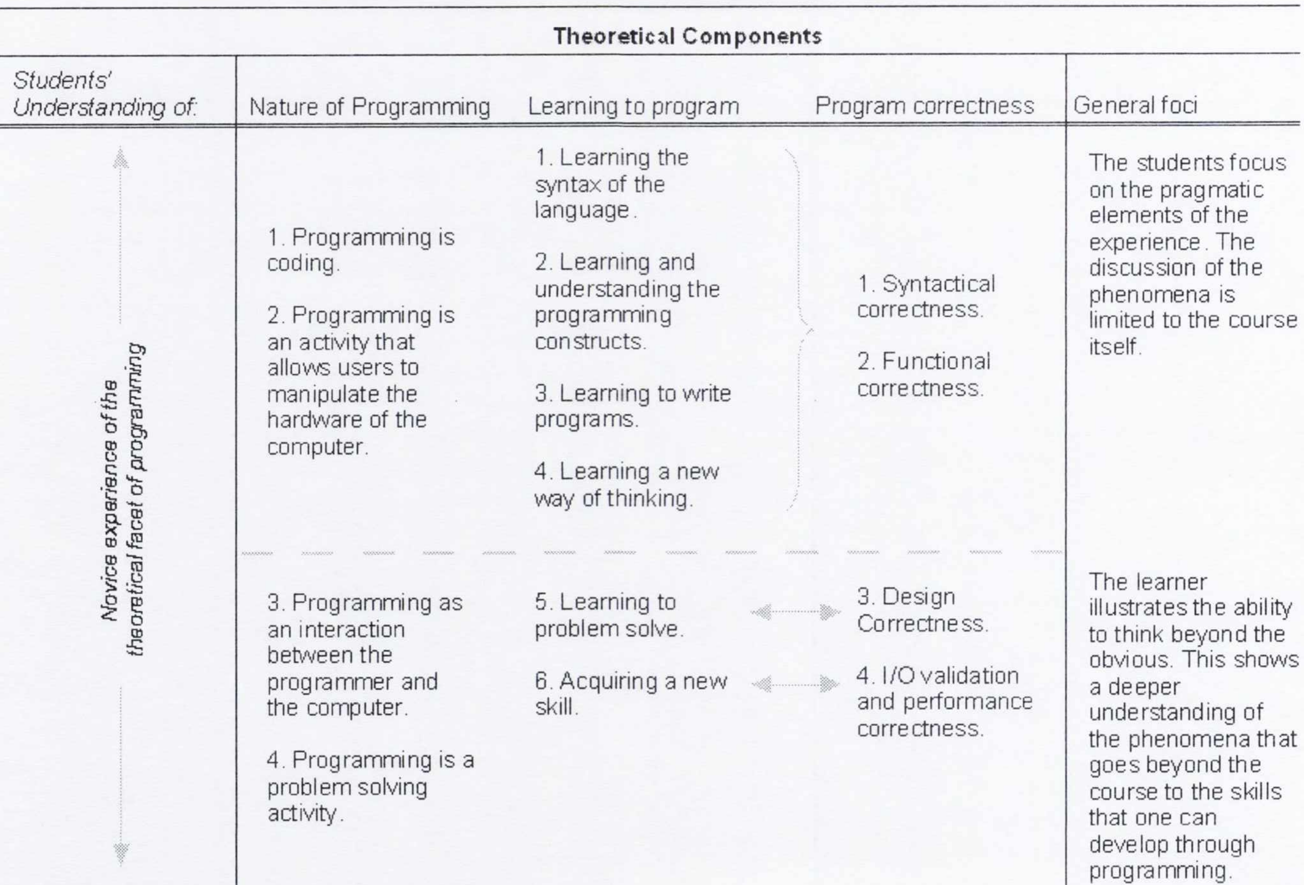
The last theoretical theme investigated was how students experience program correctness. The phenomenographic analysis yielded four qualitatively different ways in which students understand when their programs are correct. The categories progress in a similar fashion to those of the other two themes; with the first two focusing on the tangible elements of correctness (as one student said: *"if the red lines disappear from eclipse"*) while the last two relate to other, non-functional aspects of programming such as I/O validation and design.

Looking at the categories as a whole we can see that their progression and development follows similar patterns. In the early categories, students focus on the obvious and have a basic understanding of the theoretical facets of programming. In Section 5.4 we discussed the relationship between the categories of description that reflect the experience of learning to program and the understanding of correctness. It was observed that there is a correlation between the first four categories of learning to program and the first two categories of correctness, while a one-to-one relationship was found between the last two categories of each theme (as illustrated in Figure 8.1). For example, a student who experiences learning to program as learning to problem solve voiced an understanding of program correctness that was focused on the correct design of the solution.

These observations bring out two points of note: firstly, that in the progression of the categories there is a natural distinction between those that are more naive and those that illustrate a more advanced understanding and secondly, that the students' level of understanding and experience of an individual theoretical component often extends to a similar level of understanding in another theme.

Based on the first point of note, the findings for the theoretical components of the study can be divided into two broad categories based on their general foci. These are illustrated in the last column of Figure 8.1. The general foci of the first categories relate to the pragmatic elements of the course and thus, those in these categories discuss the themes strictly within the boundaries of the course. When discussing the correctness of

Figure 8.1: Relationships between the theoretical components.



Students' Understanding of:	Nature of Programming	Learning to program	Program correctness	General foci
Novice experience of the theoretical facet of programming	1. Programming is coding. 2. Programming is an activity that allows users to manipulate the hardware of the computer.	1. Learning the syntax of the language. 2. Learning and understanding the programming constructs. 3. Learning to write programs. 4. Learning a new way of thinking.	1. Syntactical correctness. 2. Functional correctness.	The students focus on the pragmatic elements of the experience. The discussion of the phenomena is limited to the course itself.
	3. Programming as an interaction between the programmer and the computer. 4. Programming is a problem solving activity.	5. Learning to problem solve. 6. Acquiring a new skill.	3. Design Correctness. 4. I/O validation and performance correctness.	The learner illustrates the ability to think beyond the obvious. This shows a deeper understanding of the phenomena that goes beyond the course to the skills that one can develop through programming.

a program many students expressed their frustration with low grades and the marking system in general. This shows that the students attitude towards learning and studying is affected by their understanding of the nature of programming and learning to program. Moreover, this influences the focus of their study and, in some cases, their capability to study and learn.

The general foci of the later categories show that students whose understanding is embodied in these categories experience programming in a wider context that is not restricted to the programming language and the course. These students seek to improve and extend their personal skills as they emphasise the role of the user and the end-product when discussing correctness. Thus, when the experience is broader and deeper the students' attitude towards learning is different. They approach learning by focusing more on problem-solving rather than by concentrating only on the syntactical details of the language.

Therefore, we can see that learners develop a general attitude towards programming and its various aspects. Thus, it is imperative that educators try and encourage students towards the understanding encapsulated in the latter categories of the themes. One way to do this is by taking into account the critical understandings and the dimensions of variation in teaching that brings forth the awareness of a specific understanding. During the analysis of the theoretical components, specifically in Subsections 5.1.1, 5.2.1 and 5.3.1 of this thesis the educationally critical aspects and the dimensions of variation in the awareness were presented in detail for all three theoretical themes. It is thoroughly documented in (Marton et al., 2004) (and subsequent chapters of that book) that when the dimensions of variations of a subject are taken into account in teaching, students achieve a more complete understanding.

Furthermore, instruction that is highly focused on the programming language and its components is more likely to give rise to strong code-oriented understandings of the nature of programming and the experience of learning to program. An alternate design based on solving problems would favour the development of deeper understandings (as in Categories 3 and 4 of the nature of programming and Categories 5 and 6 of learning to program) as it would encourage students to develop and use the higher cognitive aspects in Bloom et al.'s taxonomy (such as Analysis, Synthesis and Evaluation, see Section 2.2.3) (Bloom et al.,

1956).

The development of this advanced understanding could also be stimulated by the encouragement of discussion on programming problem solutions, either in groups or individually, in order for the students to experience the programming language and the code as a means for solving problems and expressing solutions. Hence, discourse and alternative ways of assessing programs would bring forth the variation that encourages a deeper understanding of the nature of programming and learning to program. The studies of (Fleury, 1993; Booth, 1992) have researched the effects of using discourse on students' understanding of programming with positive results.

It is pragmatic to expect that students' experience of what constitutes a correct program is highly influenced by instruction and assessment. In most programming courses educators are satisfied with a student's program if he/she can demonstrate a working program that can go through a few execution cycles (Ben-David Kolikant and Pollack, 2004; Ben-David Kolikant, 2005). This means of assessment can assure the lecturer that the student's solution will compile and run but at the same time encourages students to experience program correctness as both syntactical and functional: if a program compiles and executes providing a reasonable output, it is right. However, this unintentionally suggests that design and other non-functional considerations are not parts of a correct program. Thus for students to develop the latter understandings about program correctness, educators should use assessment criteria that incorporate elements of design and non-functional requirements. By explicitly discussing the criteria and attributes that a correct solution should have, educators can cause students to redefine their perception of correctness. Research in different educational environments that investigate students views of correctness in programming have highlighted the need for more emphasis on design and non-functional requirements (Fleury, 2000; Ben-David Kolikant, 2005; Scott et al., 2004).

The important point to make in relation to teaching is that problem-based instruction could encourage more holistic understandings (Eckerdal, 2006; Fleury, 2000). In particular educators should be aware of the students' understandings and the variations in awareness that encourage the development of conceptions. Another important aspect in instruction is discourse. Educators should engage students in verbally expressing their thinking process.

Research has shown that this is an effective way of evaluating students' understanding and subsequently improving it (Mannila, 2006; Lister et al., 2004). This would allow the students to become aware of their current level of understanding and then the educator could further provide assistance and explanations that would suit the students' current level of knowledge. As stated above, the relationships that were found to exist between the theoretical components showed that students' level of understanding of a theoretical component often extends to a similar level of understanding to other themes. Thus influencing students' understanding of one of the themes discussed in this thesis may lead students to adopt a different attitude towards programming that would positively influence the outcomes of their learning. This could be further assessed through future research.

8.2 Object-Oriented Components and Implications for Teaching

The object-oriented components explored in this study are the constructs of object, class, attribute, method and constructor. The phenomenographic analysis in Chapter 6 revealed students' understanding of these within the context of a first year introductory course in Computer Science. Figure 8.2 gives the qualitatively different ways that learners experience these central constructs of the object-oriented paradigm and illustrates the relationships these have to their foci.

In Section 6.7 we discussed how the findings of Chapter 6 related to Holmboe's framework of knowledge for object-orientation (Holmboe, 1999). There we found that the categories of our study agree with the four categories that Holmboe suggests. However, when we consider how the conceptions of the object-oriented components develop in relation to each other and independent of any external framework, we observe the emergence of a pattern in their general foci.

For each of the five constructs the first categories found are limited to the description and analysis of the code used to implement the construct. The students gave very basic descriptions, for instance that an attribute is a container like any other variable or that a method is a piece of code. During the interviews the students were asked probing questions

to encourage them to elaborate further, but their understanding appeared to be limited to the programming language and the syntax. This level of understanding was identified for all the constructs and is not incorrect, since all the constructs are part of the programming language and have a syntax of their own. However, either the theoretical extension of each construct and their meaning within the object-oriented paradigm is not fully comprehended by the students or they do not have any awareness of them.

Among the categories found another general foci relates to a pragmatic and practical understanding of the concept. The focus has shifted from the code and moves to the use, and purpose, of the construct. For example, when a constructor is understood as a method used to initialise objects or when classes are experienced as constructs that provide structure to the program. This understanding is important as it encompasses the practical use of a construct and also shows an awareness of how the underlying constructs are part of the object-oriented paradigm.

The last categories of all the object-oriented components, except that for students' understanding of constructors, illustrate that the learner understands the connection between reality and the implemented problem, since the components are experienced as modelling the entities of the real-world phenomena. As the categories are inclusive, the last categories encapsulate an holistic understanding of the constructs. The understanding that is encapsulated in these categories' general foci shows that the students have reached a level of abstract thinking, which may be considered the desired outcome of any programming course. Since the concept of constructor is a more technical construct, which is not as complex as the other object-oriented components, it does not have natural parallels in the real-world and thus is not part of this general foci.

In order for the students to reach an advanced, and complete, understanding of the object-oriented components discussed in this thesis, they need to become aware of various facets of the constructs. As each of the categories of description encapsulates an important aspect of the constructs, an awareness of all of them together is necessary to facilitate the acquisition of a complete understanding. In Chapter 6 we presented the critical aspects of each of the constructs that brings about the awareness of an understanding (see Subsections 6.1.1, 6.2.1, 6.3.1, 6.4.1 and 6.5.1). With the use of variation theory, as discussed in Chapter

Figure 8.2: Relationships between the object-oriented components.

	Object oriented Components					
Students' Understanding of:	Object	Class	Attribute	Method	Constructor	General foci
Novice experience of the Object Oriented components of programming	1. Object as code.	1. Class as code.	1. An attribute is a container.	1. Method is just a piece of code.	1. A constructor is used for creating new objects.	The students focus solely on code and the programming language.
	2, a) Object as a programming construct that holds values.	2. Class provides structure to the program.	2. Attributes determine the functionality of a class.	2. Methods are use to manipulate objects.	2. A constructor is used to initialise object attributes.	The learner focuses on the pragmatic and practical use of the construct when programming.
	2, b) Object as an active entity in the program.	3. Class is a template for objects.			3. A constructor is a specialised method.	
	3. Object is seen as a model of real-world phenomena.	4. Class is seen as a model of real-world phenomena.	3. Attributes are characteristics of the object.	3. A method defines the functionality of a class.		Students experience the connection between reality and the implemented program that is modelled through the use of classes, object attributes and methods.

3, we identified the dimensions of variation that encourage a specific understanding. The use of variation theory has been shown to be an effective tool in teaching and, thus, learning. Several examples of this can be found in (Marton and Tsiu, 2004). Thus the dimensions of variation identified for the object-oriented constructs can be incorporated within the teaching and learning environment to enhance the learning outcomes.

Moreover, practical knowledge of the constructs needs to be acquired in a manner that does not lose track of the theoretical extensions of the programming paradigm and the constructs within it. For example, explicit discourse on the theoretical implications of the construct of class and object is necessary to achieve deeper understanding. However, what usually happens in most introductory courses is that the students are initially introduced to the constructs at the theoretical level, usually during lectures, and then they spend the rest of course implementing and coding without referring back to, or re-evaluating, their theoretical knowledge. Thus, the focus of discourse on principles of the object-oriented paradigm and the theoretical aspects of the programming constructs should be maintained throughout the course. We believe that this would encourage students not to lose track of the bigger picture as their practical knowledge and experience increases. This suggestion for the need of explicit discourse is often brought out in educational research studies of programming and computer science in general (Eckerdal et al., 2005; Bruce et al., 2004; Berglund, 2005; Booth, 1993; Fleury, 2000), however its impact has not been practically assessed.

To help students gain the understanding encapsulated in the latter categories of the object-oriented constructs with respect to their relation to the real-world, it would be beneficial for students to carry out assignments and problems that analyse real-life situations. During the interview sessions students expressed frustration when they did not understand the purpose of having classes for implementing simple tasks. However, with problems that require the design and implementation of several classes and the creations of several objects, students would become aware of the modelling purpose of the programming constructs of object and class. If students were explicitly asked to identify the required attributes and methods before implementing them and to justify their choice, this would help to avoid undesirable situations where, for example, counter or other variables are cast as attributes

of a class. This would further encourage engagement in the higher cognitive processes of Synthesis and Evaluation as per scribed in Bloom et al.'s taxonomy (Bloom et al., 1956).

Practical knowledge of the constructs and their syntax in the programming language is also very important. Thus a great deal of practise and code development should be part of any introductory programming course. However, most programming courses put far too much emphasis on the programming language and the code, neglecting the encouragement of abstract thinking and theoretical understanding of the paradigm. Lister et al. have brought together several ideas for laboratory exercises, examinations etc. that are based on Bloom's taxonomy and that utilise a grading philosophy that emphasises the balance between practical knowledge and abstract thinking skills (Lister and Leaney, 2003). Thus a teaching approach that incorporates such elements in its assessment and exercise philosophy should have beneficial effects on students' learning.

8.3 General Programming Components

The general programming components investigated in this study were students' understanding of algorithms, arrays and iteration. These constructs were chosen because they are common to most programming languages, independent of the programming paradigm. A full comprehension of these is fundamental for students as they are essential in programming and problem-solving. The general programming constructs differ in complexity to those of the object-oriented and theoretical components previously discussed. For example, there is a theoretical aspect to the construct of algorithm as it is an abstract concept, however the construct of an array and the iteration mechanisms are more technical in nature. This meant that less variation was observed in students' understanding due to the heterogeneity and specific technicality of the general programming components. In the following sections we discuss these components separately, proposing ways of enhancing their teaching to encourage deeper level of understandings.

8.3.1 Students' Views on Algorithms and Teaching Implications

The students' conceptions of algorithms were explored as part of their experience within the first year programming course. Since the focus of the course is on object-oriented

programming, algorithms were introduced in an informal manner as a means of problem-solving rather than formally through the use of mathematics, as they would be in a data structures and algorithms course. The investigation yielded three categories of description:

1. Algorithms are seen as methods that include a series of commands designed to solve the problem at hand.

2. Algorithms are seen as a means of reducing the inherent complexity in the problem specification.

3. Algorithms are seen as procedures that represents a step-by-step solution that can be applied to both the physical world and any programming paradigm.

The first category was frequently voiced by the students and was the most common found among the population. This is probably due to the fact that this understanding was actively encouraged through the method of instruction used for this course. When algorithms and problem-solving were initially introduced, students were asked to solve small problems in the form of algorithms, later in the course these were used as part of methods in classes. Thus, students implicitly experienced algorithms as methods.

In the second category the understanding captured by the first category is still present, however the focus has shifted to the result of using various algorithms in a program. Students emphasise the process of decomposing the problem into sub-algorithms that are them implemented in a solution. The result of this process is to reduce the complexity of the problem at hand. To become aware of this understanding of algorithms the students need to experience the process of breaking down problems into smaller sub-problems and the recombination of these to solve the overall problem.

In the last category, the students demonstrated that they understand the relationship between constructing algorithms and solving real-world problems. The students may become aware of this facet of algorithms when solving real-world problems that bring out the modelling aspect of algorithms.

Section 7.1.1 provides a detailed account of how variation theory is used to find the dimensions of variation that encourage a specific understanding of algorithms. A review of several popular textbooks for introductory object-oriented programming courses found

that algorithms are explained and defined in a few lines where very little emphasis is placed on a deep understanding of the construct. Thus, while a deep understanding of the notion of an algorithm is desired, all efforts towards achieving that are implicit. In general, students are not explicitly introduced to the meaning of an algorithm and its purpose in programming in introductory courses.

Thus it is pragmatic to suggest that by explicitly introducing algorithms as procedures that model solutions or situations of the real-world and through carefully chosen assignments that encourage problem-solving and the combination of algorithms to form a single solution, educators can encourage a deep understanding of the construct of algorithm. Heberman et al. have studied novice programmers perception of algorithms in a different educational setting to that of this study and have concluded that "students should be taught that explanations and justification are inherent elements in the solution of an algorithmic problem. Rules of written and oral discourse may be elaborated in order to establish reliable and coherent student-teacher and student-peers communication" (Heberman et al., 2005). It has been shown in (Levitin and Papalaskari, 2002) that puzzle-like, real-world problems encourage the development of creative problem-solving skills. Finally, Parrenet et al.'s preliminary results further support our argument that assignments that incorporate algorithms within program development encourage abstract thinking and deeper understanding of algorithms (Perrenet et al., 2005). Thus our suggestion from theoretical observation and analysis is shown to be effective in practise.

8.3.2 Students' Understanding of Arrays

An array is a data structure found in many programming languages that is frequently used in the development of programs. An array is usually the first data structure that students encounter during a first year programming course and thus it is important that they fully understand it. The phenomenographic analysis in Chapter 7 showed that students experience arrays in the following ways:

1. Arrays are experienced as "something" that can contain a number of objects, or rather as a collection of objects.

2. Arrays are experienced as tools for dynamic object access.

3. Arrays as collections of pointers: this understanding illustrates a "deep" understanding of the memory allocation of pointers and how these are used through an array.

In the first category, the notion of an array is not very clear to the students. Arrays are not understood as structures with properties and methods, rather the focus is placed on the elements that are contained in it. As students create a variety of arrays with different contents in their program solutions they become aware of this understanding. However this is a naive understanding, since arrays are not viewed as constructs. This may impede the students in their utilisation of the construct and in their appreciation of its capabilities.

In the second category, which was voiced by the majority of students in this study (see Table 7.4, Section 7.2.3), arrays are experience as constructs emphasising their capability of dynamic access. Accessing the stored elements of an array in a non-sequential manner brings this understanding to the foreground of the students' experience.

Only in the third category did students express an understanding of how the construct works in terms of memory representation. As Java is a high level object-oriented language memory management and pointers are hidden from the programmer. However an understanding of the internal operations of the construct demonstrate a richer understanding that allows for better use of the construct.

In Section 7.2.4 we analysed and discussed students' understanding of whether an array is a class and therefore if an array is created, it is an object. Although most students did not experience an array as an object, they were aware of the fact that it was not a primitive data type but at the same time they could not identify it as an object as it was not similar to the ones that they have seen and developed themselves. During lectures the structure of an array as a collection of pointers, and the fact that it is a built-in class, was specifically presented to the students. This is also explicitly stated both in their suggested text book and their lecture notes. However during both the interview and tutorials the students did not demonstrate this understanding. Thus a theoretical presentation of the architecture of the structure of array is not sufficient for students to experience it and understand it in its entirety. Most of the time students focus on understanding how to use a construct to solve the problem at hand and thus the underlying architecture of the construct is not as important for them. We believe that in order to encourage a deeper understanding

educators should approach the introduction of the architectural aspect of arrays through practical programming examples. Howe et al. argue for the appropriateness of component-first teaching of programming (Howe et al., 2004). Most importantly, they point out that when the students are only taught about arrays through practical use, misconceptions might arise due to the fact that students have not achieved a deep understanding of the architectural aspects of the construct (Howe et al., 2004).

Ventura et al. (Ventura et al., 2004) investigated whether introducing `HashMaps` as the first structure that students encounter in an introductory programming course would ease students' understanding of arrays. Their results of experimenting with both `HashMaps` and collections (arrays) did not show any significant differentiation as to whether one is easier to understand than the other. Our findings show that students can use arrays easily, even if they are not always aware of the architectural structure of the construct. Thus we believe that not only should the architectural details of an array be included in instruction, they should also be presented through exercises that lead students to think, and become aware, of the correct architecture via experience. This was evident when students where asked if an array they had created is an object: they simply had not thought about it. For example some of the students concluded that an array of integers is an object only after thinking about it during the interview sessions. If students were encouraged to think of this aspect of the construct then it may be the case that they would develop a deeper, more complete understanding during their course of instruction.

8.3.3 Students' Understanding of the Mechanism of Iteration in Programming

Iterations in Java are implemented through the constructs of `for`, `while`, and `do-while` loops. The participating students were asked to discuss their understanding of the nature of iterations irrespective of the construct that was used to achieve it. The phenomenographic analysis yielded two qualitatively different ways of experiencing the nature of iteration in programming.

1. Iterations as a static mechanism of repetition. A loop is experienced as a mechanism that allows a block of code to be executed a predetermined number of times.

2. Iterations as a dynamic mechanism of repetition. As above, but now the condition that determines the number of iterations can also be dynamic or based on an event.

In the first category the student is not aware of the dynamic conditions that can determine the number of iterations, while in the second the dynamic properties are part of the conception. Obviously the first understanding can lead to problems when the number of iterations is not known beforehand or depends on a condition set at run-time execution. Static iteration with the use of counters is usually introduced first as it is easier for students to understand. However, it is imperative that students gain an understanding of the dynamic conditions of the repetition mechanisms.

As discussed in Section 7.3.1, students have primarily used `while` loops both with counters and other dynamic and run-time conditions. However, when asked to verbalise their understanding and even when explicitly asked if there could be loops without a predetermined number of iterations, a significant number of students failed to demonstrate that they fully understood the notion of dynamic or run-time conditions, even if they have used them. This is evidence of the poor reflective capability of the students, as some were not in a position to explain how they developed the loops but would say *"I just did"*.

We believe that, in order to address this issue, educators should approach instruction of the repetition mechanisms with carefully selected examples that emphasise both their static and dynamic properties. Introducing loops through formal techniques such as loop invariants stating the pre- and post-conditions, can prove powerful tools for educators (see Section 7.3.3). However, the most important element would be to encourage students to explain their decisions and learn how to reason and explain their programs. Through discourse, students develop the vocabulary they need to achieve such outcomes (Soloway, 1986). Moreover, talking will allow them to reflect on their own knowledge and become aware of the properties of programs that would otherwise remain in the background of their experience.

8.4 The Effects of the Learning Environment

The learning environment is an important element of students' experience of learning programming. At the last interview session students were asked to comment on their

learning environment as regards the beneficial parts of it and the things that they thought could be improved. A detailed account of the questions that students were asked can be found in Appendix B.

Generally, students were content with the way the material was introduced and the volume of course work and feedback they got throughout the course. As this particular class was fairly small, around 40 people, there was enough time for the demonstrators and teaching assistant to attend to each student's individual needs. The students were asked specifically to comment on the learning environment in terms of the lectures, laboratory and tutorial sessions.

As regards the lectures, the majority of the students said that they found them repetitive and not always very exciting. Alan expresses that feeling quite strongly in his response:

> **Interviewer:** You said before that you found lectures to be less beneficial than the tutorial and lab. Do you care to explain a bit more?
>
> **Alan $_4$:** He tended to repeat things... it was kind of boring to be honest because we would sit there and then he would talk about these... and then I would take my laptop and play basically [laughs]. Probably theory it is not that interesting but it is really required I can see that. But in the tutorials it was different... You would do something and then if it wasn't right or you didn't know how to do it, you guys would explain the theory and what was wrong straight away so it would stick better it was generally easier. I mean in 1 hour tutorial I would learn whatever I learnt in 2 hours of lectures. I think it was too repetitive.

Alan recognises that the theoretical part of programming is important and necessary but at the same time when it is conducted solely without practical examples and applied exercises students experience it as repetitive and not very interesting. Alternative Alan and other students find tutorials to be more beneficial as they can discuss their understanding, or the lack of it, at a time when they become aware of it. Additionally, Declan says about the lectures:

> **Declan$_4$:** I liked the arrangement but I felt that we could have done more examples, because when the demonstrator substituted once he had lots of examples and gave us time to do a little bit of code and I found that very helpful.

Both claims support our suggestion for increased discourse at the level of each student's understanding. They also suggest that theory and the various uses of the constructs should be introduced through a combination of theory and practical applications.

Students also commented that they found the tutorial sessions to be very helpful due to the group work involved. Although the students were supposed to complete the programming tasks individually, it was usually the case that they would form small groups and would discuss their ideas, problems and solutions together. Brian and Neil summarise this experience in the following:

> **Brian$_4$:** Hmm I would probably say the tutorials would be the best. Because in the tutorials... I don't know if it should be this way... but it is more of a group effort like (giggles) because there is hardly going to be any case that you wont be able to do it. So sharing ideas with someone else was just a great way of learning really, because we were all at the same level so it was interesting and we were kind of learning from each other as well! It is just better learning this way than sitting in a lecture and then theory.
>
> **Neil$_4$:** Probably the tutorial just because you are able to code with other people, although most probably the other people they are just copying you and using your stuff (laughs) but you exchange ideas and this is awesome!

It is evident from these statements that students find it beneficial to work in small groups where they learn from their peers. Patrick points out that during the tutorials he could practise the "problem-solving part", as he expressed in the following:

> **Patrick$_4$:** I would say definitely the tutorial because you get hands-on experience and if you don't understand something then somebody will explain it to you straight away, so I would say the labs and tutorial. But I suppose the tutorial most because you get the problem-solving part of it as well, because syntax is not the most important feature I guess. The labs were most about compiling the thing and making it show that it runs above all.

Thus by designing the course in such a way that exercises can be completed within small groups, educators can effectively enhance their students' learning experience and potentially achieve a better outcome.

Students also commented positively on the learning benefits of the laboratory sessions. Students who had previous programming experience found the laboratory sessions especially beneficial as they actually got to work with, and write in, the syntax of the language. Stephan said the following about laboratories:

> **Stephan$_4$:** Hmm the labs definitely, the lecturers are obviously the most beneficial but the labs were the more you get basically to write a lot less than in the tutorials and you get to see it run as well and the errors that you make because the environment highlights them to you alright.

Few students found laboratory sessions more beneficial than tutorials due to the fact that they could see if their solution worked without having to wait for feedback and the corrections to their tutorial exercises. Seeing the output of your solution is a big part of the joy and sense of accomplishment one gets from programming. Moreover, programming as an activity is about working with computers. Thus the weekly laboratories are essential for experiencing this aspect of programming. As Alan summarises in his response:

> **Alan$_4$:** [...] In the tutorials we just write the whole thing on paper and you don't really know if this is going to work like... While in the labs you can test it immediately and then say yeah it works. Hurrah!

In the laboratory sessions students had the opportunity to experiment with the programs and the syntax through trying out ideas and getting familiar with the language. This reflected positively in, and enriched, the students' learning experience. A big part of learning to program is practising with language and different problems, as Liam points out:

> **Liam$_4$:** I don't think you can lecture somebody to programming I think he should be giving more exercises to do like the tutorials or the labs even, so I think more interaction is needed more exercises.

The students' responses draw a clear picture of the beneficial aspects of the learning environment and highlight improvements that could be incorporated into the instruction used on the course. Students preferred a more practical, hands-on method of instruction instead

of a pure, theoretical introduction to the various programming constructs. They found discourse and practical work within small groups to be beneficial and enjoyable while they felt the experience of working in the laboratories with the programming environment was essential for their learning and familiarity with the programming language. These suggestions agree with the recommendations that we have provided in this chapter in relation to the development of a deeper understanding of the components of programming discussed in the thesis.

8.5 Impact of the Context on the Findings

This study investigated students' understanding of object oriented programming within the context of a specific course module. The module comprised an articulated teaching strategy, an underlying programming language, Java, and a specific development tool, Eclipse. These factors undoubtedly had an effect on the variation of understandings that may be present within the source material, more so in some constructs than in others. The tools and the environment within which one learns affect the process of reasoning and thus the understanding that one develops.

For some of the themes in this study these effects are minimal, for example the theoretical components as they are less reliant on the language. For others, such as constructors, arrays and object-oriented themes that depend on the language itself, this may be more important.

To more fully explore the significance, and impact, that the context of the study had on the findings, it would be necessary to perform similar studies in different educational settings, with different programming languages etc. Since other such studies do not currently exist, it is difficult to accurately portray and assess the impact of the context on the findings. This study explicitly details the educational setting, course, syllabus and how the material was taught in order to allow the reader to independently evaluate any potential influences of the context of the study that may be reflected in the findings.

The specific selection of the course and the educational setting was important in seeking to minimise the effect of context on the studys' findings. The course and the programming language used in the course selected are as generic as reasonably possible - thereby mirror-

ing the educational setting in other first year object-oriented programming courses. The students were similarly representative of typical undergraduate cohorts. Some had prior experience, some had never programmed before while others were repeating or students who had transferred from other courses. This diversity within the population should help ensure that the sample population is representative of an typical first year programming course grouping.

Throughout the analysis it was observed that students with programming experience frequently (although not always) perceived the constructs and themes in a more complete and advanced way. Where relevant this is highlighted within this thesis but the purpose of this study was not to investigate if students' prior programming experience (or other factors such as gender) affects their performance and their perception of programming. Thus, when prior programming experience could have influenced the existence of an understanding it is noted accordingly to allow the reader to draw his or her conclusion of the possible influence of the context in the findings of this study. Similarly any influences that were caused by the teaching strategy in the form of a programming exercise or example are also noted.

8.6 Summary

This chapter discussed the variation in students' experience of learning about theoretical, object-oriented and general programming constructs. Based on the educationally critical aspects that encourage the development of specific understandings, we have identified several ways in which the educational environment and instruction could be enhanced to potentially improve the learning outcomes. These suggestions for teaching have been proposed based on the critical aspects of students' understanding identified herein and, thus they may act as a guide to educators who want to improve their teaching. Their actual effectiveness has not been empirically established as this falls outside the scope of this study. However, evidence of the effectiveness of the suggestions deduced herein can be gleaned from the literature where others have documented similar outcomes via different methodologies.

A common implication for instruction that arose in relation to all of the components of this study was discourse. Writing a program that compiles and runs should not be

considered sufficient by either the student or the educator. Students should be encouraged to explain their design decisions and also learn how to reason and explain their programs. Through discourse students can develop the vocabulary needed to explain their program solutions, while at the same time talking about these will allow them to reflect on their own knowledge and become aware of aspects of programming that would otherwise remain in the background of their experience. The students' experience of their learning environment strongly supports these arguments as they specifically requested more group work and an amalgamation of the predominantly theoretical lectures with the more practical tutorial and laboratory classes.

Chapter 9

Conclusions

Programming is an essential, core module of every Computer Science and Engineering degree course. Object-oriented languages are now the dominant programming paradigm in introductory programming courses. Many argue that the ongoing switch from procedural languages to Java and C++ is a step forward, as it leads to better program organisation and the development of abstract thinking skills (Berge et al., 2003; Gries, 2002). However, learning to program has proven to be a challenging activity from the students' perspective (Aharoni, 2000; Barr et al., 1999; Carter and Jenkins, 1999). Moreover, research in the field has shown that more abstract types are required when learning to program within the object-oriented, rather than the procedural, paradigm (Box and Whitelaw, 2000). Furthermore people have high expectations of programming students, as Gries points out *"[...] programming is more than a bunch of facts. It is a skill, and teaching such a skill is much harder than teaching physics, calculus or chemistry. People expect a student coming out of a programming course to be able to program any problem. No such expectations exists for calculus or chemistry students."* (Gries, 2002). Put simply, learning to program within the object-oriented paradigm is a complex process.

The primary motivation for this study was to gain an insight into students' understanding of object-oriented programming in its various facets. In order to draw a complete picture of what it means for novice Computer Science students to learn how to program, we have explored their understanding in relation to the theoretical, object-oriented and general programming aspects of the experience. The longitudinal study that is presented

in this thesis provides an in-breadth and in-depth insight into the students' experience and understanding of the most fundamental concepts of learning to program within the object-oriented paradigm using the methodologically-anchored research approach of phenomenography.

The analysis of the data was performed in a systematic way. The development of the categories is fully documented through notes, memos and diagrams in the analysis software that was used, ATLAS.ti. The data were analysed by recourse to the initial research questions that formed the basis for this work. The categories of description that emerged for each theme of this study were established through an iterative and systematic study of the data, where feedback from previous iterations gave rise to further refinement for each category. The guidelines for ensuring the validity, reliability and generalisability of the study, discussed in Chapter 3, were rigorously applied from the early stages of data gathering to the final presentation of the findings. Before the complete set of categories were finalised these were discussed with both the researcher's supervisor and also with other colleagues in the Computer Science department. Excerpts of the interview data were presented throughout the analysis in order to support the findings and conclusions of this study. The results, in the form of categories of description, are thoroughly discussed within the phenomenographic tradition. The critical aspects of understanding are also explained through the use of variation theory. The rich findings of this study are presented alongside, and are related to, relevant results from other studies in the field, allowing the reader to fully appreciate their relevance and importance within the wider field.

The research questions presented in Chapter 1 were fully addressed throughout this thesis. More specifically:

- The question of *"How do students experience learning to program?"* was investigated explicitly during the interviews. Discussions on the nature of programming and on the students' understanding of program correctness were used to provide a more complete view of the students' experiences. These results were given in Chapter 5.
- The question about students' understanding of the object-oriented components, was explored through the investigation of their conceptions of the constructs of object, class, attribute, method and constructor. These findings were presented in Chapter

6.

- The third research question *"How do students reason about specific programming problems, and programming constructs as a whole?"*, was explored through the investigation of understanding of the general programming constructs of algorithms, arrays, iterations and selection, for further details see Chapter 7.
- The effects of the learning environment and how it is experienced by students was explored in the last interview session which was analysed in Chapter 8.
- The final research question, *"How students' understandings can be used to enhance teaching and therefore improve the quality of learning?"*, was addressed throughout this thesis. Chapter 8 presented the themes in their totality and discussed ways of enhancing teaching based on the findings of this study while it relates the proposed interventions and teaching techniques to learning theories and established research in the field.

Therefore, this thesis realised its overall goal of providing an in-depth and complete picture of how Computer Science students understand and experience object-oriented programming. The findings were presented in detail to allow the reader to recognise their validity, reliability and generalisability; while extensive discussions and comparisons with other relevant projects were used to site the work clearly within the wider field.

9.1 Significance of this Study

The study presented in this thesis incorporates elements from three different areas, namely: Computer Science education research, Computer Science education, and Computer Science. Thus, it constitutes a contribution to all three of these fields. Both the contribution and significance of this empirical study to the body of knowledge for these fields is considered in this section.

Two major contributions to Computer Science education research and Computer Science are identified in this theses, namely.

- The investigations of the themes included in this study, together with the extensive data collected provides a novel in-depth and in-breadth insight into students' learning. This constitutes a significant contribution to the field of Computer Science education research.

- The study as a whole provides a valuable insight into the students' overall understanding of computer programming. It provides a complete picture of the experience of first year programming, ranging from the theoretical aspects of programming to the technical ones. Hence this work provides educators with the necessary confidence and knowledge to adapt and enhance their teaching to create a learning environment best suited to the achievement of their desired learning outcomes, and thus contributing to the field of Computer Science education.

Apart from these two major contributions there are other points of significance in this study that are discussed below.

The field of Computer Science education research is a relatively new field when compared to others such as Mathematics education research. Indeed, the discipline of Computer Science itself is young and rapidly evolving. As presented in the literature survey, studies that investigate the students' experience of learning Computer Science topics are very few. Thus, the completion of this study presents a significant contribution to the field of Computer Science education research, while also linking the qualitative results to enhancement of teaching.

In addition, this study is conducted within the phenomenographic tradition, and as such it adds to the body of knowledge, development and dissemination of this research approach.

The findings have been used throughout this work to explore the educationally critical aspects that significantly effect learning to program, as identified by the learners themselves. Moreover, the findings illustrate the level of students' understanding achieved. Therefore, the implications of the findings allow us to provide suggestions on ways to enhance the teaching of key programming concepts with the aim of improving learning. This constitutes a clear contribution to the field of Computer Science education.

This research project has been conducted within the subject domain of Computer Sci-

ence and, as such, is primarily targeted at evaluating learning and enhancing teaching within this subject area. However it has employed a theoretically anchored approach, namely phenomenography and variation theory, to study the students' understanding. Therefore, it may be applied to other similar educational settings. We have already compared our findings to a number of relevant studies that were conducted in different educational settings and found that our results can be further generalised, therefore contributing to programming education in general.

As mentioned in earlier chapters of this thesis, other phenomenographic studies have been carried out in the field of Computer Science that are focused on programming, more specifically (Booth, 1992; Bruce et al., 2004; Eckerdal, 2006) (see Chapter 2 for a detailed description). However, none of these has the depth of analysis or the breadth of themes present in this study. Additionally, these studies are not focused on investigating object-oriented programming within Computer Science, rather they explore specific constructs and elements of programming. Therefore, the breadth of the themes investigated, and the depth of our analysis, along with the extensive discussion of the findings in the light of related studies and theories, brings out the novelty of our research.

9.2 Further Research

The data collected for this research project are extremely rich and diverse; thus allowing for the investigation of many detailed research questions. Only some of these were analysed in this thesis. Not only does this data set contains the transcripts of detailed interviews with the students but it also contains their solutions to a number of programming exercises, their grades on assignments and laboratory exercises, along with a comprehensive set of observational data. Thus it will be possible to carry out further research using the same set of data. For example, it is possible to analyse students' strategies when learning to program or when developing algorithms and then further correlate these results with the findings of this study. Although interesting, these questions are outside the scope of this study, but should make for compulsive future research.

Interview data were also gathered for other themes such as students' understanding of inheritance, encapsulation, software engineering, object-orientation and their beliefs on

the characteristics of a good programmer. These themes were beyond the scope of this thesis. Preliminary sorting and analysis was conducted on these data, preparing the way for future research.

Through the analysis and discussion of the findings of this work we have identified the critical aspects that encourage specific understandings of programming and programming constructs. Based on these we have made a number of logically grounded suggestions for enhancing teaching and the learning environment. Empirical support for the effectiveness of these suggestions is now required.

A final research suggestion would be for a similar study on understanding to be carried out with the educators, teaching assistants and demonstrators that teach object-oriented programming. As pointed out in the analysis of the themes of this study, teaching and the structure of the learning environment in terms of lectures, tutorials and assignments affects students' experience and the understanding they develop on programming constructs. Gaining an understanding of how educators experience these, and comparing them with the students' conceptions, would help to improve learning outcomes by explicitly guiding teaching and allowing educators to become more aware of their own understandings, strengths and shortcomings.

9.3 Final Remarks

It is expected that the research presented in this thesis will lead to an improvement in teaching and student learning of programming, specifically of the object-oriented paradigm. This will, in turn, result in better-prepared programmers who have acquired an holistic experience of learning to program, allowing them to become informed practitioners in the field. As programming paradigms develop and computers become ever more pervasive, programming will remain a critical element of any Computer Science degree program. This reinforces the need for continuous research in order to assimilate ongoing developments. We hope that this work is an inspiration to others to carry out research that extends our knowledge of students' understanding, allowing us to enhance the quality of students' experience and significantly improve their learning outcomes.

Bibliography

Aharoni, D. (2000). Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science Education*, pages 26–30.

Akerlind, G. (2002). Principles and Practices in Phenomenographic Research. In *Proceedings of the International Symposium on Current Issues in Phenomenography*, page 17p.

Akerlind, G. (2005). Phenomenographic Methods: A Case Illustration. In Bowden, J. and Green, P., editors, *Doing Developmental Phenomenography*, pages 103–127. RMIT University Press, Melbourne.

Akerlind, G., Bowden, J., and Green, P. (2005). Learning to do Phenomenography: A Reflective Discussion. In Bowden, J. and Green, P., editors, *Doing Developmental Phenomenography*, pages 74–100. RMIT University Press, Melbourne.

Anderson, L. W. and Krathwohl, D. A. (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Addison-Wesley.

Arnow, D. (1994). Teaching Programming to Liberal Arts Students: Using Loop Invariants. In *Proceedings of the 25th SIGCSE technical symposium on Computer Science Education*, pages 141–144.

Barnacle, R. (2005). Interpreting Interpretation: A Phenomenological Perspective on Phenomenography. In Bowden, J. and Green, P., editors, *Doing Developmental Phenomenography*, pages 47–55. RMIT University Press, Melbourne.

Barr, M., Holden, S., Phillips, D., and Greening, T. (1999). An Exploration of Novice Programming Errors in an Object-Oriented Environment. In *Working group reports from ITiCSE on Innovation and Technology in Computer Science Education workshop*, pages 42–46.

Ben-Ari, M. (2001). Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73.

Ben-Ari, M. (2004). Situated Learning in Computer Science Education. *Computer Science Education*, 14(2):85–100.

Ben-David Kolikant, Y. (2001). Gardeners and Cinema Tickets: High School Students' Preconceptions of Concurrency. *Computer Science Education*, 11(3):221–245.

Ben-David Kolikant, Y. (2005). Students' Alternative Standards for Correctness. In *Proceedings of the 1st International Computing Education Research Workshop*, pages 37–43.

Ben-David Kolikant, Y. and Pollack, S. (2004). Establishing Computer Science Professional Norms Among High-School Students. *Computer Science Education*, 14(1):21–35.

Berge, O., Fjuk, A., Groven, A. K., Hegna, H., and Kaasbøll, J. (2003). Comprehensive Object-Oriented Learning - An Introduction. *Computer Science Education*, 13(4):331–335.

Berglund, A. (2004). A Framework to Study Learning in a Complex Learning Environment. *Research in Learning Technology*, 12:65–79.

Berglund, A. (2005). *Learning Computer Science in a Distributed Project Course: the What, Why, How and Where.* PhD thesis, Uppsala: Uppsala University, Interfaculty Units, Acta Universitatis Upsaliensis.

Berglund, A., Daniels, M., and Pears, A. (2006). Qualitative Research Projects in Computing Education Research: An Overview. In *Proceedings of the 8th Australasian Computing Education Conference*, pages 25–33.

Berglund, A. and Eckerdal, A. (2006). What Do CS Students Try to Learn? Insights From a Distributed, Project-Based Course in Computer Systems. *Computer Science Education*, 16(3):185–195.

Beyer, S., DeKeuster, M., Walter, K., Colar, M., and Holcomb, C. (2005). Changes in CS Students' Attitudes Towards CS Over Time: An Examination of Gender Differences. In *Proceedings of the 36th SIGCSE technical symposium on Computer Science Education*, pages 392–396.

Biggs, B. J. and Collis, F. K. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy.* Academic Press, London.

Bishop-Clark, C. (1995). Cognitive Style, Personality and Computer Programming. *Computers in Human Behaviour*, 11(2):241–260.

Björkman, C. (2002). *Challenging Canon: the Gender Question in Computer Science.* Licentiate thesis. Karlskrona: Blekinge Institute of Technology.

Bloom, B. S., Mesia, B. B., and Krathwohl, D. R. (1956). *Taxonomy of Educational Objectives (two vols: The Affective Domain andThe Cognitive Domain).* Addison-Wesley.

Booth, S. (1992). *Learning to Program a Phenomenographic Perspective.* PhD thesis, Acta Universitatis Gothoburgensis.

Booth, S. (1993). A Study of Learning to Program From an Experiential Perspective. *Computers in Human Behavior*, 9:185–202.

Bowden, J. (2005). Reflections on the Phenomenographic Team Research Project. In Bowden, J. and Green, P., editors, *Doing Developmental Phenomenography*, pages 11–31. RMIT University Press, Melbourne.

Box, R. and Whitelaw, M. (2000). Experiences When Migrating from Structured Analysis to Object-Oriented Modelling. In *Proceedings of the ASE Australian conference on Computer Science Education*, pages 12–18.

Brooks, R. (1977). Towards a Theory of the Cognitive Processes in Computer Programming. *International Journal of Man-Machine Studies*, 9(6):737–751.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., and Stoodley, I. (2004). Ways of Experiencing the Act of Learning to Program: A Phenomenographic Study of Introductory Programming Students at University. *Journal of Information Technology Education*, 3:143–160.

Bruner, J. (1960). *The Process of Education.* Harvard University Press.

Bruner, J. (1990). *Acts of meaning.* Cambridge, MA: Harvard University Press.

Byrne, P. and Lyons, G. (2001). The Effect of Students Attributes on Success in Programming. In *Proceedings of the 6th SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, pages 49–52.

Cahill, V. (2001). Learning to Program the Object-Oriented way with Java.

Carbone, A. and Kaasbøll, J. J. (1998). A Survey of Methods Used to Evaluate Computer Science Teaching. In *Proceedings of the 6th annual conference on the Teaching of Computing and the 3rd annual Conference on Integrating Technology into Computer Science Education*, pages 41–45.

Carter, J. and Jenkins, T. (1999). Gender and Programming: What's Going On? In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, pages 1–4.

Clancy, M., Stako, J., Guzdial, M., Fincher, S., and Dale, N. (2001). Models and Areas for CS Education Research. *Computer Science Education*, 11(4):323–341.

Clements, D. and Gullo, D. (1984). Effects of Computer Programming on Young Children's Cognition. *Journal of Educational Psychology*, (76):1051–1058.

Cohen, L., Manion, L., and Morrison, K. (2000). *Research Methods in Education.* Routledge Falmer.

Conway, D. (1999). *Object-Oriented Perl.* Manning Publications Co.

Cope, C. (2000). *Educationally Critical Aspects of the Experience of Learning about the Concept of an Information System.* PhD thesis, La Trobe University, Bundoora, Australia.

Cuba, E. (1981). Criteria for Aseerting the Trustworthiness of Naturalistic Inquiries. *Educational Communication and Technology Journal*, 29(2):75–91.

Dale, N. (2005). Content and Emphasis in CS1. *ACM SIGCSE Bulletin*, 37(4):69–73.

Deitel, H. M. and Deitel, P. J. (2005). *JAVA: How to program: with an introduction to J2SE 5.0 with the UML 2*. Prentice-Hall.

Détienne, F. (2002). *Software Design - Cognitive Aspects*. Springer, London.

Dijkstra, W. E. (1982). How Do We Tell Truths That Might Hurt? *ACM SIGPLAN Notices*, 17(5):13–15.

Doyle, E., Stamouli, I., and Huggard, M. (2005). Computer Anxiety, Self-Efficacy, Computer Experience: an Investigation throughout a Computer Science Degree. In *Proceedings of the 35th ASEE/IEEE FIE, Frontiers in Education Conference*, page 5p.

Dukes, J. (2005). 1ba2 Introduction to Programming, `https://www.cs.tcd.ie/courses/ba/localMerged/jf/syllabus.php/`.

Eckerdal, A. (2006). *Novice Students' Learning of Object-Oriented Programming*. PhD thesis, Uppsala University, Department of Information Technology.

Eckerdal, A., Thuné, M., and Berglund, A. (2005). What Does it Take to Learn 'Programming Thinking'? In *Proceedings of the 2nd International Computing Education Research Workshop*, pages 135–142.

Eclipse-Foundation (2005). Eclipse.org Home Page, `http://www.eclipse.org/`.

Eriksen, T. H. (2001). *Small Places, Big Issues: An Introduction to Social and Cultural Anthropology*. Pluto Press, London, UK.

Felder, R. M. (1996). Matters of Style. *ASEE Prism*, 6(4):18–23.

Fincher, S. and Petre, M. (2004). *Computer Science Education Research*. Routledge Falmer.

Fitzgerald, S., Simon, B., and Thomas, L. (2005). Strategies that Students Use to Trace Code: An Analysis Based in Grounded Theory. In *Proceedings of the 1st International Computing Education Research Workshop.*

Fleury, A. E. (1993). Students' Beliefs about Pascal Programming. *Journal of Educational Computing Research*, 9(3):355–371.

Fleury, E. A. (2000). Programming in Java: Student-Constructed Rules. In *Proceedings of the 31st SIGCSE technical symposium on Computer Science Education*, pages 197–201.

Gal-Ezer, J. and Harel, D. (1999). Curriculum and Course Syllabi for High School CS Program. *Computer Science Education*, 9(2):114–147.

Gall, D. M., Borg, R. W., and Gall, P. J. (1996). *Educational Research: An introduction.* Longman Publishers, USA.

Glaser, B. G. (1992). *Basics of Grounded Theory Analysis: Emergence vs Forcing.* Mill Valley, Ca.: Sociology Press.

Gray, A., Jackson, A., Stamouli, I., and Tsang, S. L. (2006). Forming Successful eXtreme Programming Teams. In *Proceedings of the IEEE, Agile International Conference*, pages 390–399.

Green, P. (2005). A Rigorous Journey into Phenomenography: From a Naturalistic Inquirer Point of View. In Bowden, J. and Green, P., editors, *Doing Developmental Phenomenography*, pages 32–46. RMIT University Press, Melbourne.

Green, T. R. G. (1989). Cognitive Dimensions of Notations. In Sutclife, A. and Macaulay, L., editors, *People and Computers V*, pages 443–460. Cambridge University Press.

Greening, T. (1996). Paradigms for Educational Research in Computer Science. In *Proceedings of the 2nd ACSE Australian conference on Computer Science Education*, pages 47–51.

Gries, D. (2002). Where is Programming Methodology These Days? *ACM SIGCSE Bulletin*, 34(4):5–7.

Hazzan, O. (2002). Reducing Abstraction Level when Learning Computability Theory Concepts. In *Proceedings of the 7th SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, pages 156–160.

Hazzan, O. (2003). How Students Attempt to Reduce Abstraction in the Learning of Mathematics and in the Learning of Computer Science. *Computer Science Education*, 13(2):95–122.

Heberman, B., Averbutch, H., and Ginat, D. (2005). Is it Really an Algorithm - The Need for Explicit Discourse. In *Proceedings of the 10th SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, pages 74–78.

Hoc, J. M., Green, T. R. G., and Samurcay, R. (1991). *The Psychology of Programming*. Academic Press, London.

Holland, S., Griffiths, R., and Woodman, M. (1997). Avoiding Object Misconceptions. In *Proceedings of the 28th SIGCSE technical symposium on Computer Science Education*, pages 131–134.

Holmboe, C. (1999). A Cognitive Framework for Knowledge in Informatics: The Case of Object-Orientation. In *Proceedings of the 4th SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, pages 17–20.

Holmboe, C. (2000). A Framework for Knowledge: Analysing High School Students' Understanding of Data Modelling. In *Proceedings of the 12th Workshop of the PPIG Psychology of Programming Interest Group*, pages 267–279.

Holmboe, C. (2005). *Language and Learning of Data Modelling*. PhD thesis, University of Oslo, Norway.

Holmboe, C., McIver, L., and George, C. (2001). Research Agenda for Computer Science Education. In *Proceedings of the 13th Workshop of the PPIG Psychology of Programming Interest Group*, pages 207–223.

Howe, E., Thornton, M., and Weide, W. B. (2004). Components-first approaches to

CS1/CS2: principles and practice. In *Proceedings of the 35th SIGCSE technical symposium on Computer Science Education*, pages 291–295.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 153–156.

Jeliot-Team (2006). Jeliot 3. `http://cs.joensuu.fi/jeliot/`.

Johnson, G. C. and Fuller, U. (2006). Is Bloom's Taxonomy Appropriate For Computer Science? In *Proceedings of Kolin Kolistelut-Koli Calling, 6th Finnish/Baltic Sea Conference on Computer Science Education, Koli, Finland.*

Kinnunen, P. and Malmi, L. (2004). Do Students Work Efficiently in a Group? - Problem-Based Learning Groups in Basic Programming Course. In *Proceedings of Kolin Kolistelut-Koli Calling, 4th Finnish/Baltic Sea Conference on Computer Science Education, Koli, Finland*, pages 57–66.

Koffman, B. E. and Wolz, U. (2001). *Problem Solving with Java, 2nd edition.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Kolling, M. (2006). BlueJ - The interactive Java Environment. `http://www.bluej.org/`.

Kvale, S. (1996). *InterViews: An Introduction to Qualitative Research Interviewing.* Sage publications, London.

Le Compte, M. and Preissle, J. (1984). *Ethnography and Qualitative Design in Educational Research.* Academic Press, London.

Levitin, A. and Papalaskari, M. (2002). Using puzzles in teaching algorithms. *ACM SIGCSE Bulletin*, 34(1).

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A multinational study of reading and tracing skills in novice programmers. *SIGCSE Bulletin*, 38(3):119–150.

Lister, R. and Leaney, J. (2003). Introductory programming, criterion-referencing, and Bloom. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pages 143–147.

Lister, R., Simon, B., Thompson, E., Whalley, L. J., and Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, 38(3).

Mannila, L. (2006). Progress Reports and Novices' Understanding of Program Code. In *Proceedings of Kolin Kolistelut-Koli Calling, 6th Finnish/Baltic Sea Conference on Computer Science Education, Koli, Finland.*

Marshall, D., Summers, M., and Woolnough, B. (1999). Students' Conceptions of Learning in an Engineering Context. *Higher Education*, 38(3):291–309.

Marton, F. (1981). Phenomenography - Describing Conceptions of the World Around us. *Instructional Science*, 10:177–200.

Marton, F. (1986). Phenomenography: A Research Approach to Investigating Different Understandings of Reality. *Journal of Thought*, 21:28–49.

Marton, F. and Booth, S. (1997). *Learning and Awareness.* Lawrence Erlbaum Associates, London.

Marton, F., Dall'Alba, G., and Beaty, E. (1993). Conceptions of Learning. *International Journal of Educational Research*, 19(3):277–300.

Marton, F., Runesson, U., and Tsiu, B. M. A. (2004). The Space of Learning. In Marton, F. and Tsiu, B. M. A., editors, *Classroom Discourse and the Space of Learning*, pages 3–40. Lawrence Erlbaum Associates, London.

Marton, F. and Tsiu, B. M. A. (2004). *Classroom Discourse and the Space of Learning.* Lawrence Erlbaum Associates, London.

Mayers, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, 13(1):121–141.

McIver, L. (2000). The Efffect of Programming Language on Error Rates of Novice Programmers. In *Proceedings of the 12th Workshop of the PPIG Psychology of Programming Interest Group*, pages 181–192.

Muhr, T. and Freise, S. (2003). ATLAS.ti The Knowledge Workbench, V5.0 User's Guide and Reference. `http://www.atlasti.com/downloads/atlman.pdf`.

Nguyen, H. P. and Xue, J. (2004). Strength Reduction for Loop-Invariant Types. In *Proceedings of the 27th Australasian conference on Computer Science*, pages 213 – 222.

Pang, F. M. (2003). Two Faces of Variation: On Continuity in the Phenomenographic Movement. *Scandinavian Journal of Educational Research*, 47(2):145–156.

Perrenet, J., Groote, F. J., and Kaasenbrood, E. (2005). Exploring Students' Understanding of the Concept of Algorithm: Levels of Abstraction. In *Proceedings of the 10th SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, pages 64–68.

Perrenet, J. and Kaasenbrood, E. (2006). Levels of Abstraction in Students' Understanding of the Concept of Algorithm: the Qualitative Perspective. In *Proceedings of the 11th SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education*, pages 270–274.

Piaget, J. (1954). *Construction of reality in the child.* Routledge and Kegan Paul.

Pong, W. Y. (1999). The Dynamics of Awareness. In *Proceedings of the 8th European Conference for Learning and Instruction, Göteborg University*, page 12p.

Ragonis, N. and Ben-Ari, M. (2005). A Long-Term Investigation of the Comprehension of OOP Concepts by Novices. *Computer Science Education*, 15(3):203–221.

Ramsden, P. (1992). *Learning to Teach in Higher Education.* Routledge.

Reid, N., Mancy, R., Stamouli, I., Higgins, C., and Begum, M. (2005). ExploreCSEd: Exploring skills and difficulties in programming education. In *Proceedings of the 6th Annual Conference for the Higher Education Academy Subject Network for Information Computer Science*, page 4p.

Riding, R. and Rayener, S. (1998). *Cognitive Styles and Learning Strategies.* David Fulton Publishers, London, UK.

Roberts, S. E. (1995). Loop Exits and Structured Programming: Reopening the Debate. In *Proceedings of the 26th SIGCSE symposium on Computer Science Education*, pages 268–272.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172.

Romero, P., Cox, R., DuBoulay, B., and Lutz, R. (2003). A Survey of External Representations Employed in Object-Oriented Programming Environments. *Journal of Visual Languages and Computing*, 14(5):387–419.

Säljö, R. (1982). *Learning and Understanding: A Study of Differences in Constructing Meaning From a Text.* Göteborg Stidies on Educational Sciences (41) Acta Universitatis Gothoburgensis.

Sandberg, J. (1996). Are Phenomenographic Results Reliable? In Dall'Alba, G. and Hasselgren, B., editors, *Reflections on Phenomenography: Toward a Methodology?*, pages 129–140. Göteborg Studies in Educational Sciences 109. Acta Uninersitatis Gothoburgensis.

Sandberg, J. (2000). Understanding human competence at work: an interpretive approach. *Academy of Management Journal*, 43(1):9–25.

Scott, E., Zadirov, A., Feinberg, S., and Jayakody, R. (2004). The Alignment of Software Testing Skills of IS Students with Industry Practices A South African Perspective. *Journal of Information Technology Education*, 3:1–12.

Scott, T. (2003). Bloom's taxonomy applied to testing in computer science classes. *Journal of Computing in Small Colleges*, 19(1):264–274.

Sime, M. E., Green, T. R. G., and Guest, D. J. (1973). Psychological Evaluation of Two Conditional Constructions Used in Computer Languages. *International Journal of Man-Machine Studies*, 5(1):105–113.

Soloway, E. (1986). Learning to Program - Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29(9):850–858.

Spohrer, C. J. and Soloway, E. (1986). Novice mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7):624–632.

Stamouli, I., Begum, M., and Mancy, R. (2005). ExploreCSEd: Exploring Skills and Difficulties in Programming Education. In *Proceedings of the 10th SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education (Abstract poster)*, page 371.

Stamouli, I., Doyle, E., and Huggard, M. (2004). Establishing Structured Support for Programming Students. In *Proceedings of the 34th ASEE/IEEE FIE, Frontiers in Education Conference*, page 5p.

Stamouli, I. and Huggard, M. (2006). Object Oriented Programming and Program Correctness: The Students' Perspective. In *Proceedings of the 2nd International Computing Education Research Workshop*, pages 109–118.

SunMicrosystems (2006). Java, `http://java.sun.com`.

Tall, D. (1989). New Cognitive Obstacles in a Technological Paradigm. In *Research Issues in the Learning and Teaching of Algebra, N.C.T.M.*, pages 87–92.

Ventura, P., Egert, C., and Decker, A. (2004). Ancestor Worship in CS1: On the Primacy of Arrays. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 68–72.

von Glasersfeld, E. (1995). A Constructivist Approach to Teaching. In Steffe, L. and Gale, P., editors, *Constructivism in Education*, pages 3–15. Hillsdale, NJ: Erlbaum.

Vygotsky, L. (1986). *Thought and Language.* MIT Press Cambridge, Mass.

Walker, M. H. (1998). Modules to Introduce Assertions and Loop Invariants Informally Within CS1: Experiences and Observations. *ACM SIGCSE Bulletin*, 30(2):31–35.

Weinberg, G. (1971). *Psychology of Computer Programming.* Dorset House Publishing Co., Inc. New York, USA.

Wellington, J. London, U. (2000). *Educational Research Contemporary Issues and Practical Approaches.* Continuum.

Whalley, L. J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., and 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.*

Appendix A

Empirical study

This appendix relates to Chapter 4 and it contains:

1. The initial questionnaire that was completed by all first year Computer Science students in the academic year 2004-2005 (Figures, A.1 and A.2).
2. The selection of the theoretical sample as compared to the whole class (Figure A.3).
3. The consent form that participating students signed (Figure A.4).

Figure A.1: Initial background questionnaire, page 1.

Department of Computer Science
Trinity College Dublin
Ioanna Stamouli

Michaelmas Term
2004

Name:______________________________

Gender: ☐ Female ☐ Male

Age:______

Course:_____________

Educational background:

☐ Secondary school
☐ Mature student
☐ Exchange student
☐ Transfer student, if yes please specify:______________________
☐ Other:____________________________

Have you taken any seminars, courses or certifications related to computers or computer programming?

☐ Yes ☐ No

If yes please specify:

Do you have any work experience related to computers?

☐ Yes ☐ No

If yes please specify:

Figure A.2: Initial background questionnaire, page 2.

Which of the following programming languages have you used or heard about?

I've used...	*I've heard about...*
☐ Java	☐ Java
☐ C	☐ C
☐ C++	☐ C++
☐ Visual Basic	☐ Visual Basic
☐ ASP	☐ ASP
☐ PHP	☐ PHP
☐ Perl	☐ Perl
☐ ML	☐ ML
☐ Lisp	☐ Lisp
☐ Prolog	☐ Prolog
☐ Other(s)____________________	☐ Other(s)____________________

For what purposes do you use your computer most often?

☐ Web browsing / Email
☐ Coding / Web development
☐ Games
☐ Other____________________

What are the reasons that motivated you to choose a Computer Science based degree?

What do you expect to learn in the 'Introduction to Programming' course?

Would you be interested in being interviewed a few times during this year, for a fee?

☐ Yes ☐ No

If yes, please provide your email:______________________________________

Figure A.3: Theoretical sample as compared with the class as a whole. (sample size 16, class size 40) (a) age, (b) gender, (c) educational background.

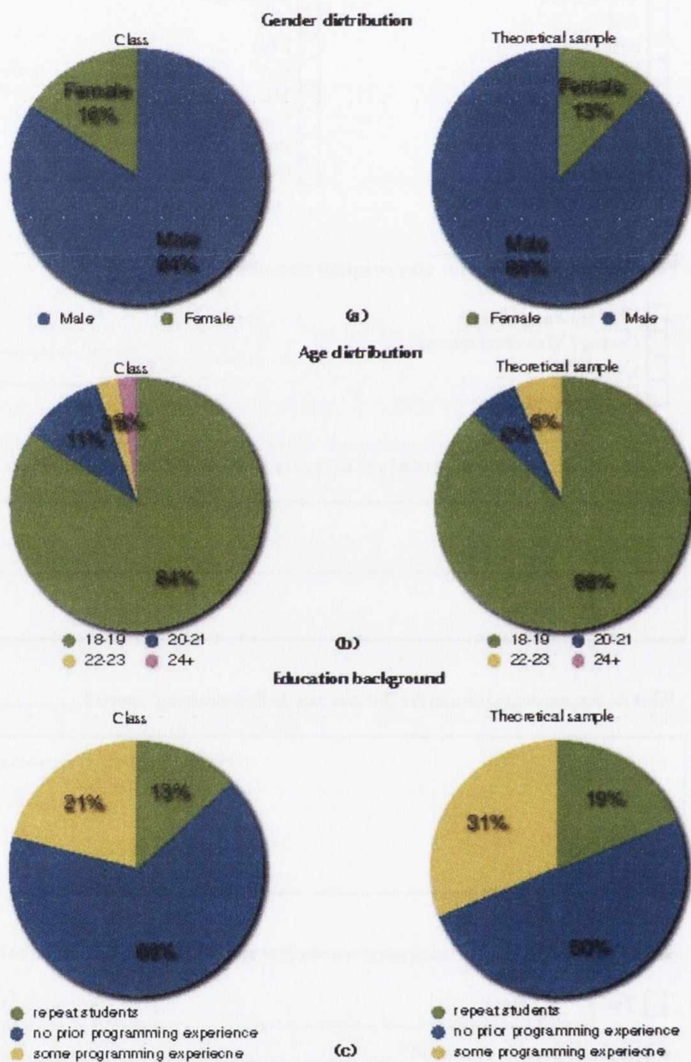


Figure A.4: Consent form signed by all participants.

Consent Form

I give my informed consent to participate in this phenomenographic study on the relationship between learning to program the object oriented way and cognitive styles.

I consent to the publication of the study results so long as the data is kept confidential, so that no identification can be made. I understand that my responses are anonymous and will be identified by number only.

1) I have been informed that my participation in this study will require my involvement in a series of semi-structured interviews were I will be answering questions.

2) I have been informed that my participation in this study will require my completing two cognitive style tests. The tests that will be involved in this study are the Group of Embedded Figures Test (GEFT) and the Felder's Learning Styles Questionnaire.

3) I have been informed that this phenomenographic study is involved in identifying the relationship between learning to program the object oriented way and cognitive styles. Thus the task of the research study is not an IQ or performance test.

4) I have been informed that there are no known expected risks or discomforts involved in my participation in this study. This judgement is passed upon a relative large body of research with people undertaking tasks of a similar nature.

5) I have been informed that participation is voluntary. I have been informed that withdrawal from the study prior to its completion will result in not receiving the agreed fees.

6) I have been informed that the researcher will gladly answer any questions about the research when the interviewing and experimental session is completed.

Concerns about any aspects of this study may be referred to Ioanna Stamouli, a PhD student in the Computer Science Department at Trinity College Dublin.

Signed: ________________________ ________________________

(Researcher) (Participant)

Appendix B

Semi-structured interviews

This appendix relates to Chapters 5, 6, and 7 and it contains:

1. The outlines of the four semi-structured interview sessions (Tables B.1, B.2, B.3 and B.4).
2. The three exercises that students completed out loud in the second and third sessions (Table B.6).

Table B.1: Outline of the first interview.

Interview No1 - Question sheet
1. Explain the study. 2. Give him/her the consent form to sign. 3. Start the Dictaphone!
General - Talk about their responses to the questionnaire. - What programming languages they have used. - What have they studied in school etc.
Motivation - Why did you choose this course? - What are your expectations from 1ba2?
Programming - What is programming for you? - What do you understand it to be?
Programmer - What would you say characterises a good programmer? - What is the approach to study that you are following?
Algorithms - From what you have done so far is there anything that interests you particularly? - Has anything proven particularly challenging? - What wound you say are the most important concepts you've met so far? - What would you say an algorithm is and how would you describe it?
Objects - What do you think of objects and classes so far? - How would you describe a class? - How would you describe an object?

Table B.2: Outline of the second interview.

Interview No2 - Question sheet
1. Start the Dictaphone!
Catch up from the last time
- What have you learnt since we last met?
- Is there anything you though was especially interesting?
- Is there anything that you have had to stop and think about?
- What was it? How did you sort it out?
Reflection
- What do you think your classmates think of Java?
- Do you think Java is hard?
- Do your classmates think Java is hard?
- Do you think they have the same problems that you have?
Object Orientation
- How do you find thinking in object and classes?
- Does this comes naturally to you?
- Is it easy (given a problem) to identify the classes you should have?
- Do you see any benefits in this approach?
Constructor
- What is a constructor?
- How do you understand it?
Problem 1
- Would you read and work out this problem? (Give problem 1)
- How is your lab work going?
- How do you usually start solving a problem?
- How do you proceed with it?
- Discuss the student system or bank system from the laboratory exercises.

Table B.3: Outline of the third interview.

Interview No3 - Question sheet
1. Start the Dictaphone!
Programming / Learning to program
I am going to repeat some of the questions we talked about in the previous interviews.
- What is programming for you?
- How would you describe it?
- What does it take to learn how to program?
- Would you say (at this point) that you know how to program?
Decision making
- What do you use when you need to make a decision about something in a program? (example the leap year problem)
- Can you explain to me how *if* statements work?
- Can you give me an example?
- Present the student with the exercise
1. Can you attempt this problem?
2. Could you please think out-loud when you do it?
Iterations
- Imagine I know nothing about loops can you explain the underlying idea for me ?
- When do you use loops?
- How many kind of loops do you know about?
- What are the condition that help you in deciding which one to use?
- Present the student with the exercise
1. Can you attempt this problem?
2. Could you please think out-loud when you do it?

Table B.4: Outline of the fourth interview (part 1).

Interview No4 - Question sheet
1. Start the Dictaphone! *Software engineering* - What do you believe software engineering is? How do you understand it? - In your understanding, what are the elements involved in it? *Objects and classes* - Assume that I know nothing about classes, objects, attributes, methods how would you describe them to me? - How do you understand the concept of a class? - How do you understand the concept of an object? - How do you understand the concept of an attribute? - How do you understand the concept of a method? *Arrays* - Assume that I know nothing about arrays, explain them to me in your own words. - Why do we need arrays? - Where are arrays used more frequently? - 2D, arrays what are they? - When and why do you use them? - Are you able to loop through a 2D array? - Why do you use the key word new when you create an array? *Correctness* - What actually is a correct program? - How can one know if a program is correct? *Inheritance* - What is inheritance? - When do you use it?

Table B.5: Outline of the fourth interview (part 2).

Abstraction encapsulation - Do you know what abstraction is? - How do you understand encapsulation? *General* - Do you feel that 1ba2 has fulfilled your expectations? - What did you find was the most helpful feature of this course? - What was the least helpful feature of this course? - Are you happy with the lecturers' presentation of the course? - Do you feel happy with the material presented? Was it too much? Did you expect more? - Did you find that you had to study the theory on your own to understand the concepts presented? - What was your approach to study? Did you had to change it as the course progressed? - What was the hardest thing for you to understand from the whole course? - Do you feel you have the same level of understanding as the rest of your classmates? - Have you developed any skills during the course? - Do you feel any of these could be transferable to other courses? - Do you feel confident in programming in Java? - Did you enjoy 1ba2 in general? - During the interviews did you find that you couldn't reply to some questions? - When you realised that you didn't really understand a concept did you go back to study it?

Table B.6: Exercises given to students to solve out loud during the interviews.

Problem 1:

You are required to design a college student record system that will allow store information about the students' previous education and current status in college. The system should also manage information about the degrees and courses provided. The lecturer's information should also be available along with the course they are teaching. The system functionality should allow:

- For a student to be transferred from one degree to another.
- A lecturer to be added.
- A course to be added.
- A lecturer to be allocated to another course.

Assumptions:

- Each lecturer can be allocated to only one course.
- Each degree can have only 4 years and maximum 10 courses per year.

Problem 2:

Write a Java application that will read two distances, one expressed in the imperial unit of inches and the other expressed in metric units (i.e. metres), and which will report the sum of the distance as yards, feet and inches and the difference between the distances as metres.

- you should allow for the constructor to take in yards, feet and inches

Hint 1 yard is 36 inches, 1 foot is 12 inches, 1 inch is 2.5 cm

Problem 3:

Write a method that report weather a person is eligible for a student's discount. A person can get a student discount if he/she has:

- a valid student id
- if his age is less than 21 and his/her parents income is no more than 20.000 euro per annum
- if he is older than 21 his income should be no more than 15.000 euro per annum
- If he is a post graduate he should be less than 26.

```
Class Student{
      String student_id;
      int age;
      String name;
      String address;
      boolean postgraduate;
      int income;
      int parent_income;
.........
}
```

Problem 4:

Using a loop (either a for or a while loop) print the following in the standard output.

```
********************
*******************
******************
*****************
****************
***************
**************
*************
************
***********
**********
```