How do they understand? Practitioner perceptions of an object-oriented program

A thesis presented in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Education (Computer Science)

at Massey University, Palmerston North,

New Zealand

Errol Lindsay Thompson

2008

Abstract

In the computer science community, there is considerable debate about the appropriate sequence for introducing object-oriented concepts to novice programmers. Research into novice programming has struggled to identify the critical aspects that would provide a consistently successful approach to teaching introductory object-oriented programming. Starting from the premise that the conceptions of a task determine the type of output from the task, assisting novice programmers to become aware of what the required output should be, may lay a foundation for improving learning. This study adopted a phenomenographic approach. Thirty one practitioners were interviewed about the ways in which they experience object-oriented programming and categories of description and critical aspects were identified. These critical aspects were then used to examine the spaces of learning provided in twenty introductory textbooks. The study uncovered critical aspects that related to the way that practitioners expressed their understanding of an object-oriented program and the influences on their approach to designing programs. The study of the textbooks revealed a large variability in the cover of these critical aspects.

Acknowledgements

A large number of people have had some influence on this work either directly or indirectly. The journey had its origins in my transfer from an industry position back into academia in 1991. An early influence on my thinking with respect to teaching was my students at Unitec Institute of Technology in Auckland who helped me revise my approach to teaching programming. Their influence on my thinking cannot be ignored. At the same time, I assisted a retired minister and theological researcher. I am indebted to the late Rev. Harold Turner for his discussions on theology, epistemology and introducing me to the writings of Michael Polanyi. His input made a profound difference in my approach to teaching and this research.

The journey of this research has seen a number of people contribute in different ways. The first of these are those that have supervised this work along the way and provided valuable feedback and suggestions.

Professor Kinshuk was my first contact when I began to explore the possibility of completing a PhD through Massey University. He has advised on choice of primary supervisors and remained an assistant supervisor for the duration of the research. He has provided unfailing friendship and support for the journey.

Dr Lynn Hunt provided the initial supervision of this research and led me to explore educational literature especially related to metacognition, and the writings of John Biggs, Paul Ramsden and Noel Entwistle. Unwittingly and possibly against her better judgement this lead me to phenomenography and the writings of Ference Marton and Shirley Booth. Her questioning of my research process led me to explore issues of research methodology and ultimately to the writings of Thomas Kuhn. It is a pity that we were unable to see this journey to its end together.

I appreciate the efforts of Associate Professor Janet Davies efforts in picking up the supervision when the direction was well established and her efforts in helping me clarify my thinking with respect to the research method and the educational content. She has provided much needed encouragement and support. Although she retired before the thesis was submitted, she has provided valuable input that has helped see this journey to its end.

The research students in the Advanced Learning Technologies Research Centre for their friendship and sharing. Their belief in my ability to give something to their research despite my own struggles has been an encouragement on this journey.

My students at Massey University who put up with my ideas and reflections, and nearing the end of this project my strict hours of availability so I could concentrate on writing. I really appreciated those who recognised that they could learn from me and not just pass the courses or encouraged me to pursue some of my teaching strategies. I thank them for their feedback on the way that I was teaching and on what they were learning from me.

I thank Professor Erkki Sutinen, the students and staff of Joensuu University who provided a safe environment for three months in which I could do analysis, and write.

For my wife, Marilyn, who has provided undying support and help regardless of the struggles. She became the initial proof reader of all that I wrote and has ensured that food has been on the table over the last nine months as I have worked exclusively on completing this project.

I particularly want to thank all the participants who gave me at least an hour of their time while I interviewed them about their understanding of object-oriented programming. Without their input there would have been no thesis. Their openness has challenged my own thinking on the themes of this thesis and provided challenges for further work. There are numerous others with whom my research was discussed and who have offered encouragement and support along the way. All in their own way have influenced the work and enabled me to complete this journey.

Ethics Committee Approval Statement

This project has been reviewed and approved by the Massey University Human Ethics Committee, Wellington Application 05/69. If you have any concerns about the conduct of this research, please contact Professor Sylvia Rumball, Chair, Massey University Campus Human Ethics Committee: Wellington, telephone +64 6 350 5249, email humanethicswn@massey.ac.nz

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

In the computer science community, the learning of programming is perceived as difficult and current teaching strategies are not overcoming this perception. There is considerable debate about the appropriate sequence for introducing object-oriented concepts to novice programmers. Research into novice programming has struggled to identify aspects that would provide a consistently successful approach to teaching introductory programming both procedural and object-oriented. Research in relation to expertise highlights the differences between the ways that novice and expert programmers approach the comprehension of program code. Chapter 2 of this thesis examines the research with respect to learning to program and the understanding of how novice and expert programmers comprehend programs.

As a lecturer teaching programming at a number of tertiary institutions, the researcher had experienced some success in the teaching of procedural programming using a strategy that focused on laying a conceptual foundation of the nature of a procedural program and of the programming task. Having moved to the teaching of object-oriented programming and not achieved the same level of success, the researcher sought to understand the conceptual differences between a procedural approach and an object-oriented approach to programming. Coming from a computer science focus, the researcher began to explore literature relating to tertiary learning to understand how to improve his approach to teaching and fostering learning.

This exploration of tertiary learning literature led to literature that emphasised the impact of task representation on the approach to learning and the learning outcomes. A number of factors, including the teaching context, influence the learner's task representation. This raised the question of what task representations were effective for the learning of object-oriented programming. Phenomenographic research has been used by other researchers to uncover the different conceptions that learners have of learning and of topics being learnt. The outcome of phenomenographic studies are represented in outcome spaces containing an ordered hierarchy of categories of description.

Starting from the premise that the conceptions of a task determine the type of output from the task, assisting novice programmers to become aware of the nature of the required output may lay a foundation for improving learning. The hypothesis behind the

research for this thesis is that if the critical aspects for the categories of description for a phenomenon as recognised by practitioners in the field[1] can be identified, then these can be used as the basis for planning teaching. It will be argued that learning involves a change of conception and that to achieve a change of conception a space of learning has to be created based on variations in critical aspects. This understanding of tertiary learning is discussed in chapter 3.

In order to uncover the conceptions of the object-oriented programming task, a phenomenographic approach is used to uncover practitioners' awareness of an object-oriented program. To achieve the objective of the hypothesis, it was planned to use a teaching strategy that would open up a space of learning based on the results of the practitioners' awareness. Due to changing institutional circumstances, this was not able to be achieved so introductory programming textbooks were examined to determine the spaces of learning that they opened up.

The first part of the research plan involved conducting interviews with a broad range of practitioners with varying degrees of experience in the use of object-oriented programming. The focus of the interviews was on how the practitioners described their awareness of an object-oriented program and the way that they approached the task of writing an object-oriented program. These interviews were analysed to uncover the categories of description present and the critical aspects that provided the distinction between the categories. Having identified the critical aspects, the second part of the research was an analysis of the textbooks to determine how they placed emphasis on these critical aspects. Chapter 4 describes the research strategies utilised.

The study uncovered two outcome spaces that relate to the way that practitioners expressed their understanding of an object-oriented program and the influences on their approach to designing programs. For each of the outcome spaces, the critical aspects

[1] A practitioner in the field is a person who is using object-oriented techniques in their work. They may be developing software, teaching object-oriented programming, or engaged in research. These are people who may claim to have a sound understanding of the core concepts and how they are applied in the development of software.

were identified that provided the variations between the categories of description. The outcome spaces and their critical aspects are discussed in chapter 5.

The second set of results endeavours to identify how the critical aspects are covered in the introductory programming textbooks. This showed that the critical aspects are present in the textbooks but are covered in an inconsistent manner. The analysis of the textbooks revealed a large variability in the cover of these critical aspects. Chapter 6 describes the results of the textbook analysis.

The results of the participants' interview analysis and the textbook analysis are re-examined in the light of the literature on learning and programming to uncover the impact that the critical aspects have in relation to other research. It also highlights the issues that remain for further investigation. This examination revealed that the critical aspects are present in the literature but that some of the variations in the critical aspects are not covered. Chapter 7 discusses these issues revealing where the results of the current study parallel issues covered in the literature and where it differs.

Based on the findings and issues raised and the two outcome spaces identified, recommendations are made for academics and textbook authors. A proposed strategy for further research is outlined with specific emphasis on an approach to evaluating a teaching strategy. The outcome spaces also raise issues beyond learning and some of these are also discussed. Chapter 8 provides this forward looking summary.

Chapter 2. Problems of learning to program

Weinberg (1992) makes an interesting statement on the nature of programming in the preface of his book on quality software management when he says:

> "When I didn't think right about a program, the program bombed. The computer, I learned, was a *mirror* of my intelligence, and I wasn't too impressed by my reflection"

He contends that a program is a 'mirror' on the intelligence of the programmer. If we are to accept Weinberg's comment as being realistic about the nature of programming, then we need to understand how to train the programmer so that they understand the intellectual issues that are involved in programming. Weinberg's statement does not give us any insight into how he as a programmer understood what a program was or what he conceived were the intellectual characteristics that would be reflected in a good program.

A question that needs to be asked with respect to literature on programming is whether these are the views of practitioners or whether they are based on research. In this thesis, the emphasis is placed on research but there will also be frequent references to practitioner writings. In part this is because the focus of the writing is quite different. Practitioners may not have completed a detailed data gathering exercise to justify their point of view but they do relate their experience of the field. Building a picture of the software development environment without heeding the practitioner reports and reflections would ignore a large volume of literature on the subject and would also ignore the differences that occur between the academic pursuit of understanding and the practitioners' understanding developed through the practical graft at the coal face.

Starting with exploring the nature of programming, this chapter examines research into programming and learning to program. The issues raised are examined in the context of how they relate to object-oriented programming.

2.1 The nature of programming

The objective of computer programming is to create a program that is executable by a computer and performs a required task (Bronson, 2006, p 7; Guzdial & Ericson, 2005, p 6; Lewis & Loftus, 2006, p 25). Although this objective tells us the focus of the programming task, it tells us little about what is involved in the writing of a program.

Guzdial and Ericson (2005) provide some additional insight when they say

> "A *program* is a description of a process in a particular programming language that achieves some result that is useful to someone. … An *algorithm* (in contrast) is a description of a process apart from any programming language. The same algorithm might be implemented in many different languages in many different ways in many different programs - but it would all be the same *process* if we're talking about the same algorithm" (p 6)

These authors contend that a program is language dependent where an algorithm is language independent. The programming task would therefore appear to be the translation of a generic algorithm to a specific programming language so that it can be executed on a computer and achieve something useful for a user. In this section, the nature of a program as it is described in programming literature will be explored.

2.1.1 The programming paradigms

Object-oriented programming is described as a programming paradigm. A programming paradigm is defined in a number of different ways. Ambler, Burnett, and Zimmerman (1992) describe a programming paradigm as "a collection of conceptual patterns that together mold the design process and ultimately determine a program's structure" (p 28). In contrast, Stolin and Hazzan (2007) initially talk about paradigms in a generic sense that reflects the concept of a paradigm as stated by Kuhn (1996). Stolin and Hazzan initially talk about programming paradigms as "a way of doing and seeing things, a framework of thought in which we interpret one's world" (p 65). After acknowledging that references define programming paradigms in different ways, they

finally conclude that for their investigation into how the concept of a programming paradigm is understood, they will use the definition

> "Programming paradigms are heuristics used for algorithmic problem solving. A programming paradigm formulates a solution for a given problem by breaking the solution down to specific building blocks and defining the relationship among them" (p 65).

A programming paradigm would appear to be a way of thinking and constructing software solutions. Each paradigm will bring its own tools and techniques and its own way of thinking through how to construct software.

The influence of the programming paradigm on the software development process is also the emphasis of Brookshear's (2007) definition. He contends that the programming paradigms are really software development paradigms since the alternative paradigms "have ramifications beyond the programming process" (p 273). He argues that the entire software development process is impacted since programming paradigms are "fundamentally different approaches to building solutions to problems" (p 273).

In contrast, Von Roy and Haridi (2004) argue for computational models that provide a "set of techniques and reasoning about programs" (p xiii). They see the models as having deep relationships which allow one model to be built by adding new concepts to a previous model. Von Roy and Haridi appear to be arguing for commonality among the paradigms rather than clear differentiation.

The idea of one paradigm building on another is promoted by Felleisen (2005) who claims that by teaching students the functional paradigm first the students are better prepared

> "for the true essence of object-oriented programming according to Alan Kay: the systematic construction of small modules of code and the construction of programs without assignment statements" (p 1).

Becker (2007), in discussing why he wrote his own textbook, says that the textbooks he examined changed the programming language used in teaching programming and possibly the programming paradigm (imperative (Pascal) to object-oriented (Java)) but

had retained the same approach to teaching. He says of these textbooks in his preface that “the programming language had changed from Pascal to Java, but the approach had not” (p xv). He then expresses his view that “A second change was necessary: a change in pedagogy” (p xv). This is further discussed in “Object-Oriented Pedagogies” (p xx) where he talks about the combinations of writing and using objects. He argues that these are ‘write and use’, ‘write, then use’, and ‘use, then write’. Although he says that his textbook uses a ‘use, then write’ pedagogy, he does not clarify the issue of the impact of a paradigm on the approach to teaching programming or whether there is any fundamentally different way of thinking about programming brought about by a change in paradigm.

Ambler, Burnett, and Zimmerman (1992) group the programming paradigms into two groups, operational and definitional. This split is based on paradigms that are designed to tell the computer how to solve the problem (operational) versus those that describe the problem and use some form of problem solving engine to arrive a solution (definitional). Within each of these groups, they then go on to describe a number of different programming paradigms. For example, in the operational category, they include imperative, object-oriented, and functional (operational), and in the definitional category, they include functional (definitional), logic, transformational, and constraint. The implication is that by being able to group the paradigms, there are some common features despite some of the concepts being applied differing from each other.

The imperative programming paradigm is regarded as the traditional approach to programming and is based on providing a sequence of instructions (Brookshear, 2007). The declarative programming paradigm is based on describing the problem to be solved. This description is used by a “preestablished general-purpose problem-solving algorithm” (p 273). The functional paradigm is based around the notion of a function that receives inputs and produces an output. For the object-oriented paradigm, a program is built around a collection of units called objects.

Farrell’s (2006a) argument that object-oriented programming is an extension of procedural programming is based on the view that there is only a “slightly different approach to writing computer programs” required (p 4). However, she also says that if objects are learnt first then it will avoid unlearning procedural logic in order “to write in an object-oriented fashion” (2006b, p xv). This perspective is reflected by Wirth (2007)

in reviewing the history that led to the development of Oberon. Wirth argues that, in Oberon's type extension and type inclusion, "object-oriented programming is effectively a style based on (inheriting) conventional procedural programming" (p 3-8). An underlying assumption is that the additional feature of object-oriented programming is abstract data types. This view is also expressed by Burton and Bruhn (2003) when they say "while OOP undeniably represents a new paradigm, it in no way replaces the older paradigm that preceded it; rather, it is in addition to it" (p 111). They argue, without supporting evidence, that the writing of sensible programs in OOP requires the programmer to "be highly proficient with Procedural Programming" (p 112). However, Bronson (2006) seems to be of the view that the object-oriented paradigm is significantly different. He argues that a "noticeable switch, both in thinking and coding" is required "when a class-centered approach is introduced" (p xiii). Hu (2005; 2008) argues that abstract data types are the essence of object-oriented programming and that teaching the use of objects without data is harmful. Vilner, Zur, & Gal-Ezer (2007) support the abstract data types view when they discuss the fundamental concepts of CS1[1]. They describe questions that they used to compare the understanding of a group trained in procedural programming against a group trained in object-oriented programming. Their questions show that they expect similar performance from both groups and a use of classes to implement abstract data types. Because the course materials are not being examined, it is difficult to determine the exact focus of the teaching but there is a strong indication that they believe the fundamental concepts remain unchanged regardless of the paradigm used for teaching.

Berglund and Lister (2007), in conducting a study on teachers understanding of object-first approaches to teaching programming, present a preliminary finding that there are two conceptions of the *objects-first* debate. These are that object-oriented programming is an extension of imperative programming, and that it is conceptually different from imperative programming. In looking at the dimensions of variation, they contend that with respect to seeing object-oriented as an extension of imperative programming, the teachers saw objects as passive entities that are used in programs, polymorphism was seen as different objects, and modification as change in code. Those holding the view

[1] CS1 is the introductory computer science course that usually introduces programming.

that it was conceptually different from imperative programming saw objects as active entities with object interactions providing the algorithm, polymorphism is seen in the interaction between objects, and modifications as “adding to holes and hooks”. These two conceptions provide fundamentally different understandings of the role of objects in programming.

There is another view that seems to have little recognition and this is a view based on multi-paradigm languages (Coplien, 2000). Coplien defines this as:

> “A paradigm, as the term is popularly used in contemporary software design, is a way of organizing system abstractions around properties of commonality and variation. The object paradigm organizes systems around abstractions based on commonality in structure and behavior and variation in structure and algorithm” (p xvii)

He contends that Bjarne Stroustrup, the creator of C++, designed the language to support multiple paradigms. It could be argued that a programming language such as Java is a multi-paradigm language as it supports both object-oriented and imperative styles of programming. Coplien bases his argument around commonality and variability analysis as a basis for finding appropriate abstractions. An emphasis on multiple programming paradigms would add yet another dimension to the debate of where to start teaching programming.

Views on the relationship between the programming paradigms and their influence on approaches to teaching are strongly held. Lister, Berglund, Clear, Bergin, Garvin-Doxas *et al.* (2006a) report, they could find no “evidence of literature studying converse paradigm shift, from learning objects-first to learning imperative programming” (p 149). The confusion over how different the object-oriented paradigm is from imperative paradigm and which should be taught first has fuelled debate within the Computer Science Education community (Lister et al., 2006a). The primary focus of this debate is the relationship between the imperative paradigm and the object-oriented paradigm but as Felleisen’s (2005) perspective shows, there are alternative views based on the other paradigms. The domination of the debate around imperative versus object-oriented is reflected in the literature examined in this chapter.

2.1.2 The nature of a program

The foundation for imperative programming is expressed by Wirth (1976) as "algorithms + data structures = programs". Wirth presents fundamental data structures and the algorithms used to process them. The focus on algorithm remains quite strong and is reflected in the literature where the nature of a program is discussed.

Guzdial and Ericson (2005), as quoted above (see section 2.1.1, page 6), contend that an algorithm is language neutral and is like a recipe in nature (p 3). They see the study of algorithms as an important part of what computer scientists do. They provide no definition of what an algorithm is or how any particular algorithm should be described. Lewis and Loftus (2006) are more explicit, clearly defining an algorithm in terms of "a step-by-step process for solving a problem" (p 324). They do acknowledge that there are different ways to implement algorithms (p 504).

It would seem legitimate to question the nature of an algorithm and its ease of implementation when confronted with a strategy for implementation using an object-oriented approach and the representation of algorithms as design patterns[2] (Nguyen, Ricken, & Wong, 2005). Design patterns with the focus on object relationships appear to be an alternative way of expressing an algorithm that is not step-by-step in nature. Gamma, Helm, Johnson, & Vlissides (1995) argue that expert designers "reuse solutions that have worked before in the past" (p 1). Design patterns represent such reusable solutions (Alexander, Ishikawa, & Silverstein, 1977, p x; Gamma et al., 1995, p 2). For Gamma et al., design patterns are "*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*" (p 3). The pattern needs to be adapted to be implemented in the design and language of the project (p 3).

Clancy and Linn (1999) have argued that "textbook writers and course designers need to create design patterns that students find accessible and can connect to new problems" (p 40). They argue this on the basis of using them to scaffold learning as proposed by

[2] Design patterns represent outline solutions used to solve common program design problems. They normally involve multiple objects and the interactions between them.

Rosson and Carroll (1996). The patterns that Clancy and Linn seek to have used are more like Beck's elementary patterns[3] (1997; 2007) rather than the more complex design patterns documented by Gamma et al. (1995).

Returning to the examination of algorithms, Brookshear (2007) states that "an algorithm is abstract and distinct from its representation. A single algorithm can be represented in many ways" (p 214). How does such an understanding of algorithms differ from the ideas of elementary patterns or implementation patterns (Beck, 1997, 2007)? Beck (2007) argues that "patterns are based on commonality" (p 5). They are things that programmers repeatedly do. In the case of implementation patterns, they are applied "every few seconds" (p 2). Design patterns may be used a few times a day (p 2). Beck's implementation patterns are more low level than algorithms, and algorithms are not seen as design elements. There is a commonality in that they express common ways of solving programming problems that are independent of the programming language although not necessarily of the programming paradigm. The algorithm is regarded as a major area of study in computer science and is discussed more generally as being paradigm independent but the roots of an algorithm are in a set of 'step-by-step' instructions that is the primary focus of the procedural paradigm.

An algorithm, as Guzdial and Ericson (2005) argue, needs to be implemented in a programming language. Lewis and Loftus contend that

> "A program is written in a particular *programming language* that uses specific words and symbols to express the problem solution. A programming language defines a set of rules that determines exactly how a programmer can combine the words and symbols of the language into *programming statements*, which are the instructions that are carried out when the program is executed" (p 25).

[3] An elementary pattern represents a common coding practice used to produce quality code. An example is the complete constructor that ensures that all the properties of the object are set when the object is created.

The nature of a program, both in terms of the language constructs used and the algorithms / patterns to be implemented, is a key aspect in learning to program. The algorithms and patterns have to be refined and shaped for the particular task in hand and possibly to the programming paradigm and language. A programming language has a structure defined by its syntax rules, and the statements of the language carry meaning that relates to what the statement will do. Without knowledge of the syntax of a language, a student will struggle to write even the simplest of programs. This is evidenced by reported findings that the major cause of problems for which students seek help are basic mechanics (i.e. syntax related problems) (Garner, Haden, & Robins, 2005; Hanks, 2007; Robins, Haden, & Garner, 2006).

A program is constructed in a particular programming language implementing algorithms or patterns that solve a particular programming problem.

2.1.3 The nature of the programming task

Problem solving is seen as one aspect of the programming task (Becker, 2007; Bronson, 2006; Lewis & Loftus, 2006). It may not be expressed as problem solving but rather as some form of reasoning about programming or the problem domain (Barker, 2005; Felleisen, Flatt, Findler, Gray, Krishnamurthi *et al.*, 2001). In problem solving, the programmer may be seen as developing a model of the problem domain (Barker, 2005; Wiener, 2007). This idea of modelling the domain is also a key component of model-driven architecture (Kleppe, Warmer, & Bast, 2003; Mellor, Clark, & Futagami, 2003). Evans (2003) talks about the model as a ubiquitous language, that enables a common understanding between the developer and the client. The developer endeavours to capture the problem domain using terminology from the problem domain and in a manner that enables the client to understand the model and confirm that the developer is producing the type of system that is required.

Robillard (1999) argues that the software development process is knowledge-intensive. Robillard explores cognitive science perspectives of knowledge processing and discusses the gap between the practices of software scientists and practitioners. He discusses five key knowledge concepts that relate to software development. These are procedural versus declarative knowledge, knowledge structure (schema), formal knowledge representation (proposition), chunking, and making knowledge structures.

Armour (2000a) reflects a knowledge acquisition theme when he argues that software is not a product but rather a medium. He contends that it is simply "the most recent in a series of media that exist to store knowledge" (p 19). Software is different from other forms of knowledge media in that you can execute it. Based on this perspective, he argues that software development "is a knowledge-acquiring activity" (p 21). He contends that many of the software structures such as representational models, "are required by the people" and not the software, machine, or even the knowledge (p 22). He contends that the software development workplace should be structured for learning and not for the building of products (p 22). He further contends that in training developers, the focus should first be on the "domain in which the knowledge operates" then on "the ability to acquire knowledge", and finally on the mechanisms for knowledge representation (p 22). The software development process, he contends, should be treated as a knowledge acquisition process (p 22). These views have parallels in the computational literacy approach (diSessa, 2000) and in Guzdial and Ericson's media computation (2005). This view leads Armour to argue for the "closing of the learning application gap" (2003a) and for five laws of ignorance (2000b; 2003b).

Abstraction is argued to be a key skill in computing or assumed to be part of the process of building software (Bennedsen & Caspersen, 2006; Hazzan, 2008; Kramer, 2007; Pennington, Lee, & Rehder, 1995). Kramer (2007), in raising the issue of the importance of abstraction to computer science, argues that there are two relevant aspects of abstraction. He contends that these are the removal of unnecessary detail, and the formulating of general concepts. In terms of object-oriented programming, defining a class with its relevant attributes and methods or using a hierarchy of classes is interpreted as abstraction because it focuses on the removal of unnecessary details to form a definition for an object or hierarchy of object definitions that are relevant to the problem domain (Or-Bach & Lavy, 2004). An object in an object-oriented program can be used to represent a physical thing or concept (Sprague & Schahczenski, 2002).

For some, the emphasis is placed on abstraction being part of the process of modelling the real world (i.e. the model is an abstraction of the real world) (Barker, 2005). For others, abstraction is part of the process of defining classes and inheritance hierarchies (Felleisen, 2005). The programmer in the object-oriented context creates an abstraction of the problem domain. Guzdial and Ericson (2005) contend that abstraction is not the

place to start when teaching students to program (p xxiv). They base their argument on the work of Fleury (2001) where student's reluctance to encapsulate is taken as meaning a difficulty with applying abstractions. However, as Fleury points out, this may be more due to students not being introduced to good object-oriented design from the start.

Hazzan (2002; 2003) takes a different approach to defining levels of abstraction in his research into how students endeavour to reduce the level of abstraction in learning computational theory. He defines three interpretations with respect to the level of abstraction and students approaches to reducing abstraction. The first of these interpretations relates to the quality of the relationship between the object of thought and the thinking person. Different people will have different levels of abstraction in the sense that they have formed a different number of connections with the object. The higher the number of connections or the more detail in their understanding of the object then the more concrete and less abstract is their understanding. In this interpretation of abstraction, a person reduces the level of abstraction by endeavouring to develop a more detailed understanding of the object. This closely relates to the notion of an object in an object-oriented program having some relationship to a real world entity. The second interpretation of abstraction sees the level of abstraction as a reflection of the process-object duality. If an algorithm is seen as being implemented in a specific language and in a specific way, then the algorithm is seen as less abstract than it would be if the algorithm was seen as a generic description to solving particular styles of problems. When a process or algorithm is viewed as an entity that can be studied and analysed then it is regarded as an object whereas when simply seen as something that is applied then it is seen as a process and is less abstract. Within the object-oriented context, design patterns may be seen from a process perspective as solutions that are implemented to solve specific problems or as objects that can be adapted and explored to address a range of possible problems. The third interpretation of the levels of abstraction sees the level of abstraction as the degree of complexity of the concept of thought. The lower level of abstraction endeavours to work with simpler solution strategies in order to avoid the compound possible solutions that might exist. In the case of object-oriented programming, an individual may focus on a specific language rather than the paradigm concepts which may require an understanding of a set of languages.

The nature of the programming task can be described as a problem solving process that captures knowledge about the problem domain. This knowledge may be seen in the form of a domain model that is an abstraction of the real domain. The novice programmer in learning to program has to learn these skills of abstraction along with knowledge of the programming language, its syntax and semantics. The model will be represented in a way that is consistent with a particular programming paradigm and will implement some form of algorithm.

2.2 Learning to program

The previous section dealt with the nature of the programming task. It gave some idea of the issues that the novice programmer faces in learning to program. This section reviews the literature on learning to program and the difficulties perceived in teaching and learning of programming skills.

2.2.1 Perceptions of learning to program

Programming is generally perceived as difficult to learn with considerable debate as to the appropriate teaching strategies and starting paradigm (Lister, Adams, Fitzgerald, Fone, Hamer *et al.*, 2004a; Rountree, Rountree, & Robins, 2002). It is argued that computer programming involves understanding of:

1. what programs are for and what they can do (general *orientation* (du Boulay, 1989));
2. how the computer executes programs (the *notional machine* (du Boulay, 1989));
3. the structure of programming language constructs and the meaning of those language constructs (*notation* (du Boulay, 1989));
4. the algorithms and patterns that can be used to solve programming problems within a particular programming paradigm (*structures*, *schemas* or *plans* (Bergin, 2003; du Boulay, 1989));
5. the skills of planning, organizing, managing, developing, testing, and debugging (pragmatics (du Boulay, 1989; Upchurch & Sims-Knight, 1998));
6. the thinking frameworks that surround the use of those constructs (mental representations (Burkhardt, Détienne, & Wiedenbeck, 1997; Corritore & Wiedenbeck, 1999));

7. the domain knowledge being captured in the program (Armour, 2000b, 2003b; Evans, 2003; Robillard, 1999); and
8. the development of, or revision of, development processes for new product situations (Armour, 2000b, 2003b).

A wide range of different understandings are reflected in this list. These understandings vary from a technical focus, such as understanding how the computer executes a program, to a process focus, such as the skills of planning. The diversity of knowledge and skills adds to the difficulty of the task.

2.2.2 Effective and ineffective novice

Despite the difficulties, some learners quickly learn the key concepts and complete assigned programming exercises while others struggle to complete programming exercises (Thompson, Hunt, & Kinshuk, 2006a). Robins, Rountree, & Rountree (2003) suggest that this difference in performance can be attributed to the distinction between *effective* and *ineffective* novices. They contend that an effective novice learns to program "without excessive effort or assistance" and that ineffective novices do not learn to program or that they do so "after inordinate effort and personal attention" (p 165). They acknowledge that there is a range of potentially relevant factors including prior knowledge, motivation, confidence, etc but that the most significant factor is the strategies used by novices to assess and apply the knowledge of the programming language to program comprehension and generation.

Davies (1993) supports this emphasis on strategies with the contention that, for the novice programmer, success in programming may be related more to the strategic elements of programming skill rather than the knowledge-based components. Perkins, Hancock, Hobbs, Martin, & Simmons (1989) contend that key strategies are neglected by unsuccessful novice programmers. They also argue that those who do succeed do so because they bring with them "characteristics that make them good bootstrap learners in the programming context" (pp 277-278). This is supported by Sheil's (1981) observation that a novice's performance is "largely a function of how well they can bring their skills from other areas to bear" on programming problems (p 119).

These conclusions are based on the differences in novice programmers on a continuum between ineffective and effective novices, and not the distinction between novices and experts. The novice-expert divide reinforces these distinctions (Hoc, Green, Samurçay, & Gilmore, 1990; Sims-Knight & Upchurch, 1998; Soloway & Sphorer, 1989). The novice expert distinction is consistent with research on expertise in other fields (Chi, Glaser, & Farr, 1988). A key distinction between a novice programmer and an expert programmer is that the expert programmer recognises patterns in the code rather than the line-by-line details. They have a conceptual understanding of what they expect to see in a program. An expert programmer will outperform a novice programmer when faced with well-written code, code that conforms to the familiar patterns, but a novice may outperform the experts when the tasks faced are unnatural (Adelson, 1984). “The explanation being that when code is well-written and the tasks are natural, experts are able to reason about code at a more abstract level than novices” (Lister, 2007).

The emphasis on strategies in the study of novice programmers is primarily related to the skills required to plan, model, code, test, debug, and repair programs (du Boulay, 1989; Upchurch & Sims-Knight, 1998). They are process-oriented strategies that are influenced by the perception that the programmer has of the particular software development task. Learning is central to these process-oriented strategies since they must come to understand the problem domain and the medium in which the knowledge is to be captured (Armour, 2003b). The novice’s perception may not recognise the learning nature of the task but rather see it as the application of defined techniques and code structures. The key factor in determining an expert from a novice is the patterns or perceptions that they have about code and what the code is likely to be doing. This is based on code reading ability rather than code writing ability.

2.2.3 Naturalness of Object-oriented programming

Neubauer and Strong (2002) contend that the object-oriented paradigm concepts are not more natural or familiar for novice programmers than imperative paradigm concepts. They say that they “assume that "more natural" means more intuitive -- more easily understood and more consistent with existing patterns of thought” (p 280). Their argument is based on the number of things around us that are based on imperative style instructions. They acknowledge that there are things that are more object-oriented in character but that this is not the dominant pattern of thought in relation to those things.

They propose the use of a football game as a possible analogy for object-oriented systems.

Neubauer and Strong's view is based on their own understanding of how these things are seen or understood by the population at large. It is unclear what exactly they see as being object-oriented. There is an assumption that this terminology has a consistent meaning for their readers.

The researcher's own experience of teaching imperative programming would suggest that novice programmers do not find even the imperative paradigm natural until they are introduced to the notion of what a computer program is through drawing on their awareness of programmes[4] that they use every day (Thompson, 1992). Neubauer and Strong (2002) propose that introducing the object-oriented paradigm using familiar non-computer experiences which have the characteristics desired for an object-oriented program, may lead the learners to a better understanding.

Détienne (1997) endeavoured to evaluate claims that object-oriented programming is natural and easy to use. Based on studies of object-oriented (OO) novice designers, she found that the empirical evidence did not support this claim. She says, "previous experience in the procedural paradigm causes interference when learning an OOP language" (p 51). Her view is based on novice OO designers' approach to defining methods "both following their execution order (i.e. a procedure-centred strategy) and according to their functional similarity (i.e. a function-centred strategy), regardless of the objects to which they are associated" (p 52). Her understanding of object-oriented paradigm is:

> "A key difference between the object-oriented paradigm and the procedural paradigm is that in the procedural paradigm data and functions are separated, whereas in the object-oriented paradigm they are integrated. Objects are program entities which integrate a structure defined by a type

[4] The UK English spelling of programme is used to refer to programmes that are not computer programs. The American spelling is used for computer programs since this is the dominant convention in the computing community.

and functionalities. The concept of class integrates both the structure and behaviour of objects" (p 48).

The novice object-oriented designers did not seem to grasp the notion of bringing together the data and functionality. The difference in plans between the imperative paradigm and the object-oriented paradigm makes it difficult for novices to transfer prior programming knowledge to the new paradigm (p 52). As Détienne says, further studies with 'real' novices (those with no prior programming experience) may reveal different findings (p 51). These studies would seem to support the view that learning object-oriented programming after imperative programming is difficult since the imperative strategies are not consistent with those used in object-oriented programming.

Wiedenbeck, Ramalingam, Corritore, and Sarasamma (Corritore & Wiedenbeck, 1999, 2001; Wiedenbeck & Ramalingam, 1999; Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999) have completed a series of studies endeavouring to compare mental representations developed by object-oriented programmers and procedural programmers in relation to object-oriented programs. Two of these studies included novice programmers (Wiedenbeck & Ramalingam, 1999; Wiedenbeck et al., 1999). The framework for determining the mental representations drew on a proposed framework developed by Pennington (1987a; 1987b). Pennington's framework had been used for evaluating imperative programmer mental representations of programs. The categories used are:

1. "Elementary operations asked whether a certain specific operation occurred in the program, e.g., 'Is variable temp initialized to 0?' These questions corresponded to a single statement of the source code.
2. Control flow questions asked about the temporal ordering of actions in the program during execution, e.g., 'Is variable X assigned the value 0 before variable Y is assigned the value 0?' Questions were included which asked about control flow within a module and between modules. […] Both kinds of control flow occur in small programs which are written with functions.
3. Data flow questions asked about the relationship or dependencies of variables and variable transformations, e.g., 'Does the value of variable X affect the value of variable Y?' These questions all involved variable-to-variable data flow in

which one variable may modify the value of another variable either directly by assignment or indirectly by controlling a conditional expression.

4. Function questions asked about the principal functions carried out by the program, e.g., 'Does the program calculate the average of a group of inputs?'" (Wiedenbeck et al., 1999, p 264)

These are further reduced to a domain model and a program model where the domain model is made up of the function and data flow, and the program model of the operations and control flow (Corritore & Wiedenbeck, 1999; Pennington, 1987a, 1987b; Wiedenbeck et al., 1999). The control flow and data flow questions assume a sequential nature to the program. At a simple object-oriented program level these categories may have some meaning but the more that inheritance and polymorphism is used in the program, the more difficult it would be to determine whether these flows actually occur. This may be reflected in the results reported when Wiedenbeck et al.(1999) say "The scores of the OO subjects on function and data flow questions making up the domain model were very low, around 55% correct" (p 274). They contend that the procedural programmers performed better in all categories when working with large procedural programs. This result may be more a reflection that these types of questions have more meaning for a procedural program than for an object-oriented program rather than an indication of a greater difficulty of comprehension. This is acknowledged to some extent by Wiedenbeck and colleagues when they say:

> "With respect to control flow, we argue that, within a single program module, procedural and OO programs do not differ because in both styles local flow of control involves sequence, branching, and iteration. On the other hand, between module control flow may be clearer in a procedural program because a procedural program is normally based on a hierarchy in which a top-level function calls lower-level functions to carry out smaller parts of the overall task. It is relatively easy to determine where the top is and to understand the calls through successive layers of decomposition. In OO programs there is no top level, but rather parts of a task are distributed across objects which pass messages to other objects to act on their behalf" (Wiedenbeck et al., 1999, pp 259-260).

The difficulty is that this research does not seem to have considered how object-oriented programs differ from imperative programs when it comes to the mental representations involved. Rather Wiedenbeck et al. are assessing whether those programmers familiar with the object-oriented paradigm perform with similar characteristics to those doing imperative programming. These may be the incorrect measures with relation to object-oriented programming.

This makes some of the conclusions questionable. When Wiendenbeck et al. (1999) say,

> "The distributed nature of control flow and function in an OO program may make it more difficult for novices to form a mental representation of the function and control flow of an OO program than of a corresponding procedural program" (p. 276)

and

> "We tend to believe that the comprehension difficulties that novices experienced with a longer OO program are attributable partly to a longer learning curve of OO programming and partly to the nature of larger OO programs themselves." (p. 277).

It is difficult to assess whether this is implying that the OO paradigm is a not a "natural" way of conceptualizing and modelling real world situations. If the measures used for verifying the mental representation are not accurately reflecting the object-oriented mental representation then these types of claims are difficult to support.

In endeavouring to encourage further research into the psychology of programming with respect to object-oriented programming, Sajaniemi and Kuittinen (2007) have argued that:

> "Object-oriented programming is so much more complicated than imperative and procedural programming—both at the concrete notational level and at a more abstract conceptual level" (p 87).

They argue that there is a higher notational requirement for object-oriented over procedural (pp 88-90), that object-oriented requires a more complex notational machine

(p 90-91) and that object-oriented is data focused rather than functionally focused (pp 91-92). They argue that "educational and psychological research into novice imperative and procedural programming tells us that even the simplest imperative notional machine is hard for students to learn" (p 93) They say that "OO designers seem to base their solutions on the problem domain itself, whereas procedural designers use generic programming constructs for structuring their solutions" (p 94). They argue that we need to know more about object-oriented experts' mental representations with respect to object-oriented programming, the cognitive development of novices with respect to the paradigm, methods to convey the object-oriented notational machine to novices, and novices' and experts' program comprehension processes (p 96-97).

The assessment used for the naturalness of understanding program code has remained constant despite the change in programming paradigm. Sajaniemi and Kuittinen (2007) argue that further research needs to be conducted to identify what mental representations are used by object-oriented programmers.

2.2.4 Predictors of success

One approach to overcoming the failure rate in learning to program is to endeavour to identify the predictors of success in learning to program. The following describes some of this research.

Byrne and Lyons (2001) examined student attributes and their relationship to the students success in introductory programming. The style of programming being taught is not discussed in the paper. Factors considered included gender, experience, examination scores in non-computing subjects, and student learning styles. Lack of experience was shown to be a disadvantage and they concluded that there was a "clear link between programming ability and existing aptitude in mathematics and science subjects" (p 52). There were no other significant conclusions although they note that there was a predominance of convergent style learners in the study group and these appeared to perform better than student with other learning styles.

Wilson and Shrock (2001) explored a number of factors including gender, past programming experience, past non-programming experience, comfort level, encouragement to do computer science, work style preference, attributions, self-

efficacy, and maths background. They concluded that none of the factors was a clear indicator of success although they do argue that the factor with the highest rating (comfort level) should be fostered in the class setting.

Rountree et al.'s (2002) study focussed on novice programmers or those with limited programming experience. The factors considered included status (gender, age, enrolment status …), background (previous study – mathematics, humanities, etc. and previous programming language experience), and expectations. This study concluded that there are no clear indicators although the students' own assessment that they would get an A grade in the subject correlated highly with success. This study would suggest that the attitudes of the students toward programming may be the strongest indicator.

Rountree et al. (2004) reanalysed their data from the previous study (2002) using decision tree analysis and uncovered not predictors of success but "danger zones" in which students with certain combinations of attributes had a high probability of failure. They suggest that based on their analysis, it might be possible to identify students who are "at risk" and provide additional support.

Ventura and Ramamurthy (2004) sought to determine whether prior programming experience had any impact on an objects-first approach to teaching programming. They found that there was no correlation even where students had used a language that supports object-orientation or claimed to have reasonable knowledge of the object-oriented paradigm. They could only speculate as to what students meant by having reasonable knowledge of the object-oriented paradigm.

Bennedsen and Caspersen (2005) argued that there were significant differences in the way that object-first teaching of object-oriented programming is conducted and that the results of any study such as Ventura and Ramamurthy's (2004) can only be applied to approaches similar to that which they used. In their course, Bennedsen and Caspersen applied a model-driven programming approach. They said, "The course utilizes an apprenticeship-based pedagogy where students are exposed to how an expert programmer works" (p 157). From their study, they claim that the high school maths grade of the student and the student's results in course work were good indicators of success in the exam for their approach to teaching.

Bennedsen and Caspersen (2006) examined the relationship between abstraction ability and the learning of object-oriented programming. They analyse abstraction ability based on stages of cognitive development. This gives them a measure that is independent of programming ability. They found that there was no correlation between the cognitive development measure and programming ability in their model-based introductory programming course. They accept that the cognitive requirement in their course may be very low and that this may explain the lack of correlation.

Garner et al. (2005) and Hanks (2007) completed studies to determine the problems for which novice programmers were asking for help from tutorial assistants. The two studies used very similar methodologies and arrived at very similar results. The primary area of difficulty was language mechanics (i.e. syntax problems). These studies do not report the approach taken to teaching but it does at least show that at two different institutions similar problems are being encountered.

De Raadt, Hamilton, Lister, Tutty, Baker et al. (2005) explored the correlation between the student's approach to learning and their mark in a programming course. They contend that most other studies are based on Biggs' 3P (presage, process, product) model (J. B. Biggs & Moore, 1993) and endeavour to find pre-existing attributes that would provide an indication of success in programming. They argue that there is a positive correlation between the approach to learning and the marks in the course. There is no discussion of the influences on the approach to learning.

Fleury has conducted two variations of a study aimed at uncovering student understanding of Java programming rules and stylistic practices (2000; 2001). In the first of these studies (2000), the students were asked to "predict which programs […] would work according to specification" (p 197) and to explain their reasons for these predictions. Fleury argues that students construct their own rules during instruction and that students indicate that programming assignments were their primary source for developing these rules. Fleury found that the students had "constructed incorrect rules by misapplying correct rules" (p 199). Their knowledge was only partial because they had never seen, or been asked to apply, the alternatives that would have highlighted the correct form of the rule. Student suggestions on how to improve their understanding included "providing a workbook having very simple problems with answers for students to work before tackling more difficult sample programs and programming assignments"

(p 199), and ensuring that “they had time to “play” with the programs” (p 199). She argues that in teaching, it is important to have students explain their reasoning and to encourage them to set goals that involve understanding.

In the second study, Fleury (2001) used a very similar research approach to explore student understanding of “stylistic criteria including ease of comprehension, ease of debugging, ease of modification, ease of reuse, and overall quality of design” (p 189). They were again asked for the reasons for rating programs against stylistic criteria. Students argued that reduction in program size led to easier understanding of the code even though this might lead to duplication. However, there was also a willingness to accept duplication if this meant that the role of a class was more obvious arguing that this gave a clearer meaning of purpose. Fleury argues that “a real appreciation of the issues comes only with experience” (p 192) so it is important to “Show students good object-oriented design from the beginning” (p 192). In line with Hoadley, Linn, Mann, & Clancy (1996), Fleury recommends that “abstract comprehension of the whole procedure or program” should be emphasized and “not just line by line code comprehension” (p 192).

These studies show that there is no clear correlation between background factors and possible success in object-oriented programming. However, there is a further difficulty in that most of the studies, except for that of Bennedsen and Caspersen (2005, 2006 #4420), make no attempt to explain their approach to teaching object-oriented programming, nor do they define what they mean by that term. Fleury (2001) does make some recommendations on approaches to teaching based on constructivist approaches. Like the studies on the naturalness of object-oriented programming and mental representations for object-oriented programming, there is an assumption that every reader will understand what object-oriented programming means. There is also an assumption that all methods of teaching object-oriented programming are equivalent. This lack of definition makes it difficult to evaluate the results of these studies.

2.2.5 A learning outcome focus

Another approach to evaluate learning to program is to focus on graduates and endeavour to determine whether they really have learnt to program. From this

perspective, the focus goes off predictors for success or the analysis of particular reasons for failure and moves to what the students have actually learnt.

McCracken et al (2001) endeavoured to assess what graduates knew through the assessment of their programming skills. The assessment task involved the design and writing of a small program based around being able to evaluate different forms of programming language expressions. Some have questioned whether these were appropriate exercises for this type of study. The results of the study showed that graduates had not learnt the skills required to complete the assessment. The graduates performed worse than the research team had anticipated.

Lister et al (2004a) took an alternative approach and assessed student's reading and tracing skills of programs as they completed their first course in programming. They argued that a child is not taught to write before they can read. A learner should be able to read program code before they are expected to write code. Their assessment therefore focussed on the learner's ability to trace code segments. The results of this study showed that students had a poor comprehension of program code.

In follow up studies (Lister, Simon, Thompson, Whalley, & Prasad, 2006b; Thompson, Whalley, Lister, & Simon, 2006b; Whalley, Lister, Thompson, Clear, Robbins *et al.*, 2006), the questions from Lister et al. (2004a) were revised based on the perceived difficulty with respect to the 'understand' category of the revised "taxonomy of educational objectives" (L. W. Anderson, Krathwohl, Airasian, Cruikshank, Mayer *et al.*, 2001). The questions were predominately multi-choice questions (MCQs) with two questions asked that could be analysed using the SOLO taxonomy[5] (J. B. Biggs & Collis, 1982). These were an "explain in plain English" question and a code classification question. The aim was not simply to see whether students could trace code but to discover why they might be having difficulty with tracing the code.

[5] SOLO stands for Structure of the Observed learning Outcome. It is based on a quantitative measure (a change in the amount of detail learned) and a qualitative measure (the integration of the detail into a structural pattern. The lower levels focus on quantity (the amount the learner knows) while the higher levels focus on the integration, the development of relationships between the details and other concepts outside the learning domain.

In the first report on these studies (Whalley et al., 2006), it was reported that those who performed best in the code tracing exercises tended to use a relational understanding[6] with respect to the "explain in plain English" question. They tended to see the overall functionality of the code or have an abstract comprehension of the code rather than a line-by-line comprehension. Continuing studies are being conducted to confirm this perspective and to see whether this flows into better understanding of the writing of code.

The analysis of the code classification question also supported the view that there were students who were able to compare the code based on a relational perspective but that the poorer performing students focussed on particular language constructs rather than the code's purpose (Thompson et al., 2006b). The skills of being able to identify similar functionality would seem to be key in identifying possible code reuse opportunities and for the removal of code duplication but the students were clearly not showing this capability.

Like Fleury (2001) and Hoadley (1996) the authors of this study argued that there was a need to develop students' abstract comprehension ability. The authors recognise that the questions asked are procedural programming in nature and that the results so far do not include the evaluation of code writing ability. The application of these results to object-oriented programming and the relationship between the style of thinking and ability to write code are to be explored in follow up studies to be conducted in 2008.

2.2.6 Conceptions of learning to program

Another way of exploring the issues surrounding learning to program is to endeavour to determine the conceptions that students have of different phenomena in the subject area. One of the methods of conducting this type of research is through phenomenographic studies (see section 4.2, page 83). In this section, a number of studies related to programming and student conceptions are reviewed.

---

[6] The SOLO relational level focuses on the integration of the detail. In the context of these studies, relational understanding was interpreted as being able to summarise the purpose of the code rather than simply being able to trace the execution of the code line by line (multi-structural).

Booth (1992) explored, using a phenomenographic study, student conceptions of programming. She examined the student conceptions of the nature of programming, the nature of a programming language, and learning to program. With respect to the nature of programming, Booth identified three conceptions that students had. These are:

1. "**Programming as a *computer oriented* activity**, in which programming is conceived of as an activity that focuses on the computer;
2. "**Programming as a *problem oriented* activity**, in which the main focus is on the problem that the programming activity is intended to solve, rather then on the computer itself; and
3. "**Programming as a *product oriented* activity**, in which the main focus is on the program as a product, in the sense that programming is an activity for producing programs for potential users, and/or accessed for maintenance by other programmers" (p 94).

Booth argues that students expressed these conceptions of the nature of the programming task. Any particular student may have expressed these views at different times during the interview or simply have been focused on a particular conception. The difference in these categories is based on the focus of the task. At the lowest level, the focus is toward the computer and endeavouring to get the computer to do something. At the highest level, the focus is on the result of the programming task as something that others would use. It is important to recognise that these are conceptions of programming represented in the students' views and are not dependent on the learning environment although the learning environment may influence these conceptions. These categories may assist the teacher in endeavouring to provide a focus in the teaching of programming but they lack a relationship to the characteristics of a particular programming paradigm and how it might be taught to foster any given conception.

In the previous discussion (see section 2.1.2 on page 11), these aspects were also evident. There is a computer oriented activity focus in discussing the nature of an algorithm (Guzdial & Ericson, 2005), and in describing design patterns (Gamma et al., 1995). These are the things that need to be described and provided to the computer in order to have a functioning program. Robbillard (1999) has a problem oriented activity focus with his emphasis on software development as a knowledge-intensive activity. Armour (2000a) argues for something which is beyond the product oriented activity

category when he argues that software is not a product but his use of the term ‘product’ is to argue against a particular software development process paradigm. He is product focussed in the sense that he argues that the software is a medium that captures knowledge and the process is a knowledge-acquiring activity (problem oriented activity focus). This emphasis was possibly not evident in the interviews that Booth conducted with students.

What is important to recognise with Booth’s categories is that the higher categories are inclusive of the lower categories. A product oriented activity focus does not do away with the need to problem solve or to be aware of the needs of the computer system. However, it is possible to be focused on the computer requirements and ignore the problem oriented and product oriented activities.

Fleury (2001) claimed that her “study examined students’ understandings of programming as a product-oriented activity” based on Booth’s descriptions (p 189). The difficulty is that Fleury does not justify this claim or show how this orientation influenced the selection of students or her questioning. Fleury may believe that the issues that she was examining came from this product oriented focus but there is no guarantee that this is the conception held by the students that took part in her study. She provides no evidence in the report that would indicate this attitude was actually critical to the study’s findings or that describe how the teacher or researcher endeavoured to ensure this was the conception held by the students. It may be that the programming exercise set in the course was related to the development of a product but that does not mean that the students held this conception when completing the course work or when being interviewed by Fleury. The answers provided by the students may be more indicative of their focus on the nature of the program rather than the questions asked by the researcher.

With respect to the nature of programming languages, Booth (1992) identified four conceptions. These are:

1. “**Programming languages as a *utility program***, in which a language is seen as an object in its own right, which enables programs to be written to have certain properties, such as suiting a particular domain, and being fast in execution;

2. "**Programming languages as a *code***, in which a language is seen as a set of instructions, commands, symbols, and constructs which are the stuff programs are made of;
3. "**Programming languages as a *means of communication*** between the parts of the programming system, such as between the programmer and the computer – in which a language is the means the programmer uses to make the computer do the tasks required – or between the computer and program's ultimate user; and
4. "**Programming languages as a *medium of expression***, in which a language facilitates the programmer in expressing a solution to a problem or in expressing an idea in a way which enables the computer to effect it" (p 107).

Armour's concept of software as medium of expression (2000a) is more evident in this set of categories. The idea of a "programming language as a means of communication" is also evident in the focus on programming by intent and the argument that the program code will tell you when it needs to be refactored (Jeffries, 2004). In the agile approaches to programming, the program code is seen as the embodiment of the design and this should be clearly visible in the actual code and not require supporting comments or design documentation. If that clarity is not there then the code needs to be reshaped to better express that design. In this latter respect, the code has gone beyond being a medium of communication to become a medium of expression.

There seems to be another subtle difference between Booth's description of these categories and the expressions used by agile proponents (Auer & Miller, 2002; Beck, 2000; Jeffries, Anderson, & Hendrickson, 2000). In Booth's categories, there is still an emphasis on the programming language as being something aimed at getting the computer to do something. The distinctions in her categories tend to be on the way that code is developed. Underlying the agile perspective is the idea that the code is not just "expressing an idea in a way which enables the computer to effect it" (Booth, 1992, p 107) but the code is expressing the design and the solution to the problem in a way that other programmers can understand how that solution was created. The use of the test-driven approach to software development is seen as further enhancing this expression by helping communicate what the programmer understood the problem to be. If the tests do not cover a particular issue then the programmer did not perceive that to be part of the programming problem.

Booth's (1992) final set of categories relate to "conceptions of learning to program" (pp 119-129). Here she presents four conceptions. These are:

1. "**Learning to program as *learning a programming language*,** in which focus is on learning the features and the details of one or more programming languages;
2. **Learning to program as *learning to write programs in a programming language***, in which focus is on learning to write programs which make use of available techniques and features of the programming language;
3. **Learning to program as *learning to solve problems in the form of programs***, in which focus is on learning to produce programs according to the needs of problems;
4. **Learning to program as *becoming part of the programming community***, in which focus is on learning to solve problems and write programs in collaboration with, or for, someone else, and thereby participate in the world of programming" (p 119).

When we look at the issues being investigated in learning to program research (see section 2.2.1, page 16), the distinction between novices and experts (see section 2.2.2, page 17), and the naturalness of programming (see section 2.2.3, page 18), many of these conceptions are evident. Booth arrived at her categories by examining the ways that learners expressed their understanding of what they were doing.

Bruce, Buckingham, Hynd, McMahon, Roggenkamp *et al.* (2004; 2003) have used a phenomenographic approach to explore the ways of experiencing the act of learning to program. They say:

> "Analysis revealed that students might go about learning to program in any of five different ways: by (1) Following – where learning to program is experienced as 'getting through' the unit, (2) Coding – where learning to program is experienced as learning to code, (3) Understanding and integrating – where learning to program is experienced as learning to write a program through understanding and integrating concepts, (4) Problem solving – where learning to program is experienced as learning to do what it takes to solve a problem, and (5) Participating or enculturation – where

> learning to program is experienced as discovering what it means to become a programmer" (C. Bruce et al., 2004, p 143).

There are parallels between this hierarchy and those presented by Booth for learning to program (1992). Bruce et al. have an additional category at the lower end that they describe as "following" and their "understanding and integrating" category may not be the equivalent of Booth's "learning to write programs in a programming language". The outcomes here seem to validate Booth's earlier work.

Bruce et al. (2003) also describe what they see as the variations that are specific to each of their categories. They say "The dimensions of variation associated with each different way of experiencing the act of learning to program are 1) learning activities or approaches, 2) how learning a programming language is seen, 3) key motivations in learning to program and 4) ways of seeing programs and programming" (p 10). Although they discuss their outcome space in terms of the implications for teaching, the recommendations are not clear.

Eckerdal and Berglund (2005) conducted a phenomenographic study on how students understood what learning to program meant. They argue that the students clearly indicated that learning to program involved learning a different way of thinking. The students had difficulty expressing what they meant by a different way of thinking but in analysing the interviews, Eckerdal and Berglund arrived at five categories. Their categories have some similarities to those of Booth (1992) and Bruce et al. (2004; 2003) but they make no direct comparison. Instead they focus on two of their categories in relation to the work of Hazzan (2003) with its 'process-object duality'. Hazzan says "Using the process-object duality terminology we may say that solving a problem by relying on a canonical procedure is an expression of process conception of the concepts under discussion; solving a problem by analyzing the essence and properties of concepts is an expression of object conception of the concepts under discussion" (p 108). Hazzan argues that the process conception precedes the object conception; that is the learner applies a thought process for abstraction prior to being able to discuss the nature of that thought process. Working from this basis, Eckerdal and Berglund (2005) are seeking to see where the student conceptions of learning to program move from a process conception to an object conception. This clearly influences their outcome space. It is

difficult to see how this transformation is reflected in their five categories. The categories are:

1. “Learning to program is experienced as to understand some programming language, and to use it for writing program texts.
2. “As above, and in addition, learning to program is experienced as learning a way of thinking, which is experienced to be difficult to capture, and which is understood to be aligned with the programming language.
3. As above, and in addition, learning to program is experienced as to gain understanding of computer programs as they appear in everyday life.
4. As above, with the difference that learning to program is experienced as learning a way of thinking which enables problem solving, and which is experienced as a “method” of thinking.
5. As above, and in addition, learning to program is experienced as learning a skill that can be used outside the programming course.

They argue that in category two, “the students have noticed that a special way of thinking is required, but not what that is. In category four, on the contrary, the students have realized that it has to do with problem solving and a systematic way of thinking” (p 140). Their view is that the process-object duality is represented by a shift from awareness of the thinking process to an ability to recognise some of its core properties. They conclude that “it is of great importance that students reach an understanding expressed as *learning to program is a way of thinking, which enables problem solving, and which is experienced as a “method” of thinking*” (p 141).

In the preceding studies, the focus was the students’ overall understanding of the concept of programming or programming language or learning to program. There are concepts within the domain of object-oriented programming that could be studied. Eckerdal and Thuné (2005) explored student perceptions of objects and classes. The results are two parallel hierarchies that describe the student comprehension of object and class (see Table 2.1).

Conception of object	Conception of class
Object is experienced as a piece of code.	Class is experienced as an entity in the program, contributing to the structure of the code.
As above, and in addition, object is experienced as something that is active in the program.	As above, and in addition, class is experienced as a description of properties and behaviour of object.
As above, and in addition, object is experienced as a model of some real world phenomenon.	As above, and in addition, class is experienced as a description of properties and behaviour of the object, as a model of some real world phenomenon.

Table 2.1: Conceptions of object and class (Eckerdal and Thuné (2005))

In their discussion, Eckerdal and Thuné recognise the similarity in these results and that "There are few, if any, examples where students show an advanced understanding of one concept, and a poor understanding of the other concept" (p 92). The concept of a class as a description of an object is clear in the two highest categories but not so clear in the initial category. The shift from the first category to the second is a realisation that the class and object are distinct. The object is not the code of the class but rather an entity that is described by a class. Like Fleury (2000), they argue for the use of appropriately constructed examples to highlight the distinction between object and class.

2.2.7 Threshold concepts

Threshold concepts are described by Meyer and Land (2005) as:

> Threshold concepts "may be transformative (occasioning a significant shift in the perception of a subject), irreversible (unlikely to be forgotten, or unlearned only through considerable effort), and integrative (exposing the previously hidden interrelatedness of something). In addition they may also be troublesome and/or they may lead to troublesome knowledge for a variety of reasons" (pp 373-374).

Eckerdal, McCartney, Moström, Ratcliffe, Sanders et al. (2006) contend that two candidates as threshold concepts in computer science are abstraction and object-orientation. They argue that these are candidates because they are frequently reported in computer science literature as troublesome topics.

In a follow up study on threshold concepts in computer science, Boustedt , Eckerdal, McCartney, Moström, Ratcliffe et al. (2007) report that object-oriented programming and pointers were identified as two threshold concepts. The concept of pointers is quite low level compared with object-oriented programming. Students do see programming in a different light after understanding object-oriented programming but there are many concepts that comprise object-oriented programming. The threshold concept within object-oriented programming would be that which transforms the students understanding of programming and allows them to comprehend what is involved.

More importantly, if object-oriented programming is seen as a threshold concept then it is not an extension of procedural programming. There is something unique about this approach that both makes it difficult to learn and, as Boustedt et al report, the "knowledge has *transformed* how the student looks at problems" (p 506).

Meyer and Land suggest that "*ways of thinking and practising* also constitutes a crucial threshold function in leading to a transformed understanding" (2003, p 9). It would therefore seem plausible that in order to understand how object-oriented programming constitutes a threshold concept we need to explore the "ways of thinking and practising" involved in this approach to software development.

2.3 Teaching strategies

There is a lot of literature related to teaching strategies for teaching CS1, the introductory computer science paper[7] that includes an introduction to programming. Many of these are not related to particular pedagogical approaches but rather focus on lecture experiences or tools developed to aid learning. This section is not going to

[7] In New Zealand, the term paper is used in relation to a unit of study taken within a programme of study.

attempt to review all of these articles but rather to focus on articles that are relevant to this research.

2.3.1 Wirth's exemplary textbook

Wirth as the developer of the popular procedural teaching language, Pascal, described in a keynote address to the 2002 ITiCSE conference, what he calls an 'exemplary textbook'. He argues that "It should satisfy the following criteria:

1. It starts with a succinct introduction into the basic notations of program design.
2. It uses a concise, formal notation. This notation is rigorously defined in a report of no more than some 20 pages.
3. Based on this notation, the basic concepts of iteration, recursion, assertion, and invariant are introduced.
4. A central topic is the structuring of statements and typing of data.
5. This is followed by the notions of information hiding, modularization, and interface design, practiced on exemplary application.
6. The book establishes a terminology that is both intuitive and precisely defined.
7. The book is of moderate size" (Wirth, 2002, p 3).

Wirth does not argue for a particular paradigm but he does conclude that the challenge to create such a textbook "has not been met so far" (p 3). Some core principles underlie Wirth's criteria. One is related to the size of the language and its ability to be represented rigorously and succinctly. He also outlines what he sees as basic concepts, central topics, and notions. These criteria are not paradigm specific but can be applied to the writing of a textbook for any paradigm.

2.3.2 Pedagogical patterns

An avenue of investigation for the improvement of teaching and learning of any programming paradigm is the use of pedagogical patterns (Eckstein, Manns, & Voelter, 2001; Fincher, 1999; Fincher & Utting, 2002; Haberman, 2006). This has led to the writing of a number of pedagogical patterns targeted at the computer science community (Bergin, 2000, 2002; Muller, Haberman, & Averbuch, 2004). Patterns such as feedback patterns (Bergin, Eckstein, Manns, & Sharp, 2001a), active learning

(Bergin, Eckstein, Manns, & Sharp, 2001b), teaching from different perspectives (Bergin, Eckstein, Manns, Sharp, & Sipos, 2001; Bergin, Eckstein, Manns, & Wallingford, 2001), and experiential learning (Bergin, Marquardt, Manns, Eckstein, Sharp *et al.*, 2001) endeavour to capture generic educational principles in the notation of a pattern language (Alexander et al., 1977).

The argument from this approach is that to improve the learning of students in programming classes regardless of the programming paradigm, the educational lessons of good teaching practice need to be used in computer science classrooms. The teaching practices are documented as pedagogical patterns in the same way as software development patterns are documented to make them more accessible to the computer science discipline.

2.3.3 An approach for teaching programming

When the researcher was teaching novice programmers with limited computing experience, there was a need to develop a programming course that would overcome the conceptual difficulties of these students. For a course teaching procedural programming, over fifty percent of those enrolled never submitted the final assessment. A new teaching approach was developed and a workbook written (Thompson, 1992). The resulting course had a higher pass rate and a higher completion rate than the previous course offering. The success was dependant on both the materials and the way the paper was taught.

The first objective in the revised course was to build a conceptual understanding of nature of a program(me). The emphasis was on Wirth's definition of a program as algorithm plus data (Wirth, 1976). A series of resources were prepared that drew on familiar programme concepts that were wider than computer programs (see Appendix 1). There were three sets of examples. The first was a set of recognisable programmes. The second was a set of items that were not programmes but had some of the characteristics of a programme (i.e. the data). The third set contained a mix of both. In class, the students looked initially at the programme items and were asked to say what characteristics made them a programme. They were then asked to look at the non-programmes and asked which characteristics these did not have. With the final set they were deciding which were, and which were not, programmes. From a teaching

perspective, there were three key characteristics about this exercise. These were the connection with the students' backgrounds, the bringing of characteristics of a programme to the fore and associating this with a computer program, and using variations in examples that helped the key characteristics to become visible. The sets have been extended to include object-oriented concepts (Thompson, 2006).

The second activity was to look at how programs are created. This included looking at problem solving strategies and the program development life cycle. This included some key questions to help understand the nature of the programming problem and strategies for building solutions. These strategies included looking for familiar things, divide and conquer, and using building blocks. These strategies were initially discussed and later formed the foundation for interactive class sessions where small programs were written.

The students were introduced to the concepts of a programming language using simple English grammar. The key idea was to help the students understand that both the syntax and the semantics had to be valid in order for the computer to understand what was being requested. Examples of valid and invalid syntactical sentences were presented and the reason why they were valid or invalid was discussed. They were then introduced to the simple syntax (see Appendix 2) and asked to review the sentences again. There was a continual reinforcement of the key characteristics of the syntax through the examples and questions. They were then given a set of sentences that were syntactically correct but were a mix of valid and invalid semantics. This provided an opportunity to talk about the meaning of sentences, and why they needed to be both syntactically and semantically correct. This was followed by introducing them to the syntax of the language to be used in the course. This was a subset of the Pascal language.

The students were then introduced to the logic constructs as the basic building blocks for building programs. Each construct was introduced using diagrams of a non-computing example, flow charts, structure diagrams, and valid syntactical statements in the language. Having introduced the constructs, the idea of building programs by combining constructs was discussed and illustrated through worked examples. To further reinforce the learning, each construct was revisited as a pattern of logic with a series of questions that would help them think about how the logic constructs were used (Dale & Weems, 1992). In the case of loop constructs, the patterns of common usage patterns were introduced and illustrated.

In the laboratory tutorials, the students worked through a set of progressive programming problems. These started with simple programming sequences and introduced each new construct progressively. They used the patterns that were discussed in the lectures. Each construct had multiple exercises associated with it and the size of the exercises enabled the students to complete a number each week. They were asked to submit one exercise for each construct. Exercises from the progressive exercises were used as lecture exercises. This again emphasised variations in the use of the constructs and reinforced the questioning and thinking patterns introduced in earlier classes.

The final assignment for the course was to write a small invoicing program. By this stage, the progressive exercises were asking for a reasonable sized piece of code and the students had developed strategies for breaking the problem up and completing the programming task.

The course was successful (higher number of students completed and passed the course than the previous course offering) because of the sequence of the materials and the way that it was taught which promoted the key aspects as the focus of the teaching. The students were not just presented with definitions and examples but were encouraged to develop their own definitions for the key aspects that they were learning.

The teaching patterns used reflect ideas promoted through variation theory. Although this was developed for teaching procedural programming, the ideas are transferable to other paradigms. The elementary patterns approach picks up on the use of a base set of patterns for teaching object-oriented programming but not necessarily the emphasis on variations of the key aspects.

2.3.4 Using the roles of variables in teaching

Another teaching strategy being reported as successful is the use of the roles of variables (Sajaniemi, 2002). The emphasis in the reports is on having a small number of variable roles that cover over ninety percent of variable usage in novice programming exercises. These are then reinforced and repeatedly used as students work through other materials to learn the programming language and logic constructs (Kuittinen & Sajaniemi, 2003, 2004; Sajaniemi & Kuittinen, 2005).

The selection of appropriate roles was based on the analysis of programs in three introductory PASCAL textbooks (Sajaniemi, 2002, p 113). These were expanded in later studies as they looked at computer science educators' experience with them (Ben-Ari & Sajaniemi, 2004) and other programming paradigms (Sajaniemi, Ben-Ari, Byckling, Gerdt, & Kulikova, 2006).

A role of a variable is "the dynamic character of a variable embodied by the sequence of its successive values as related to other variables" (Sajaniemi, 2002, p 112). Sajaniemi contends that they "have shown that there is a small set of roles of variables covering the vast majority of variables in novice-level programs. Roles capture higher-level program information of variables that can be utilized in program visualization for novices" (p 119). The roles act as "an instrument for thinking" about program design and variable usage (Kuittinen & Sajaniemi, 2004, p 58).

To validate the roles identified, they carried out an experiment with computer science educators who were asked to identify the roles of variables in a set of programs after a short training session on the definitions (Ben-Ari & Sajaniemi, 2004). They claim that the only points of controversy were related to atypical use of variables within the code (p 56). In another study, they asked experienced programmers to sort variables and concluded that roles are "part of experts' programming knowledge" (Sajaniemi & Prieto, 2005, p 158). It is later reported that "Roles are not absolute in the sense that all expert programmers would recognize the same set of roles with the same set of distinguishing definitions" (Sajaniemi et al., 2006p 262).

From their experiments in teaching using the roles of variables, they believe that groups taught with the roles of variables approach "performed better in program comprehension and construction" exercises (Kuittinen & Sajaniemi, 2003, p356). They argue that compared with the elementary pattern approach to teaching programming, they are able to cover all the roles during an introductory programming course (Kuittinen & Sajaniemi, 2004, p 57). In their teaching strategy, the key aspects of the roles of variables become the focus of the teaching and are repeatedly revisited, and variations between the different roles are highlighted (p 59). It is argued that the students can use variable roles in the same way as elementary logic pattern as the "starting point in program design" (p 60).

They argue in favour of the use of roles of variables over elementary patterns on the basis that a wider number of design patterns exist than roles of variables but they do not discuss whether all are essential design patterns for novice programmers (Byckling & Sajaniemi, 2005, p 278; Kuittinen & Sajaniemi, 2003, p 347). In their own discussion, they acknowledge that their roles are drawn from novice programming exercises and therefore may not cover all the roles that an expert programmer may utilise. The previous example of teaching based on logic patterns would suggest that at least for procedural programming there is a minimal set of patterns that lays the foundation for learning programming.

In attempting to determine whether roles of variables have applicability beyond procedural programming, Sanjaniemi et al. (2006) looked at the functional and object-oriented paradigms. They concluded that there was general applicability although there were differences which they argue highlight the key characteristics of the different paradigms. In arguing that there might be applicability to other paradigms, they argue that:

> "Understanding a program requires, above all, an understanding of its variables—the statements then become a means to manipulate the values of the variables" and "if roles are to be a unifying concept the differences among the paradigms should not be too great" (Sajaniemi et al., 2006, p 262).

They argue that any differences in roles would "point to real differences between paradigms and can contribute to a better understanding of the effect of choosing a paradigm to teach programming, as well as to a better understanding of the pedagogy of each individual paradigm" (p 262). However, they did have problems with identifying roles in object-oriented programs since "the data flow is decentralized" (p 275). This made it difficult to identify places where variable values were changed. The focus on roles of variables therefore seems dependant on understanding the overall flow and is not dependant on the local behaviour surrounding the variable. Their understanding of the object-oriented paradigm is reflected in their statement that "Object-oriented textbooks tend to stress data modelling" (Sajaniemi et al., 2006, pp 276-277).

The roles of variables appear to provide a thinking framework that aids in the comprehension and writing of programming at least for the procedural paradigm. However, there are also signs that the teaching strategy with its focus on variations of critical aspects between the roles and repetition of use of the definitions may equally lead to the success that is reported. The authors of these studies believe that the roles of variables are independent of the programming paradigm even though they recognise difficulties in determining roles for the object-oriented paradigm. In a later paper they argue that there needs to be more research to understand the mental representations for the object-oriented paradigm (Sajaniemi & Kuittinen, 2007).

2.4 Conclusion / Research question

This chapter has reviewed research related to the learning of programming. Much of the material is related to the procedural paradigm. In many cases where research has been conducted on the object-oriented paradigm, there has been no distinction made between the characteristics of the procedural paradigm and the object-oriented paradigm. There appears to be no evidence that this is a reasonable assumption nor, as some have argued, that the object-oriented paradigm is just an extension of the procedural paradigm. Sajeniemi and Kuittinen (2007), after conducting their exploration of the roles of variables in relation to three paradigms (Sajaniemi et al., 2006), are now calling for research focussed specifically on the object-oriented paradigm. They also argue that there is no research evidence that supports the view that the object-oriented paradigm should be introduced early.

It is argued that computer science is about the study of algorithms (Brookshear, 2007; Guzdial & Ericson, 2005) but an algorithm is defined as sequential in nature. The importance of the flow of control to the understanding of object-oriented programs has not been explored. There appears to be no discussion of alternative ways of expressing algorithms that are not sequential in nature. Alternative ways of expressing algorithms may provide implications for understanding algorithms within the object-oriented paradigm.

The research, with respect to the conceptions of object-oriented programming, that has been conducted so far has looked at conceptions of objects and classes (Eckerdal & Thuné, 2005) and what it means to learn program thinking (Eckerdal & Berglund,

2005). Boustedt et al. (2007) have also argued that object-oriented is a threshold concept in computer science but it is unclear what the threshold concepts are that would enable an understanding of the object-oriented paradigm.

The current study explores the perceptions of object-oriented practitioners in order to discover how awareness of an object-oriented program varies.

Chapter 3. A phenomenographic view of tertiary learning and teaching

In the previous chapter, some of the issues being explored in relation to learning to program were discussed. Some of the reported research identified questions about the assumptions about the nature of the programming task. There were also attempts to find a generic set of skills that apply to all programming paradigms. One argument is that in order to learn a programming paradigm, there is a need to understand the key aspects of that paradigm. The issues raised in the previous chapter were specific to learning to program and understanding the issues surrounding learning to program. The focus in this chapter moves to the issues of learning and the educational theories that need to be considered. This chapter analyses the literature about the nature of learning and teaching, drawing out some of the key issues.

3.1 Key findings of research in learning and teaching

Studies into the knowledge of experts show that experts "notice features and meaningful patterns of information that are not noticed by novices" (Bransford, Brown, & Cocking, 2000, p 31). Their knowledge is conditioned in that they know the circumstances in which it is appropriate to apply their knowledge.

In relation to programming, Soloway, et al. (1988) found that expert programmers use programming plans as a way of understanding a program. They recognise the patterns in the code and infer the behaviour. This is also reflected in the multi-choice question (MCQ) answers of experts with a relational view of code in the BRACELet studies (Lister et al., 2006b). Adelson (1981; 1984) describes expert programmers as conceptualising a program in terms of what the program needs to do and novices as conceptualising a program in terms of how the program operates.

A novice software developer has a focus on detail while an expert will recognise and utilise patterns (Pennington, 1987a; Soloway et al., 1988). Novices may seek example solutions while experts can develop solutions from concepts. Novices seek to apply design patterns and processes by rule while experts see these as guidelines to be adapted for the current task.

Learning in this sense means more than learning facts. It is about acquiring understanding and the conditioning of the knowledge so that it helps the learners understand when to apply this new knowledge (Bransford et al., 2000, p 8). The learners need to see the new thinking patterns as relevant to the context in which those thinking patterns are to be applied. The learners need to develop the expert's conditioning of knowledge so that they can make appropriate choices during problem solving.

Bransford et al. (2000) describe three findings from the research on learning and teaching which they argue has implications for "how we teach". These are:

- connecting with pre-existing understandings (pp 14-16);
- building some subject matter in depth by using many examples of the same concept (pp 16-17); and
- fostering metacognitive skills (p 18).

3.1.1 Pre-existing understandings

Bransford et al. contend that a key element in teaching is to encourage learners to learn facts against a conceptual framework (2000, pp 16-17). They emphasise the need to "draw out and work with the pre-existing understandings" that learners bring with them (p 19). Part of the learners' context of learning is the conceptions of the subject and the background knowledge of key concepts that they bring into the class. These should be used and challenged to help the learners rethink their conceptual understanding of the subject area and of their conception of learning.

Lo, Marton, Pang, & Pong (2003) also recognise that background knowledge influences a person's ability to learn particular content. They suggest from a phenomenographic perspective that there is a limited range of variations in the background knowledge related to the teaching that learners bring to the classroom. By understanding these variations, it is possible to structure lessons to address specific learner needs.

The connection to background knowledge is reflected in the programming course discussed in the previous chapter (Thompson, 1992, see section 2.3.3, page 38) where the learners' existing use of the concept of a programme was made explicit and became a conceptual framework in which to learn programming. This enabled them to connect

existing knowledge with the new learning and to apply that knowledge to the problems that they were presented with.

It is less clear whether the roles of variables, used in teaching programming (Kuittinen & Sajaniemi, 2004, see section 2.3.4, page 40), connect with the learners pre-existing knowledge. There may be some familiarity in the role descriptions that aid learner understanding, which can be used to comprehend the behaviour of a program and to explain the intended purpose of a piece of code. The roles, although focussing on the way that changes of state occur for the variable, do imply certain types of behaviour in the code.

Elementary patterns (Clancy & Linn, 1999) also provide a conceptual framework. When taught using scaffolding strategies (Rosson & Carroll, 1996; L. Thomas, Ratcliffe, & Thomasson, 2004), they are connected with pre-existing knowledge and used to challenge existing conceptions and assumptions.

3.1.2 Build some content in depth through examples

Bransford et al. argue that learning is not about a set of disconnected facts (2000, pp 16-17). They contend that experts "draw on a richly structured information base" (p 16). Experts do not necessarily have better memories but they do have a conceptual understanding that allows them to see relationships and to draw on a vast pool of knowledge. Brandsford et al. argue that to achieve this type of information base, the learners need to build some subject matter in depth by using many examples of the same concept. This learning is further enhanced if the concept is presented in multiple contexts (p 62-63) and the learners are encouraged to generalise or abstract their conclusions (p 63).

Ramsden (1988a) emphasises what the learners do when he says "improving learning is about relations between learners and subject matter, not teaching methods and student characteristics" (p 27). Shuell has a similar argument, being "what the student does is actually more important in determining what is learned than what the teacher does." (1986, p 429) This is not to say that it does not matter what the teacher does; rather it is necessary to engage the learner in learning activities that are likely to achieve the desired outcomes.

Wirth's exemplary programming textbook (Wirth, 2002, p 3, see section 2.3.1, page 37) with its emphasis on a "concise, formal notation" and emphasis on basic concepts has relevance. Wirth does not want a programming language that requires the learning of many detailed tricks but a concise language, or possibly subset of a language, that would allow him to teach the core concepts. This avoids the distraction of lots of detail for the learners and allows them to focus on the concepts being learnt. If these core concepts form the basis for the desired conceptual understanding, then the learners can learn to relate these to other details later and adapt to changing knowledge. Wirth does not place any emphasis on what the learners are doing in order to learn, but rather on the quantity and clarity of definition.

3.1.3 Metacognitive skills

The learner's ability to assess and self-regulate the use of cognitive skills and learning strategies is called metacognition (Bransford et al., 2000, p 47). Brandsford et al. argue that the development of metacognitive skills has to be a core part of the learning process (p 50). Metacognitive skills allow learners to evaluate their learning and to revise their learning strategies based on perceived performance. Improved learning comes from a dual focus on the body of knowledge (content) and how to learn (process) (Murray-Harvey & Keeves, 1994). The process of learning how to learn involves the learners in developing learning strategies, in the assessment of their learning, and in revising their strategies (Angelo & Cross, 1993; Bransford et al., 2000; Brown, 1978, 1987).

There is a distinction between the person who struggles to learn programming (ineffective novice) and the person who can successfully learn to program (effective novice) with respect to the strategies that they utilise (Robins et al., 2003). The journey from ineffective novice to effective novice, and from effective novice to expert reflects the changing of perspectives in terms of what is being learnt and in the understanding of the learning task being performed.

3.2 Models of learning

Bransford et al. (2000) stated that learners learning was assisted by connecting to their prior knowledge. Biggs and colleagues (1993; 1999; J. B. Biggs & Moore, 1993) developed the 3P model of teaching and learning. The three Ps are presage, process, and

product. Biggs says that the "3P model describes three points in time at which learning related factors are placed" (1999, p 18). Presage relates to factors before learning takes place. Process is during learning. Product relates to the outcome of learning. Presage, process and product are related as depicted in Figure 3.1.

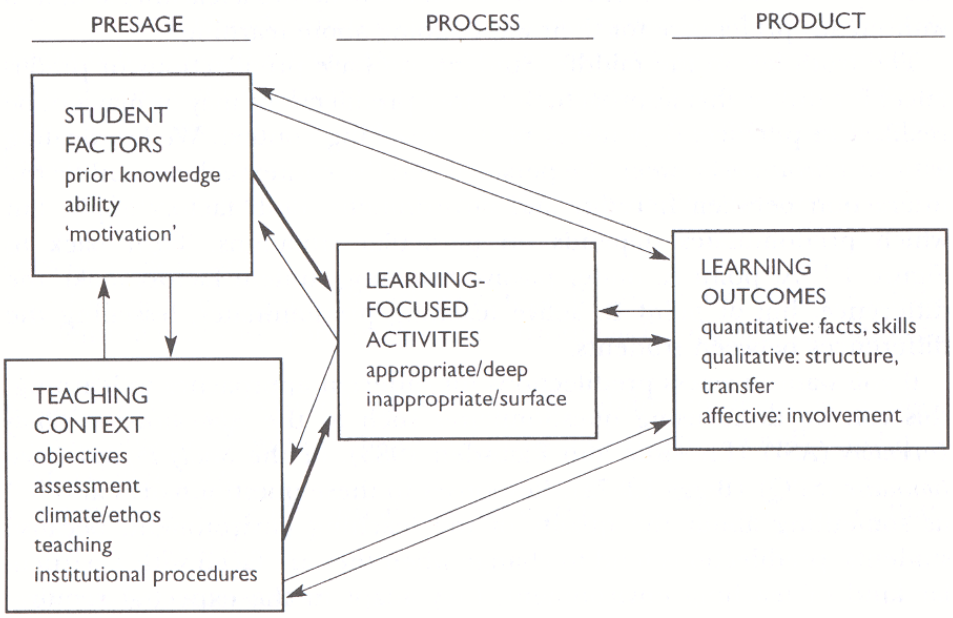

Figure 3.1: Biggs' 3P model of teaching and learning (J. B. Biggs, 1999, p 18)

Biggs claims that presage factors are of two kinds. These are the learner based factors such as the relevant prior knowledge that is related to the topic, the learners' ability, their commitment to study, motivation, etc. and the teaching context based factors as defined by the objectives, the approach to teaching, the assessment strategy, and surrounding factors in the classroom and institution. Biggs argues that these factors together influence the learner's approach to learning and that this influences the learning outcome.

Biggs' approaches to learning are influenced by the work of Säljö (1979a; 1979b), and Marton and Säljö (1976a; 1976b; 1997) who developed a hierarchy of five conceptions of learning. These were later expanded to six by Marton, Dall'Alba, and Beaty (1993). These are:

(A) Increasing one's knowledge

(B) Memorizing and reproducing

(C) Applying
(D) Understanding
(E) Seeing something in a different way
(F) Changing as a person

Biggs concentrates on three approaches to learning (surface, achieving, and deep) and uses survey instruments to assess the approaches being used by learners. Entwistle (1997) also describes three approaches to learning but defines them as deep, strategic, and surface. Biggs says that it is unlikely that a learner with little prior knowledge of the topic would use a deep approach (J. B. Biggs, 1999, p 19).

Biggs' model suggests that the teacher has influence on the learner factors and the approach to learning through the assessment criteria, objectives, teaching strategy, and the environment established in the course. In this model, the learner's perception of the task is part of the learner factors.

Ramsden (2003) also discusses the conceptions of learning and approaches to learning by drawing on the work of Säljö (1979a; 1979b), Marton and Säljö (1997), and Biggs (1987). Ramsden says that "Approaches to learning are not something a student has; they represent what a learning task or set of tasks is for the learner" (2003, p 45). Like Biggs, Ramsden does see the learning outcome impacted by the approach to learning (see Figure 3.2). However, Ramsden argues that the approach to learning is influenced by the learner's perception of the task. The perception of the task is influenced by all the factors that are part of the context of learning and the learner's orientation to studying. This includes the learner's conception of learning for the course, and the learner's perception of what it is that the learner is supposed to produce or the nature of the end product.

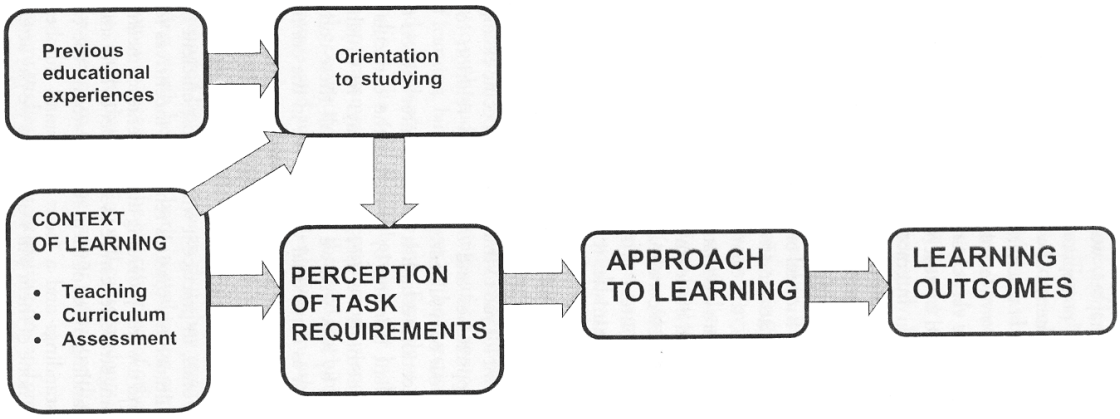

Figure 3.2: Ramsden's learner learning in context (Ramsden, 2003, p 82)

An example might be where the lecturer has set criteria for an assignment seeking to discover the learner's understanding and ability to use a number of web-development technologies but the learner perceives the assignment as being about developing a web site. The learner focuses on the design of the site and uses a tool that will enable them to demonstrate the design. In the process, the learner ignores the issues surrounding the different technologies. The learner has an excellent web site design but that is not the focus of the criteria. The learner's perception of the task has caused them to miss the intent of the assessment item.

Ramsden (1988b) contends that what is really learnt is driven by what the learner perceives as being required or by what they perceive the lecturers will reward. Bowden and Marton (2003) argue that learners "react to the learning environment as it is experienced by them" (p 8). As a result, learners may apply the opposite approaches to learning than those that would produce the lecturer's desired levels of understanding. Since the context can also influence the learner's orientation to study, Ramsden seems to be recognising more scope to the teacher to influence the learner's perception of the task and consequentially the approach to learning.

When considering the perception of the task, Marton contends that "The problem doesn't exist as such, it is always understood in one way or another" (2000, p 113), that is, any given task or problem is understood based on previous experience. The learner is either aware or unaware, or conscious or not conscious of the problem. Marton is saying that without someone experiencing or being aware of the phenomenon as a problem

then it is not a problem. In fact, in another person's experience of the phenomenon, that person may not recognise the phenomenon as a problem but as an opportunity. The concept of a problem exists only in an awareness relationship. In Marton's argument "an object of experience is not independent of the way in which it is experienced" (p 105). Each experience of the phenomenon may be different but without the experience, the phenomenon might as well not exist.

Learners will experience the learning exercises that are set in different ways. Marton illustrates this with an exercise in division (p 110-111). The teacher has a particular idea of how the problem should be solved but the learners use a range of techniques based on their understanding of the problem and how to solve it.

In terms of teaching programming, if the learners have no awareness of how to solve particular programming problems or of particular solutions to programming problems, then they will struggle to solve them or will use a solution that is not appropriate. The implications of what Marton is saying is that if learners are to see a problem in a specific way, then they need to be helped to experience it in that way.

The linking of perception of the task with the approach and outcome also offers an explanation for why the research on the naturalness of object-oriented programming may be returning a negative result (see section 2.2.3, page 18). The researchers did not investigate the perceptions held by the participants in the research but assumed that there was some consistent natural way of perceiving what an object-oriented program would be. The studies did not pursue whether the participants have some perception of what they would be looking at or whether the idea of structuring a program using objects that reflected real world entities was foreign to their way of thinking about programming.

3.3 Learning as a change in awareness

Ramsden's model (2003, p 82) places the "perception of the task requirements" as a driving force for the approach to learning and, consequentially, the learning outcomes. Bowden and Marton (2003) argue that effective learning or "the most important form of learning" involves a change in the way that the learner sees something in the world. The low level surface approach to learning strategies are not effective in helping the learner

see something differently in the world or in cultivating seeing things from a range of different perspectives. These changes in perception also involve changes in how learning is perceived.

For learners to change, there needs to be concern about how the learners perceive the subject matter and how their perceptions can be changed (Ramsden, 1988a). It is not possible to separate the idea being conceptualised from the person who is doing that conceptualisation. If the learner's perception of the subject matter is incompatible with the concept, then the learner will struggle with the concept. Learning is about "changes in people's conceptions of aspects of reality" (p 26). Knowledge of the facts and the skills are important, but the key to learning is the quality of understanding as reflected by conceptual understanding. A learner who can recall facts or apply a formula by plugging in new numbers has not necessarily learnt the concepts and thinking patterns that justify the formula, or understood the appropriate use of the formulae. In effect, they have not learnt the process of the subject area.

Barclay (1975) uses an illustration of a country man visiting the office of a city friend to illustrate how different people can be aware of different things in the same environment. The country man hears a grasshopper over the noise of the city street. His city friend cannot hear the sound. Each person has been conditioned to their environment and brings with them the perceptions and expectations of that environment. Each learner comes to the learning situation conditionalised to their environment and carrying expectations of learning and what is to be learnt. The preconceptions they bring to the learning situation about the subject matter and about learning can blind them to what is to be learnt. To enable learning, it may be necessary to remove the blindness or to challenge preconceived ideas of the subject so that the learner views the subject matter through different lenses.

Mezirow (2000) argues this from the perspective of changes in our frame of reference or world view. For learning to occur, there may need to be a transformation in the frame of reference so that the learner looks at what is to be learnt from a different perspective. This transformation may involve a shift in focus of the subject in our knowing to it becoming the object of knowing (Kegan, 2000). Kegan contends that what is "subject" of our knowing, "describes the thinking and feeling that has us", while "what is "object" in our knowing describes the thinking and feelings we say we have" (p 53). A learner

who is initially focussed on learning facts (subject – the facts dominate their thinking) needs to adjust the focus to learning about the justification for those facts or the process that brought the current commitment to those facts or way of looking at the subject area (object – the facts are now under the control of the learner's thinking). The best forms of teaching involve more than "transmitting knowledge" (Bowden & Marton, 2003). Teaching at its best enables knowledge to be transformed and extended.

In Perry's (1968; 1988) framework, these transformations are described as different ways of understanding the nature of knowledge. His framework identifies stages in the development of a learner. A learner moves from seeing knowledge as dualistic (right and wrong) through multiplicity (multiple opinions with authority able to make judgements) to a stage of being committed to knowledge based on the supporting evidence.

The concept of what is to be learnt or of learning may not be the starting point for learning. The first question may not be "what is your concept or understanding of xyz (the object of learning)?" It may come back to "are you aware of xyz? Does xyz as a term or name of something have any meaning?" The relating of "what is a program(me)?" back to concepts of programme familiar to the learners provided a basis for the learners to build a conceptual understanding of the computer programming task (see section 2.3.3, page 38). The students may have entered the class with limited conceptions on the nature of a computer program but their prior conceptions of a programme were used to address the issue of their awareness and helped build a conception that better enabled them to approach the programming task and the related learning.

Armour (2000b) has defined orders of ignorance. At the third order of ignorance, the person not only does not know something, but they do not know that they do not know and have no knowledge of a way to find out. Lister (2003) has translated these to learning. If the learner has no knowledge of something, then the learner does not know of its existence and does not know that they do not know about it. The first step in learning about that thing is to become aware that they do not know about it, that is, they need to have some awareness of its existence. Armour and Lister are talking in relation to software development and teaching but these principles equally apply to learning.

3.4 Changing of concept awareness and processes

Marton and Booth (1997) argue that we learn by increasing our differentiation and integration of our experienced world. A key concept here is our experienced world. It is not a world which we have to find or discover; it is the one that we are part of and one that we experience in ever-increasing detail (p 138).

It is variations in the ways the world is experienced that increases awareness of its richness and differences. Part of those variations are the experiences that are shared with others. Marton and Booth talk of these as mediated experiences, shared culture, and shared understanding (p 139).

As more knowledge is gained, changes occur in the person-world relationship. "Learning is mostly a matter of reconstituting the already constituted world" (p 139). When our conception of the world, our worldview, is shaken we have to reconstitute our frame of reference. This allows us to see the world in new ways and to gain new insights that possibly did not fit our previous conception of the world. Without these shifts, we may never understand some things that happen in our world.

At the foundation is the concept that a person and their world are inseparable. It is not possible to conceive a person's world without them or to perceive a person without their world (Marton & Booth, 1997, p 138). If a person has no concept of something then it is almost impossible to teach them about that thing (see also Armour, 2000b). An example given by Marton and Booth is teaching reading and writing to children who had no idea what the concepts of reading and writing might mean or the value of them. Because the children had grown up in "an environment in which the written world was of little importance", they simply "did not understand the idea of reading and writing" (Marton & Booth, 1997, p 140). In order to learn, they needed to be helped to see the relevance of reading and writing (pp 182-183).

Marton (2000) argues for a non-dualistic ontological position as the basis for phenomenographic research (see section 3.7, page 61); that is that the subject and object are *not* separate, "the subject's experience of the object is a relation between the two" (p 104). He says:

> "From a non-dualistic ontological perspective there are not two worlds: a real, objective world on the one hand, and a subjective world of mental representations on the other. There is only one world, a really existing world, which is expressed and understood in different ways by human beings. It is simultaneously objective and subjective. An experience is a relationship between object and subject, encompassing both. The experience is as much an aspect of the object as it is of the subject" (p 105).

The observer's experience is their reality. There is no other reality for that individual outside what they observe and interpret in their current frame of reference. What a person reports is their experience of a particular set of subject-object awareness relationships. In order for a person to learn, their way of experiencing or being aware of the phenomenon has to change.

Marton (2000) argues "that an object of experience is not independent of the way in which it is experienced" but this does not mean that "the object is identical with the way in which it is experienced" (p 105). Awareness of a phenomenon is dependent on there being an observer. Each observer may become aware of the phenomenon in different ways. There are different ways in which a phenomenon can be experienced and these different ways are related. If a group of different ways of experiencing a phenomenon are identified and categorised, then they form a set of categories of description, called the "outcome space," which is a "synonym for 'phenomenon'". The set of categories of description or the "outcome space" provides the complete understanding as experienced so far. This theme is expanded in the next chapter when the approach to research is examined.

Ramsden (1988b) contends that learning is about conceptual change or changes in understanding. Perry (1968; 1988) has described such changes in terms of a transition from dualistic understanding of knowledge (the knowledge is either right or wrong) to a position of commitment to knowledge based on sound reasoning. Mezirow &Associates (2000) argue for changing frames of reference. The development of a conceptual framework and an ability to apply the thinking processes of a subject area are critical components to the development of a learning process that can be consistently used for learning within a subject area domain.

To enable the transition to a conceptual framework consistent with the subject area and to the thinking processes of the domain involves the lecturer or learning facilitator helping the learner to uncover their current conceptions and thinking processes (Ramsden, 1988b). As these are uncovered, they can be challenged and revised through counter examples or variations (Bowden & Marton, 2003). Covey (1990) in talking about the habits of effective people emphasises the need to seek to understand before seeking to be understood. If the teacher wants to help a learner make a transition to a new way of thinking, then they need to start by attempting to understand what the learner's current thinking is and why the learner might see it as difficult to look at the problem in a different way.

This process of examining current conceptual understanding and adjusting those perceptions can be lecturer directed but learner centred (Angelo & Cross, 1993). It must progress the learner from dependence on the lecturer to independence. The learner "must learn to take full responsibility for their learning" (p 4). Some would argue that it should take the learner further to enable them to operate as an interdependent learner (Covey, 1990; Leach, 2003). As an interdependent learner, the learner should be able to operate in collaborative learning environments, participate constructively in group assessments and class discussions, and utilise peer teaching and assessment strategies. Bowden and Marton (2003) argue that in order to become aware of "taken-for-granted" aspects of knowledge, to explore variations, and to understand others' ways of seeing, learners and teachers should learn from each other and the wider community within the subject domain (p 14). The learner's conceptualisation of the meaning of learning must provide them with a tool set or process of learning that enables ongoing effective learning.

3.5 Teacher's perception of teaching

The emphasis on changing conceptions or ways of thinking is not always the understanding promoted by teachers. The teacher's perception of the subject matter and of teaching is also relevant to improving learning (Trigwell & Prosser, 1996a, 1996b; Trigwell, Prosser, & Waterhouse, 1999: Trigwell, 1997 #3719). Trigwell and Prosser (1997) describe six conceptions of science teaching. These are:

"Conception A: teaching as transmitting concepts of the syllabus.

Conception B: teaching as transmitting the teachers' knowledge.
Conception C: teaching as helping students to acquire concepts of the syllabus.
Conception D: teaching as helping students to acquire teachers' knowledge.
Conception E: teaching as helping students to develop conceptions.
Conception F: teaching as helping students to change conceptions" (p 246).

Based on the previous discussion, these six conceptions of teaching form the basis of the conceptual understanding of teaching. Each portrays something of the nature of teaching and learning.

Referring back to Ramsden's model (see Figure 3.1), what the teacher does also has influence on the learner's perception of the task and as a consequence the learner's approach to learning. The teacher provides a context for learning which has some influence on the learner's perception of the task requirements. If learning is seen as changing the learner's conception of the phenomenon, then the teaching focus would need to be on the higher two conceptions of teaching.

Marton and Booth discuss a pedagogy of awareness (1997, Chapter 8). They describe pedagogy as involving "being able to take the part of the other, being able to judge the success of achievement and being able to adapt the intervention according to a perception of its value to the cause" (p 167). This statement emphasises a number of key points.

- It takes the *part of the other*. It seeks to identify with the learner and work from the learner's perspective.
- It is *goal oriented*. There is some way of assessing success.
- It *adapts the interventions*. It is not a fixed set of learning activities. It recognises the progress, or lack thereof, of the learner and adjusts the activities to work toward success.

Based on the theory of desiring deep approaches to learning, experiments have been performed to determine an appropriate pedagogy. The methods that involve "pointing out" (introducing questions aimed to foster deep approaches to learning) have not produced the desired results. The learners focussed on the questions rather than on the text. In effect, introducing elaborate instructions reduced the possible range of awareness options for the learner. The learners looked at the material in fewer ways. The learners focused on the "act of learning" and not the "object of learning" (p 171).

Directing the "how" took the learners away from the content or the object of learning. Another experiment used the object of learning to explore how to learn. This produced deeper approaches to learning.

Marton and Booth argue that

> "teachers mould experiences for their students with the aim of bringing about learning, and the essential feature is that the *teacher takes the part of the learner*, sees the experience through the learner's eyes, becomes aware of the experience through the learner's awareness. If we, in accordance with our nondualistic stance, consider the learner to be internally related to the object of learning, and if we consider the teacher to be internally related to the same object of learning, we can see the two, learner and teacher, meet through a shared object of learning" (1997, p 179).

The authors later talk about using learners' questions as a point of departure (p 181), that is, the teacher utilises learners' questions as the starting point for learning and teaching. They sought to understand the learners' questions and build a conception of the subject from the learners' current perspective and understanding.

The authors state that "the teacher makes the learner's experience of the object of learning into an object of her own focal awareness: The teacher focuses on the learner's experience of the object of learning" (p 179). The teacher is learning about "the learner's experience of the object of learning". The authors argue that in order for the teacher and learner to have a meeting of awareness, a shared object of learning, the teacher needs "to take the part of the learner" and construct "a relevance structure" and through using variations in ways of experiencing the phenomenon form "the object of study" (p 179). If the pattern of variations in awareness is not of relevance to the learner, then there will be no or limited learning.

Booth (1997) in discussing the use of phenomenography to improve learning and teaching, implies that it is not possible to pursue all classroom problems using phenomenography. However, when a problem with a learner's understanding does occur, the teacher needs to be prepared to explore the issues through appropriate

questions to uncover their understanding of the problem. The teacher is then in a position to explore variations in understanding and approaches to problem solving.

When the learners give the impression of understanding the topic and what is expected of them for the assignment, it is easy to have not heard what the learners' perceptions really are. The teacher may need to ask questions until the teacher and students have a common understanding. This would uncover the perceptions that the learners have.

3.6 The nature of awareness

In the preceding discussion, reference has been made to the learner's experience or awareness of the phenomenon under study, the object of learning. What constitutes a person's experience or awareness of a phenomenon? Pang says "experiencing something is related to how a person's awareness is structured" (2003, p 148). A deer in a forest may not be experienced because it blends with its environment; likewise, a pebble on a concrete path. Unless the observer recognises some variation between the phenomenon and its context then it may not be seen, that is, they are not aware of the presence of the phenomenon.

Becoming aware of a phenomenon does not bring recognition of that phenomenon unless there is some familiarity with its structure and the features of its structure that distinguish it from similar phenomenon. A young child having learnt about ducks, sees an owl for the first time but not being familiar with the characteristics that distinguish an owl from a duck, calls the owl a duck. In order to help the child learn the difference, the parent might help the child to see the characteristics of an owl that are different from the duck. As the child learns the critical characteristics, they begin to distinguish the differences. The structure of the bird helps bring meaning.

Marton and Booth (1997) argue that in order to identify a phenomenon there are two aspects that interplay. These are the structural aspects and the referential aspect. The structural aspects relate to identifying the whole and its relationship to its surrounding context (external horizon), and the parts of the phenomenon and the relationship between them in a simultaneous manner (internal horizon). The referential aspect is the overall meaning assigned to the phenomenon. Pang says a phenomenon has to be "discerned as a particular thing and assigned a meaning to it" (2003, p 148). Further, he

says “structure presupposes meaning and meaning presupposes structure. Structure and meaning thus mutually contribute to each other in the act of experiencing” (p 149).

To become aware of a chain link on a wooden cycle track, for example, the observer has to be aware of the object’s structure, and how the object differs from its context. The chain link is metallic in nature and its form separates it from its context. Its internal structure brings meaning that enables it to be recognised as a chain link and not simply as a piece of metal or a screw.

The influence of context on understanding or attaching meaning is illustrated by Marton and Booth (1997) in relation to a sequence of numbers (Example 5.1, pp 88-89). The sequence of numbers may be seen as individual digits or as a sequence of numbers based on some method of calculation, or maybe as a financial amount. Given a particular context, such as an IQ test, the observer is more likely to “discover the rule as they looked for hidden relationships” (Pang, 2003, p 149). If it was in a gambling context or an economic setting, then the sequence may be understood as a monetary amount. The meaning attached to the phenomenon is as much dependant on context as it is on the internal structure.

In the chain link example, because it is on a cycle track, it may be assumed to be a chain link from a bicycle. However, if the observer was aware that the motor pace motorbike had just broken its chain then the chain link may be discerned as belonging to the motorbike. Meaning is related to structure and context.

3.7 A phenomenographic perspective

When faced with a phenomenon do all observers see and understand the phenomenon in the same way? As was argued earlier (see section 3.4, page 55), there are not two worlds, an objective world and a subjective world of mental representations (Marton, 2000). The world is what we experience and describe. It is both objective and subjective. We cannot separate ourselves from our experiences. The phenomenon may exist but our experience of its existence is how we understand it; that does not make the phenomenon the equivalent of our experiences, but our experiences limit our understanding of the phenomenon. A phenomenon is “a complex of the different ways in which it can be experienced” (Marton, 2000, p 105). In phenomenography, the

researcher endeavours to develop a set of categories of description, which together represent the phenomenon as experienced by a group of observers. The categories of description are logically related to each other and form the 'outcome space' relative to that phenomenon. Marton says that the 'outcome space is a synonym for 'phenomenon' (p 105).

Marton says that an object of awareness is embedded in a thematic field. The thematic field is "those aspects of the experienced world that are related to the object" which is presented (p 110). The awareness of the phenomenon under study is the theme and any given theme can be seen against a background of different thematic fields. Marton argues that "The theme appears to the subject in a certain way, it is seen from a particular point of view. The specific experience (or conception) of a theme – or an object … - can be defined in terms of the way its component parts are delimited from, or related to, each other, and to the whole" (p 110). Other things around that theme are not taken into consideration. The observer simply does not see the other things as being related to the theme. If these other things were seen, then the observer may see the object or phenomenon in a new way and provide the subject with new ways of solving problems related to that phenomenon.

Marton talks of the structure of awareness in terms of the things that are to the fore, figural, and thematised. Other things are in the background and are tacit and unthematised. Marton argues that it is not a case of two categories: explicit-implicit, figure-ground, thematised-unthematised, instead there are different degrees of awareness. A subject's awareness of an object is influenced by what they are focally aware of in observing the object.

A person reading a text will take different things from the text in each reading depending on their thematic field at the time of reading. The reader will see those things that reflect their current focus of attention and not necessarily the key concepts that the author wanted the reader to take from the text. The same applies to the learning context. What the learner is aware of will depend on the learner's focal awareness within the thematic field, their perception of the subject or task. Consequently, they may not see what the teacher wants them to see and learn from the lesson or materials. Influencing the learner's focal awareness within the thematic field will change what they learn and how they approach the learning task.

The categories of description obtained in a phenomenographic study reflect the degree of awareness of the thematic influences on the subjects with respect to the object or phenomenon. The thematic field serves as a context for the theme and influences our understanding and awareness of the theme. The categories of description endeavour to highlight the themes in the thematic field, which influence that category and the attributes of the object, which become apparent as part of that thematic field. It is also argued that the range of categories of description or ways of experiencing a phenomenon is limited for any given phenomenon (Marton & Booth, 1997, p 126). The aspects, which help distinguish the different categories of description from one another, are considered critical aspects in understanding the phenomenon.

Chik and Lo (2003) talk about simultaneity or the simultaneous aspects that the learners become aware of in the classroom. They say, “More powerful ways of understanding amount to a simultaneous awareness of those features that are critical to achieving certain aims” (p 89). They talk about *aspect-aspect relationships* being when “discerned features are seen as aspects of something”. *Part-whole relationships* relate to the discerning of the linkage between the object and its parts. An object has features that may or may not be discerned. According to Chik and Lo when these features are discerned they are seen as aspects. They argue that

> “learners will learn more effectively if the teacher is able to consciously structure the presentation in such a way as to bring out clearly the critical features of the object of learning, as well as their relationship to the object of learning and to each other.” (p 89)

A phenomenographic study endeavours to gather a range of different statements of awareness and to analyse these to uncover the different categories of description based on the different levels of awareness or the variations in the observations and descriptions of the subject. Each of the descriptions is a second-order description of the phenomenon in that it describes a way of experiencing the object. It answers questions of the type “What do people think ... is?” or “How do people understand ...?” rather than questions of the form “What is ...?”

Pang (2003) contends that phenomenography has moved from attempting “to describe different ways of experiencing various phenomena” and is now addressing questions

related to “what are the ways of experiencing something” and “What is the actual difference between two ways of experiencing the same thing?” He argues that there is a move toward theoretical questions about the nature of differences. He argues:

> “Every phenomenon that we encounter is infinitely complex, but for every phenomenon there is a limited number of critical features that distinguish the phenomenon from other phenomena. What critical features the learner discerns and focuses on simultaneously characterises a specific way of experiencing that phenomenon. But a feature cannot be discerned without experiencing variation in a dimension corresponding to that feature” (p 148).

The theme is composed of a number of different aspects or dimensions of variation that may be present or seen in different ways within the theme. The presence or variation in the aspects helps distinguish the categories of description (Cope, 2000, 2002a).

The earlier forms of phenomenography focused on describing the variation that the researcher could detect (experience) based on the expressed differences in the experiences of people of various phenomena. Pang argues that the focus is now on the “variation in various aspects of the world around as experienced by the learners” (Pang, 2003, p 148).

In an educational context, this is often how the learners are aware of the phenomenon but from a teaching perspective, the teacher wishes to open up a ‘space of learning’ by presenting variations in critical aspects that open the learner to alternative ways of being focally aware of those aspects and foster deeper understanding the phenomenon.

3.8 Teaching as opening up the space of learning

Endeavouring to change a learner’s perceptions may not be easy. Bowden and Marton (2003) contend there is a need to experience variation so that discernment can develop. They argue that the things that are constant are easily overlooked and ignored. The learner becomes used to the constants and they become taken for granted. Variations make the differences more noticeable and encourage the learner to look at different ways of viewing what is being learnt. This could be achieved in an environment of

learning from each other where all seek to become aware of each other's ways of seeing. This does not mean coming to agree with their way of seeing but rather that there is acknowledgement of the different ways of seeing. This process may be difficult as "the more fundamental a layer of knowledge is, the less visible it is" (Bowden & Marton, 2003). This fundamental knowledge is taken for granted but can become visible as we endeavour to understand other ways of seeing and reasoning.

How much does a teacher's approach to the discipline impact the way learners construct knowledge? In reporting the research on this theme, Marton and Booth contend that teachers construct objects of study for their learners (1997, p 177). These constructed objects of study are different based on the conceptions of what is to be studied that the teachers hold. Marton and Booth provide an example related to the teaching of physics.

The variations presented are a teacher who sees physics as a process of enquiry, a teacher who sees physics as a set of explanatory theories, and a teacher who sees physics as calculations. Each of these represent different objects of study for the learners. These teachers are moulding an object of learning by the way they present their approach to the subject. If the focus is on calculations then the learners will focus on memorisation and application of the formulae. If the focus is on a process of enquiry then the learners will focus on the process and see the theories and formulae as the result of that process of enquiry.

Marton and Booth (1997) contend that learners experience the object of learning sensuously. Even though it may not be a physical object, the learner can picture it and describe it in physical terms. This sensuous experience of the object flows from the learner's representation or understanding of its variations. It takes a form that expresses their understanding of the object.

Marton and Booth say, "learning is the constitution of the object of learning" (p 161). In their definition of learning, learning is the change in the person-phenomenon relationship. At the beginning of a learning episode, the learner may have no, or limited, awareness of the object of learning. As the learner adds distinctions, the learner's awareness of the object increases with the result that the object of learning is brought into being. The learner's representation of the object of learning is his description and it

will vary from other learners' descriptions of learning. If no one has a description for the object of learning, then there is no awareness of the object in the real world.

Marton, Runesson, & Tsui (2003) contend that "the object of learning is a *capability*, and any capability has a general and a specific aspect" (p 4). The general aspect relates to the cognitive skills or "acts of learning carried out". The 'act of learning' is seen as part of the 'how' aspect of learning which also includes the content of learning in the form of the indirect object of learning (Marton & Booth, 1997, pp 83-84). These are called general aspects because they can be applied to any object of learning. The specific aspects relate to the thing or subject upon which the acts of learning are carried out, the *direct* object of learning or the 'what' aspect of learning. Marton et al. (2003) say that "The *object of learning* can be defined by its critical features, that is, the features that must be discerned in order to constitute the meaning aimed for" (p 22). A *critical feature* is a way of "distinguishing one way of thinking from another" (p 24). In terms of an object of learning, it differentiates effective and ineffective ways of experiencing that object.

In order to instantiate an object of learning, "a certain pattern of variation" needs to be encountered regardless of how the teaching is approached. If this pattern of variation is not experienced or comprehended by the learner, then the learner may struggle to comprehend the object of learning as intended.

The learners often focus on the *specific aspect* or *direct* object of learning. The teacher should be focused on both. The teacher's focus is on the *intended* object of learning. The *intended object of learning* is the object as seen from the teacher's perspective (Marton et al., 2003, p 22). The teacher structures the conditions of learning in order to make the learner aware of the object of learning. What the learner becomes aware of is the *enacted* object of learning. The *enacted object of learning* is what is realized in the classroom "in the form of a particular learning space" and is what is seen by the researcher (p 22).

What the learner takes away from the learning situation is the *lived object of learning*. The *lived object of learning* is "the way that the students see, understand, and make sense of the object of learning. It is what they take away from the lesson" (p 22). What is learnt depends on the "condition of learning" (p 22). This is dependant on what is

actually happening in the classroom. These conditions make it possible for learners to "learn certain things" but cannot *cause* learning.

Marton et al. (2003) describe four patterns of variation. These are contrast, generalization, separation, and fusion. They are defined as follows:

- **Contrast** is required "in order to experience something, a person must experience something else to compare it with" (p 16).
- **Generalization** is required with respect to the object of learning (p 16). It is not simply enough to see the object; we need to see variations in the use of the object to comprehend it fully.
- **Separation** of an aspect from other aspects is required. The object needs to be looked at from different angles. In order to discern an aspect, the aspect that is being examined "must vary while other aspects remain invariant" (p 16).
- **Fusion** is where "several critical aspects" need to be considered together. Those aspects must be experienced simultaneously (p 16).

As well as defining the object of learning, Marton et al. (2003) define the space of learning. This defines the learning options available. It "refers to the pattern of variation inherent in a situation as observed by the researcher" (p 21). This definition is based on a research perspective. It defines "a necessary condition for the learner's experience of that pattern of variation unless the learner can experience that pattern due to what she has encountered in the past" (p 21). A space of learning is created by opening up any number of dimensions of variation. A dimension of variation relates to the variation with respect to an aspect of the object of learning. This variation may be through contrast with other objects in the learning environment or through variations in the frame of reference or ways of looking at the object of learning. The range of what can be learnt is defined by the space of learning and the range of variations presented in the learning space. The space of learning equates to an experience space in that it opens up ways of experiencing the object of learning (p 25).

Ko and Marton (2003) focus on the use of variation in Chinese teaching. They describe three principles (p 57-58) which are:

A) "Widening of the concept by blocking out features erroneously assumed by students to be critical features of the concept". This variation is argued to be *generalisation*.
B) "Making it possible for the students to discern what is tacitly understood, by means of constructing noninstances". The variation is a focus on *contrast*.
C) "Making it possible for the students to make distinctions between cases that they treat as members of the same set". Again this variation is classified as *contrast* but it is different in the sense that it is focussing on subtle differences.

Runesson and Mok (2003) discuss discernment, experience, and meaning (pp 63-64). They contend that the way a person experiences something "is the way the individual understands it". These ways of experiencing are different for different people.

From a teaching perspective, to determine whether the required learning has occurred, the teacher needs to uncover how the learner has experienced the object of learning. To foster a particular way of experiencing, the teacher needs to provide variation that will give the potential for experiencing the object of learning in the desired way. If the experience that is given primarily relies on definition then the teacher should not expect much more than memorisation and recall. If more is desired then the teacher needs to look for variations that uncover those deeper experiences.

3.9 Educationally critical aspects

The critical aspects of an object of learning are those features that enable discernment of the object of learning, that is, they enable the learner to develop an appropriate and deep understanding of the object of learning.

As discussed above, a phenomenographic study focuses on uncovering categories of description that are based on different ways in which observers are aware of the phenomenon. These categories are defined in terms of dimensions of variation, variations in a combination of critical aspects, used to describe the phenomenon (see left hand side of Figure 3.3). In contrast, when teaching is analysed, the variations around critical aspects are identified to determine the *enacted object of learning*, and the *space of learning* is identified (see right hand side of Figure 3.3).

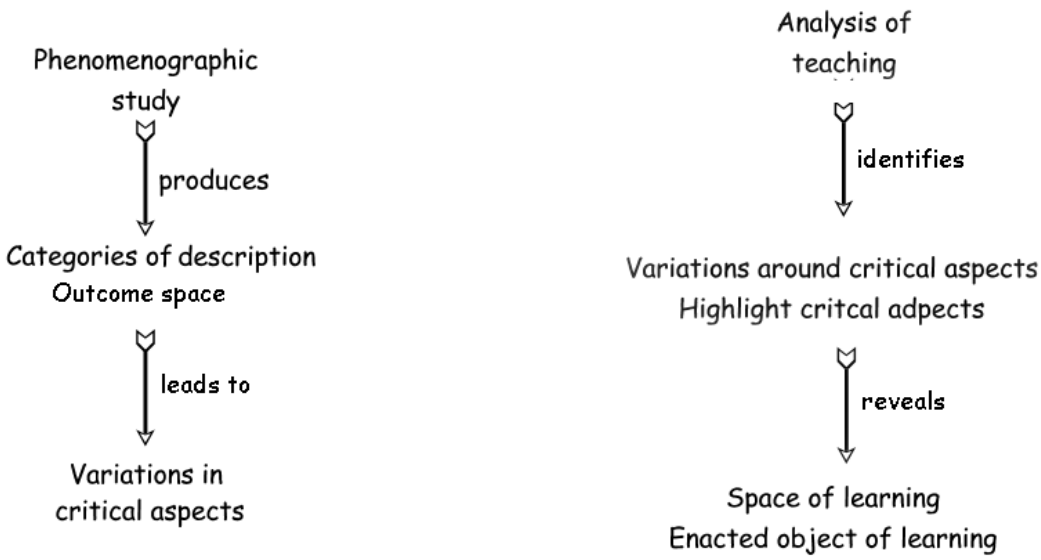

Figure 3.3: Variations in phenomenography and variation theory

The *outcome space* from a phenomenographic study is often depicted as a table that describes each category of description in terms of its referential aspect and its structural aspect (Cope, 2000, p 115, 2002a, p 74). Cope has taken this a step further and divided the structural aspect into descriptions in relation to the internal and external horizons. He has used a second table to describe the dimensions of variation or critical aspects (2000, p 167; 2002a, p 75) (See Figure 3.4). A column within the structural aspect area represents a dimension of variation or a critical aspect. A row represents a category of description or a level of understanding in Cope's terminology. A cell represents the variation of an aspect with respect to the specific category. An aspect does not need to be present in all categories and its variation may be the same in a number of categories. In Cope's table, aspect 5, "User interface and control subsystems" is not present in categories 1 through 6. Aspect 1 has the same variation ("Complex") for categories 2 through to Deep, and a different variation ("Simple") for category 1. No two categories should have the same variations for all the aspects identified.

	Technical aspects							Social aspects				Relationship between technical and social aspects	
	1	2	3	4	5	6	7	8	9	10	11	12	13
Level of understanding	Nature of queries supported	Nature of database	Who can instigate changes to data?	Location and nature of processes	User interface and control subsystems.	Relationship between user interface, control, processing and database subsystems.	No. of users served by IS.	People other than the direct operator as part of the IS?	Number of organisational functions involved	Attribution of meaning as a human activity	Informal information flow	System served/ system serving relation between social and technical aspects	User interface as physical link between social and technical aspects
Deep	Complex	Dynamic	Multiple users	Within and beyond the computer. Computerised and manual.	Accepts and validates input. Directs processing. Displays output.	Ability to conceptualise flow of control and data between subsystems.	Multiple	Yes	More than one.	Only people can interpret input to and output from the IT	An undesigned source of information important in decision making	IT should support, rather than dictate the nature of, organisational activity	User interface design influences how meaning is attributed
6	Complex	Dynamic	Multiple users	Within and beyond the computer. Computerised and manual.			Multiple	Yes	More than one.	Use of output	Some formal interaction between people		
5	Complex	Dynamic	Multiple users	Within and beyond the computer. Computerised and manual.			Multiple	Yes	One	Use of output	Some formal interaction between people		
4	Complex	Dynamic	Multiple users	Within the computer. Computerised.			Multiple		One				
3	Complex	Dynamic	User	External to the person. May be computerised or manual.			One		One				
2	Complex	Static	Not user	External to the person. May be computerised or manual.			One		One				
1	Simple	Static	Not user	External to the person. Manual.			One		One				

Table 3: Comparison of the various levels of understanding of the concept of an IS (Cope, 2000b)

Figure 3.4: Cope's table identifying critical aspects (Cope, 2002a, p 75)

In analysing teaching, the space of learning is identified by the variations present. This is not necessarily aligned with any category of description. To identify the space of learning, the variations used, in teaching, with respect to each aspect need to be identified. The space of learning may not correspond to a specific category nor cover all the variations in a specific category. However, it is unlikely that a learner will develop an awareness of the phenomenon at the deepest level if some of the variations present in that category are not included in the space of learning.

The hypothesis behind the present research is that if the critical aspects for the categories of description for a phenomenon as recognised by practitioners in the field can be identified, then these can be used as the basis for planning teaching. It has been argued that learning involves a change of conception and that to achieve a change of conception a space of learning has to be created based on variations in critical aspects.

The basis for proving this hypothesis is that having identified the categories of description and the critical aspects, these can be used to identify the desired depth of understanding as a particular category of description and the critical aspects used to plan variations for teaching.

Booth (1997) has argued that the teacher should be aware of the content of teaching both “as it *is* and as it *should be* understood by the learners” (p 137). She also argues that the teacher has to “identify those *educationally critical aspects* which might otherwise be taken for granted” (p 137). This would seem to support the view presented in Figure 3.5.

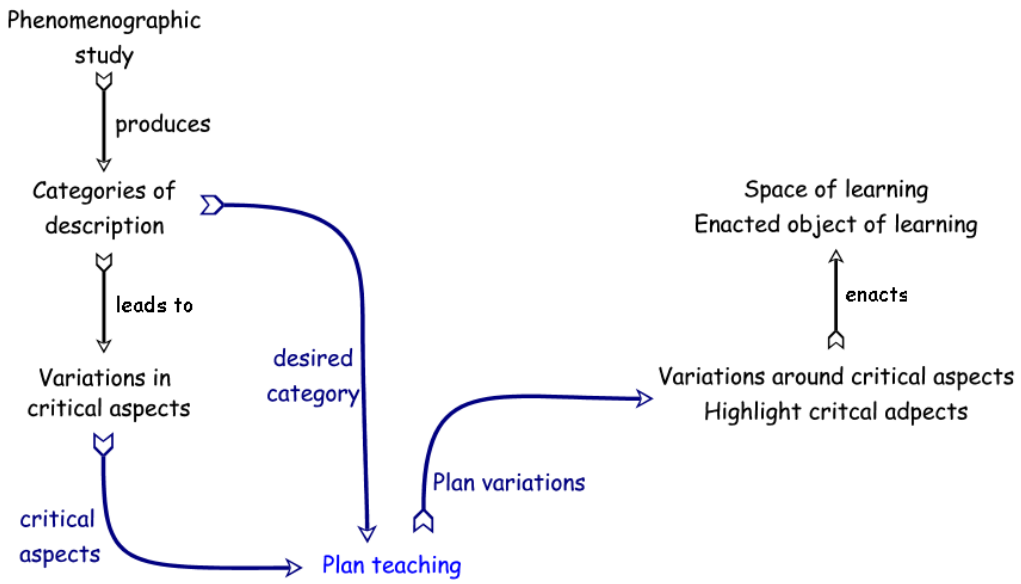

Figure 3.5: Planning teaching based on categories of description

Booth also places emphasis on ensuring "that the learners *reveal their experience of learning*, in terms of the various ways in which they understand that which is learned" (p 137). She argues that this should be done so that the learner becomes aware of the various ways of experiencing the phenomenon. But what are the *educationally critical aspects*? Booth implies the educationally critical aspects are those aspects taken for granted but essential to enabling the students to understand the phenomenon in a way that is appropriate for the desired learning outcome or intended environment of use. Cope (2000) defines *educationally critical aspects* as "those aspects of the experience of learning about a phenomenon that are critical to achieving an adequate depth of understanding" (p 6).

In Cope's study to determine the educationally critical aspects of learning information systems, Cope interviewed undergraduate students to determine their categories of understanding information systems. He then compared this with an analysis of relevant information systems education literature. This provided him with a set of critical aspects.

Cope (2002a) argues that "From the perspective of a structure of awareness, a critical difference between the level of understanding" (i.e. a critical variation in an aspect) is implied when one of the following occurs:

- "the nature of the understanding of an aspect" changed "in a significant way towards the value associated with the deep understanding";
- "an aspect […] that is an integral aspect of the deep understanding is not part of the shallower levels of understanding";
- "new or stronger relationships exist between aspects"; or
- "the nature if the boundary between the internal and external horizons is different" (p 72).

Cope (2002a) identifies the critical differences in the understanding of an aspect based on these variations and argues that teaching should be planned around uncovering these critical differences of the aspects. For his subject matter, he charts a path of learning based on the critical differences in the aspects.

3.10 The nature of a phenomenon

Phenomenography uncovers the variations in the way that people are aware of a phenomenon. When the phenomenon under investigation is learning, it is argued that the phenomenon has a 'how' and a 'what' aspect (Marton & Booth, 1997, pp 83-86). The 'how' aspect is referred to as "the experience of the way in which the act of learning is carried out" and is represented by "the act of learning" and "the type of capabilities that the learner is trying to master" (the "indirect object") (p 84). The indirect object of learning represents what the learner perceives is to be learnt. The 'what' aspect is "the direct object of learning: the content that is being learned" (p 84).

This relationship is represented in the diagram (Figure 3.6). The 'act', 'indirect object', and 'direct object' can be analysed with respect to the learner's awareness of the phenomenon. The learner's awareness is composed of referential and structural aspects (pp 86-88, see also section 3.6, page 60 of this thesis).

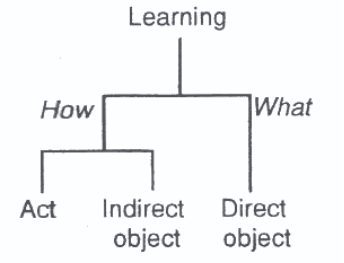

Figure 3.6: The 'how' and 'what aspects of learning

In the context of the present research, the direct object of learning is 'object-oriented programming'. Booth (1992) contends that programming has a 'how' and a 'what'

aspect (p 71). She argues that learning to program involves both ‘learning theoretical concepts” and “learning to write programs to solve problems” (i.e. programming involves both an ‘act’ and a ‘what’, the program). The critical aspects for both the ‘how’ and ‘what’ of learning need to expressed in the space of learning.

As was seen in the discussion in the previous chapter on the nature of the problem of learning to program (see section 2.3, page 36), there is considerable confusion with respect to the method of teaching object-oriented programming and whether object-oriented programming requires a different approach than teaching imperative programming. In order to determine the desired level of teaching, there is a need to understand the awareness of practitioners with respect to object-oriented programming.

3.11 Conclusion

Improving learning requires understanding the concepts that a learner has of both the subject area and of learning (Ramsden, 1988b). It involves uncovering what learners already know and providing challenges to their existing knowledge that will enable transformation of their frame of reference or their view of the nature of knowledge (Kegan, 2000; Mezirow, 2000). This transformation has to occur within the context of the subject area so that subject area processes are applied (Ramsden, 1988a). Thus the possibility for this transformation can occur if variations that highlight the critical aspects with respect to the phenomenon are used to open up the space of learning (Booth, 1997; Cope, 2002a).

A phenomenographic study provides a set of categories of description that identify different ways in which people are aware of a phenomenon. These categories lead to the identification of critical aspects with variations in these critical aspects highlighting the distinction between the categories. It has been argued that with knowledge of these categories and the critical aspects, the teacher is able to plan a teaching strategy.

The difficulty for the teaching of object-oriented programming is that limited knowledge exists of the different categories of description and of the critical aspects related to those categories. Phenomenographic research conducted has focused on specific phenomena such as class and object (Eckerdal & Thuné, 2005). Research into expertise does tell us that expert object-oriented programmers see patterns in the code,

and use plans or patterns in the writing of code. It is unclear whether these patterns are those represented in design pattern books (Freeman, Freeman, Sierra, & Bates, 2005; Gamma et al., 1995; Gestwicki, 2007; Shalloway & Trott, 2005) or whether they are the elementary programming patterns (Beck, 1997, 2007).

The following questions were developed based on the above requirements and formed the foundation for the planning of this research project.

1) How does the awareness of an object-oriented program vary among the 'practitioners'?
    a. What are practitioners' ways of experiencing object-oriented programming?
    b. What are the "critical aspects" that distinguish the different variations in awareness expressed by 'practitioners'?
2) What spaces of learning are opened up by introductory programming texts with respect to the 'critical aspects' arrived at from the previous questions?

The next chapter discusses the planned strategy for gathering data and the approach to analysis used in the current research.

Chapter 4. Research method

In the previous chapters, it has been argued that in order to improve learning, it is necessary to change the conceptions that the learner has with respect to the subject matter being learnt. The conceptions of the learner can be changed by helping the learner become aware of appropriate variations that focus on the critical aspects of the phenomena being studied. It has also been argued that with respect to the learning of programming, and in particular object-oriented programming, there is limited understanding of the possible ways of being aware of object-oriented programming, and the critical aspects that are effective in the subject area. At the end of the last chapter the following questions were proposed for the current study:

1. How does the awareness of an object-oriented program vary among the 'practitioners'?
    a. What are practitioners' ways of experiencing object-oriented programming?
    b. What are the "critical aspects" that distinguish the different variations in awareness expressed by 'practitioners'?
2. What spaces of learning are opened up by introductory programming texts with respect to the 'critical aspects' arrived at from the previous questions?

This chapter examines phenomenography as a research approach to explore these questions and to uncover appropriate answers. It describes the research design. It explores and discusses methods of data collection and analysis for the use of the phenomenographic research approach, the possible difficulties with using the proposed methods, and how these difficulties are resolved in the current study.

4.1 Philosophical foundations of phenomenography

In this section, the theoretical foundations for the current study are examined. This examines the epistemology, ontology, and the basis for ranking the categories of description uncovered. Although the current study utilises techniques that have their foundation in phenomenography, there are other theories that influence the approach taken.

Phenomenography is a research approach that has primarily been deployed within the educational context. The approach was developed within the educational setting and most research conducted has been related to educational issues (Bowden & Green, 2005; Bowden & Walsh, 2000; Marton & Booth, 1997). The research of this thesis has its origin within education and its target application in education within the computer science discipline. The result of a phenomenographic study is a set of categories of description that define an outcome space for the phenomenon. Each of the sets of categories of description may also be understood in terms of variation in critical aspects (Cope, 2000, 2002a; Marton, 2000; Marton & Booth, 1997; Marton & Pong, 2005).

Variation theory is also based in education with its key terminology describing a space of learning and the object of learning (Marton & Pang, 1999, 2006; Marton et al., 2003; Marton & Tsui, 2003; Pang, 2003; Pang & Marton, 2005). In using variation theory to analyse teaching, the emphasis is on the variations around critical aspects (Marton & Pang, 1999; Marton et al., 2003; Pang, 2003). Since the critical aspects are the key to developing awareness in accordance with a particular category of description, the teaching needs to be highlighting those aspects. This is achieved by using appropriate variations that will highlight these aspects.

Marton defines the external horizon as “how it is delimited from and related to a context” and the internal horizon as “how its parts are delimited from and related to each other and to the whole” (2000, p 113). If an observable aspect is related to the difference between the phenomenon and its surrounding context, then the observable variations are in relation to the external horizon. For example, a deer standing still in the forest may not be discerned through the characteristics of its structure (internal horizon) by the observer unless there is a point of differentiation or variation between the deer and the forest (the context in which it is standing – external horizon). Nothing about the characteristics of the deer is observable unless contrasted with its context. If an observable aspect is related to the structural characteristics of the phenomenon, then it is in relation to the internal horizon. In the context of the deer, this would represent the relationship between the head and the body. A deer is recognised by its structural characteristics and not simply by how it is differentiated from its surrounding context (Marton and Booth, 1997, pp 86-87).

Any expression of an epistemological or ontological position with respect to phenomenography, taken from a phenomenographic perspective, contains elements of awareness of these phenomena. Analysing these expressions of awareness would uncover a deeper understanding of epistemology and ontology with respect to phenomenography. Such a study is beyond the scope of this thesis. Here those descriptions that inform the researcher's awareness are discussed and the position taken expressed.

4.1.1 Ontology of phenomenography

The core foundation for phenomenographic ontology is that we only know as much about reality as we have experienced. If a phenomenon is outside of our experience then we do not know of its existence. A conception is a way of understanding and describing a phenomenon. Säljö (1988) argues that human thinking is contextually determined (p 42). The environment in which the interview was conducted may influence the response from the individual. He contends that the contextualisation of thinking is handled within phenomenography by:

1. considering conceptions of reality as not residing within individuals since the individual's conceptions of reality may vary with context and time (p 42), and
2. conceiving conceptions as a relational phenomena that are not "inherent qualities in the mind of the thinker or in the objects / phenomena themselves. Conceptions are abstractions from reality" (p 44).

It is not possible to describe a phenomenon without an observer or someone who claims to have some experience, awareness, or understanding of the phenomenon. This is the foundation for the nondualistic ontology of phenomenography (Marton, 2000, pp 104-105; Marton & Booth, 1997, p 179). A way of experiencing is an internal relationship "between the experiencer and the experienced" (Marton & Booth, 1997, p 113).

It is not possible to have a description of something without a describer. The describer's awareness influences the description (p 113). This does not mean that the object does not exist when there is no one there to describe it. It does mean, that without a describer, there can be no awareness of those things, that is, we cannot describe something without being involved as the describer.

Neuman (1997) argues that this nondualistic ontology says that "our world is a real world that is experienced by all our senses but interpreted and understood in different ways by different human beings, depending on our earlier experiences." There is only one world and not one subjective world that an individual holds in their mind and an objective world that is external. The world is both objective and subjective at the same time. The relationship between object and subject forms an experience and encompasses both. How the object is experienced by the subject and how it appears to the subject are synonymous (p 64). The existence of the world is not dependent on the subject but our knowledge is dependent on our experience of the world and we are unaware of any phenomenon that has not been experienced in some way.

Richardson (1999) contends that there is a difficulty here since the nondualist ontology implies that objects and events only exist in as far as they are experienced. He argues that to assume otherwise is to have two kinds of entities in the world; those that are currently being experienced and those that are not currently being experienced. Marton and Booth (1997) acknowledge that the physical existence of objects is not dependent on human beings and our experience of them. However, our knowledge of objects is dependent on our awareness of them. Without experience of an entity, we have no knowledge of its existence and we can not have knowledge of its existence without having experienced the object. Armour (2000b; 2003b) argues that we have total ignorance when we do not know what we do not know and do not know how to find out that we do not know. If something is outside of human experience or awareness then we do not know of its existence and we are not likely to learn of its existence unless we have a process or a means to become aware of its existence.

The phenomenon being studied in phenomenographic research does not need to be dependant on reality or a physical entity. As Säljö (1997) argues, the categories of description are based on the descriptions provided by the participants. They may represent utterances aimed at fulfilling one's communicative obligations (p 177). This may or may not cause problems for the research depending on the objective of the research. What is required is that the participants have some notion of the phenomenon and that the researcher has some awareness of the issues that surround that phenomenon to be able to understand the subjects' descriptions. This raises the issue of 'how' the phenomenon has been experienced and the influence of the 'how' on the 'what' of the

experience. If from the research, we understand ‘how’ the knowledge of the phenomenon was acquired then it may provide some clues with respect to understanding the nature of ‘what’ the phenomenon really is.

The nondualistic ontology means that we can not compare the categories of description against reality but only against an awareness of reality (Uljens, 1996). It may mean that in the research process, the researcher learns to be aware of the phenomenon in a new way because of the stated awareness of a participant or through the analysis of the interviews. A scientist’s knowledge of a scientific phenomenon is based on their experience of, and observations of, that phenomenon. The scientist does not so much re-examine the phenomenon as revisit their experience and observations of the phenomenon to ensure that they did not miss a critical aspect. This is emphasised in Kuhn’s statement that “There is, I think, no theory-independent way to reconstruct phrases like ‘really there’; the notion of a match between the ontology of a theory and its “real” counterpart in nature now seems to me illusive in principle” (1996, p 206).

The focus of this argument is that the real world is only known by the awareness that we have of that real world. There may be entities, as Richardson (1999) argues, that we have no awareness of but because we have no awareness of them we do not know of them and therefore can not argue for their existence.

The researcher is aware that there are different conceptions of object-oriented terminology. When examining the notion of object-oriented program, the phenomenon needs to be recognised as a socially constructed concept. There is no physical entity that is an object-oriented program. There are bits (magnetic recordings in computer storage) but other than the source code, there is not a physical entity that can be examined. The reader of the source code will interpret the source code according to their understanding of what the code is telling them. They may interpret it as an object-oriented program or as some other form of program. They may form an understanding of the program and the interactions that occur within it when executed. The researcher can only attempt to uncover the criteria that the reader would use to interpret the source code and uncover what enables the participant to use the terminology and this approach to programming.

4.1.2 Epistemology of phenomenography

The foundation for an epistemological view for the current study draws on Polanyi's argument that knowledge requires a knower (1958). Polanyi argues that the knower commits to knowledge based on current evidence or awareness. A community may agree to knowledge based on shared awareness or agreed evidence. There is a commitment to the phenomenographic perspective of learning and the relativistic understanding of the nature of knowledge. Polanyi also emphasises the tacit nature of some knowledge in the sense that a person acts on that knowledge but has difficulty expressing it (1966). This underlies the approach to the phenomenographic interview of exploring the subject until the interviewer and interviewee come to a common understanding or until the interviewee has exhausted their ability to express their understanding of the phenomenon.

This is reflected in the scientific community in the way that Kuhn describes the concept of paradigms (1996). Kuhn argues that a paradigm "stands for the entire constellation of beliefs, values, techniques, and so on shared by the members of a given community" (p 175) or that "it denotes one sort of element in that constellation, the concrete puzzle-solutions which, employed as models or examples, can replace explicit rules as a basis for the solution of the remaining puzzles of normal science" (p 175). In Kuhn's argument, a scientific community explores the phenomena of their scientific domain guided by the prevailing paradigm. This restricts the type of questions that can be asked and the experiment that is considered acceptable. In Polanyi's terminology, the paradigm represents the communities commitment to knowledge and the rules that would enable new knowledge to be created.

From phenomenography, awareness forms the foundation of knowing (Marton, 2000; Marton & Booth, 1997). Without awareness, there can be no knowledge of a phenomenon (Svensson, 1997; Uljens, 1996). This reflects Polanyi's perspective of knowledge requiring a knower. Individuals may see and be aware of a phenomenon in different ways and their awareness of a phenomenon may change over time. This is the basis of learning. Svensson argues the knowledge is "created through thinking about external reality" (p 165).

One way of looking at the issue from a phenomenographic perspective is the Indian fable about six blind sages.

> "Six blind sages were shown an elephant and met to discuss their experience. "It's wonderful," said the first, "an elephant is like a rope: and flexible." "No, no, not at all," said the second, "an elephant is like a tree: sturdily planted on the ground." "Marvellous," said the third, "an elephant is like a wall." "Incredible," said the fourth, "an elephant is a tube filled with water." "What a strange piecemeal beast this is," said the fifth. "Strange indeed," said the sixth, "but there must be some underlying harmony. Let us investigate the matter further." Freely adapted from a traditional Indian fable

Each sage has a different awareness of the elephant and brings a different knowledge perspective to the understanding of what an elephant is. This fable illustrates that 'how' a phenomenon is experienced influences the way that it is understood ('what'). Experience of the tail of the elephant gives the impression that an elephant is a rope. The side of the elephant is experienced as a wall. The lack of simultaneous awareness has the consequence that each of the sages expresses their awareness of the elephant in different ways. Their interpretation of the experience is related to other similar experiences. Without experiencing the whole or the parts in relation to the whole, each of the sages has a partial awareness of what an elephant is. Although there is recognition that there must be a harmony, that harmony is not seen without understanding how the whole and the parts fit together and how the elephant relates to its environment. The sages through shared reflection may develop a greater awareness as they develop a simultaneous awareness of how the parts relate and fit together.

4.2 Phenomenography in practice

In this section the issues surrounding the use of phenomenography as an approach to research are discussed. These include the nature of the categories of description (the outcome space), the formation of a space of learning from critical aspects, the impact of researcher conceptions on the research, and issues of the validity of the research.

4.2.1 Categories of description and outcome space

The result of a phenomenographic study is an outcome space. The outcome space is represented by a hierarchy of categories of description that describe different ways of experiencing a phenomenon and the variations that exist between the categories. Marton and Booth say that "the unit of phenomenographic research is *a way of experiencing something*" (1997, p 111). Further, they contend that "the object of the research is the *variation* in ways of experiencing phenomena". This is focused not on how the researcher sees them but on how "others see them". The changes between the different ways of experiencing are considered "to be the most important kind of learning" (p 111).

From a phenomenographic perspective, it is argued that the number of ways of experiencing a phenomenon is limited and these ways of experiencing are logically related (Marton & Booth, 1997, p 112). What they are contending is that there is a limited range of descriptions that are relevant to "ways of experiencing" a phenomenon (p 126) and these descriptions relate to each other in a hierarchy (p 126). The totality of a person's experiences is called awareness (p 123). It is a person's awareness that is described by phenomenography. A collective analysis of pooled data yields a category of description. The focus is on the variations rather than the collective or individual awareness (p 124). It is not a demographic or an individual characteristic. It may be observed in the collective or in the individual.

At any point in time, an individual will experience something in a particular way (p 112). Over time, the way of experiencing that thing will vary. That variation may be explicit or implicit, but it is the variation that brings awareness and learning. For an individual to be aware, Marton and Booth contend that individuals need to experience variation that has two or more possible states. In exploring the outcome space for a phenomenon, a researcher is looking for variations in the way that learners or people experience the phenomenon. The awareness of something will focus on different facets of a phenomenon. Richer variations of awareness will incorporate an awareness of more aspects simultaneously. A category of description is a drawing together of the commonalities across many subjects of the participant's way of experiencing a phenomenon. Any subject may also exhibit more than one category of description

during a single interview session depending on what is focally aware at that point of time.

Bowden (Bowden, 2000) uses an approach of categorising complete transcripts but this seems to be unique to his method of carrying out the research. Where the phenomenon has a diversity of aspects and the participant changes focus during the interview, classification of the whole transcript may prove difficult. The alternative is to categorise based on fragments of the transcripts but the fragments from a transcript have to be seen in context and constituting part of the whole phenomenon (Marton et al., 1993, p 282). Taking a disjoint statement out of context and using it in conjunction with other disjoint statements does not reflect the intent of the subjects. Marton says that some utterances have a clear meaning but that the interpretation of an utterance is in general uncovered through the context (1986). In the current study, portions of the transcript that responded to particular questions were treated as a unit for identifying the primary theme of the subject. However, different portions of a transcript have been related to different categories as the focus of the participant changed during the interview.

Marton and Booth (1997) outline the nature of the categories of description that are the object of phenomenography. They say that the categories of description:

- are *not mental* entities - they do not describe the mental processes that a person goes through to learn something (p 122);
- are *not physical* entities - “they are not merely a description of the phenomenon” (p 122); and
- represent an *internal relationship* between the subject and the world (p 122). They are not dualistic. They represent a unity of the subject and the world that highlights the relationship between two entities.

The categories of description are descriptions of the ways of experiencing a phenomenon. To ignore the phenomenon or the relationship to the phenomenon is to discard their very essence. Marton and Booth (1997) describe the categories of description by saying that the individual categories:

- "should each stand in clear relation to the phenomenon of the investigation so that each category tells us something distinct about a particular way of experiencing that phenomenon" (p 125);
- "have to stand in a logical relationship with one another" (p 125); and
- "denote a series of increasingly complex subsets of the totality of the diverse ways of experiencing various phenomena" (p 126). This is argued from an educational perspective and that some "ways of experiencing it are more complex, more inclusive, or more specific than others" (p 126). In essence, some are more accurate or more enabling than others. Some may provide a base understanding but others will liberate the learner to explore deeper or to utilise the phenomenon more completely.

Earlier in the Marton and Booth (1997) reference, they discussed example cases as a way of building their argument for phenomenographic analysis (Chapter 4, pp 56-81). In this discussion, they emphasised that the difference between the categories is that "certain aspects are held in focus in one case and others in another" (p 64). They argue for the simultaneous and sensuous awareness (p 66). Further they contend that the different ways of experiencing a phenomenon vary with respect to "completeness, their degree of explanatory power, and the extent to which they take aspects of the situation for granted" (p 76). There is a strong emphasis on the changes in figural focus of the aspects as the key ingredient in defining the categories of description.

Cope (2000; 2002a) presents the categories in two forms. He first outlines the categories in terms of their referential (meaning) aspect and a descriptive form of the structural aspect. He then identifies the critical aspects that form the structural aspect in relation to the internal and external horizons (2000, p 115; 2002a, p 74). This is achieved by analysing the categories and related transcript fragments to identify the critical aspects and the variations that apply to each category. These are placed in a table identifying the variance of the critical aspects that aid in distinguishing the categories (2000, p 167; 2002a, p 75). He contends that the different levels of awareness can be formed by comparing the categories based on "the number of aspects of the phenomenon comprising the internal horizon, the nature of an aspect, the number and strength of relationships between the aspects, and the nature of the boundary between the internal and external horizon" (Cope, 2002b, p 69).

Marton et al. (1993) in describing the categories of description in their study include the 'what' and 'how aspects as aspects of the resulting category. The 'what' aspect focuses on what is being learnt (i.e. increasing knowledge) while the 'how' aspect has for its component parts an actor (the learner), an act ("filling the head"), and an object acted upon ("pieces of knowledge") (p 285). The 'how' of learning acts upon a 'what' as the focus of learning. Describing the 'how' means identifying the nature of the 'what'. This style of analysis is carried out for each of the categories. Not all studies have treated the 'how' aspect in this way. Some have treated it as though it was a separate category arrived at from the analysis and then endeavoured to relate it to the categories arrived at by analysing the 'what' aspect.

The categories are most often arranged in a hierarchy such that the higher categories demonstrate a greater level of awareness of the phenomenon (Marton & Booth, 1997, p 125). However, the reasoning used by the researcher in arriving at the hierarchy is not always clear from the categories. Richardson (1999) has criticised phenomenography as a realist pursuit because it would appear that in forming the categories, the researcher is endeavouring to assess the relative merits of the categories based on an objective reality. Lister, Box, Morrison, Teneberg, & Westbrook (2004b) avoided the use of an ordered hierarchy in their study into the teaching of data structures as they did not want to make a value judgement about the relative merit of each of the categories. With Cope's use of the table of aspects and their variations, he has provided a basis for correlating the position in the hierarchy and the number of aspects, the relationship between the aspects, and nature of the variation in the aspects (2000, p 167; 2002a, p 75). Cope is allowing the complexity of the makeup of a category to provide its rank rather than a value judgement based on the researcher's perception of the preferred level of understanding.

There is a strong emphasis in the literature on the categories being related and in a hierarchy. A researcher should not lightly discard the ordering of the hierarchy but neither should they order it without some basis for forming the hierarchy. Cope's (2000, 2002a) approach uses a strategy that has similarities to the SOLO taxonomy (Biggs and Collis 1982) where the number of aspects may be seen as a quantitative measure of increased complexity, and the increasing relationship between aspects may be seen as a qualitative measure. In each category, there should be an increasing number of aspects

held in simultaneous awareness, and/or an increasing awareness of the interrelationships between the aspects.

An alternative method of ranking could share similarities with transformative learning theory (Kegan 2000). Using this approach, the categories would see a change in focus similar to that of the “subject” in our knowing becoming the “object” of our knowing.

Without some theoretical foundation to inform the forming of the hierarchies of the categories, they appear as judgements made by the researcher in forming the categories. If the categories relate to a phenomenon and differentiate critical aspects of the phenomenon then it should be possible to identify differences in the categories based on some theory or logical argument that enable them to form a hierarchy.

4.2.2 Critical aspects and the space of learning

Where the outcome space endeavours to describe the range of ways of experiencing a phenomenon as expressed by participants, the space of learning endeavours to describe what it is possible to learn from a teaching session (Marton et al., 2003). To analyse the space of learning, the teaching materials or a recording of a teaching session is used as the base data and analysed to uncover the variation presented (Marton & Tsui, 2003). The space of learning is not based on an analysis of learner expressions of awareness of the phenomenon, but rather on an analysis of the focus and variations that the teacher has used in presenting the materials.

The researcher examines the teaching resources or the transcript of a teaching session in an attempt to uncover the “pattern of variation inherent in a situation as observed by the researcher" (Marton et al., 2003, p 21). Studies that have endeavoured to uncover the space of learning have tended to focus on the recording of lectures or a classroom session and then transcribing verbatim before analysing for the variations present.

The terminology for theory of variation research draws heavily on that of phenomenography but the desired outcome is different. Instead of uncovering the ways that a phenomenon is experienced by a group of people, the analysis to uncover the space of learning endeavours to describe the phenomenon (“the enacted object of learning”) that is presented through teaching. The objective is to discern the critical features of what is taught (Runesson & Mok, 2003, p 64).

The analysis is based on the principle that in order to learn about a phenomenon, it is necessary to be able to discern the phenomenon from its context and the parts that make up the phenomenon. The features that are discerned with respect to a phenomenon are based on the variations that are made visible to the learner (Runesson, 1999a, 1999b).

As well as examining the way the material is presented, it is also possible to interview both teachers to uncover what they believe needs to be taught ("the intended object of learning") and students to determine what the students actually learnt from the sessions ("the enacted object of learning"). Pattrick (1998) carried out a study with this intention.

In the current study, textbooks are analysed with the objective of uncovering the enacted object of learning. Unlike a classroom transcript, textbooks do not record an interaction between the teacher and the learner. The textbooks are a static resource that supports learning and are a key reference resource for learners. The objective in this analysis was to see how the textbooks addressed the critical aspects uncovered in the categories of description and to endeavour to relate the texts to the categories of description arrived at from the analysis of the interviews.

4.2.3 Dealing with the researchers conceptions

The current study gathered data through interviews of practitioners and from textbooks. The foundation for gathering this data was based on the nondualistic ontology of phenomenography (see section 3.4, page 55). Marton has argued that it is not possible to separate the phenomenon from the participant's observation of that phenomenon (2000, p 105).

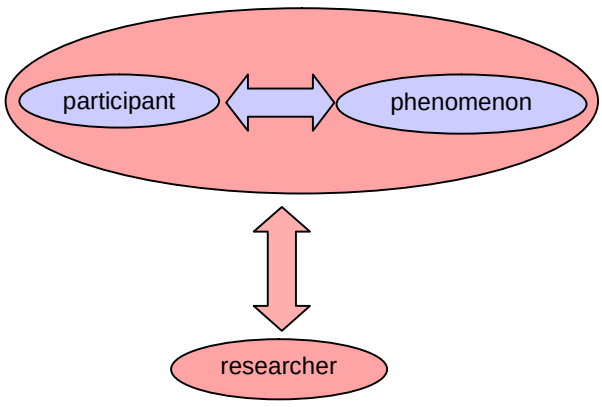

Figure 4.1: Relationship between researcher, participant, and phenomenon

A phenomenographic study sees the researcher examining a phenomenon that is the relationship between the participant and the phenomenon (see Figure 4.1). At the same time, the researcher is involved in a relationship with the phenomenon being experienced by the participant. The researcher has

an awareness of that phenomenon which during the interview, influences the questions asked and the way the researcher responds to answers. Even during the analysis phase, the researcher's awareness of the phenomenon influences what is seen or not seen in the transcripts of the interviews or material being analysed.

The researcher is also in a nondualistic ontological relationship with the relationship between the phenomenon and the participant. In the same sense as it is not possible to separate the participant's observation from the phenomenon, it is not possible to separate the researcher from the phenomenon that the researcher is observing (the relationship between the participant and the phenomenon). This relationship is of the same type as that being observed. The researcher will, like the participant, have focal awareness of what is being observed (the theme) and other themes on the periphery of that theme (focal field). How the researcher changes focal awareness during the interview and analysis process will influence the results of the study.

This is not a problem unique to phenomenography or qualitative methods. In theory driven approaches, the researcher is making decisions as to which theories or paradigms of the research field will influence the hypothesis that they are forming. Those theories and paradigms influence the choice of approach taken to proving or disproving the hypothesis. In effect, the researcher's awareness of the paradigm or phenomenon influences their decisions. Kuhn (1996) contends that the paradigm determines what are legitimate questions for research and what methods are appropriate for that research. In making the decision to work within a paradigm, the researcher is endeavouring to conform their awareness of the phenomenon to that of the selected paradigm.

Scott and Usher (1999) emphasise that even with positivistic research, the researcher influences the outcomes through the definition of the hypothesis and the selection of the data gathering techniques. There can never be a fully objective approach to research as the researcher is always part of the world which they are researching and is always interpreting the data against a particular paradigm. Kuhn (1996) contends that the prevailing paradigm within a discipline determines what is or is not valid research within that discipline. In effect, the researcher commits to a particular paradigm and conducts their research from that perspective.

One way of addressing the issue of the focal awareness of the researcher is to endeavour to ensure that as a researcher, there is a continual attempt to broaden the simultaneous focal awareness of the themes. This might be partially achieved through reading of relevant articles and texts related to the phenomenon and the research method. In the current study, the researcher became aware of object-oriented approaches and languages that he had not been familiar with prior to the beginning of the study. This led to further reading and awareness in later interviews and during the analysis process.

Revisiting some of the readings on phenomenography and the critiques of phenomenography also forced the researcher to re-evaluate his approach to analysis and revisit the data and the way that categories where formed. Interaction with his supervisors and presentation of current study progress also gave opportunity to be challenged on the thinking behind the categories and to re-examine the decisions made and the reasoning for those decisions.

4.2.4 Validity of categories of description and critical aspects

If the researcher's awareness of the phenomenon and what is said in the interviews influences the outcome space, then what gives validity to the research? In a theory driven research project, if another researcher working within the same research paradigm endeavours to answer the same questions using the same or possibly slightly modified techniques, then the second researcher would arrive at the similar results. The experiment is repeatable. Since it is repeatable, the findings are argued to be objective. Applying Marton's statement that the world is both "objective and subjective" means the results of the theory driven research are objective within the theoretical paradigm chosen by the researcher. Another researcher looking at the phenomenon using a different theoretical paradigm may arrive at a completely different proven hypothesis. The two possibly conflicting hypotheses can only be resolved by looking at the theories that enabled the hypotheses to be created. This is awareness of what is valid research within a particular theoretical paradigm.

If the researcher's awareness of the phenomenon is going to influence the way they gather the data and the way they perform analysis, then how can the integrity of the research be ensured?

To resolve this problem, Ashworth and Lewis (1998) contend that the researcher needs to "bracket out" or put aside things that might influence the direction and outcomes of the study. They argue that the 'known facts' should be set aside (p 418). The participant may use words which seem to conform to 'know facts' but actually mean something different. A phenomenographic researcher needs to explore the participant's expressed awareness to ensure that the researcher understands it, and should also endeavour to put aside the assumptions to meanings that they might have of the participant's words. Scott and Usher (1999) say that in order to avoid subjectivism, the researcher brackets their situatedness, subjectivity, and frames of reference. The researcher endeavours to push these aside so the focus is on what the data is saying rather than the researcher's interpretation of the data. Here the researcher is being asked to pull apart the "one world" of the researcher's experience of the participant's conception that is "simultaneously objective and subjective" for the researcher (Marton, 2000).

In terms of phenomenography, Booth contends that there are "three aspects of validity" (1992, p 65). These are content-related, methodological, and communicative validity.

Content-related validity is that the research must "be grounded on a sound understanding of the subject content" (p 65). The researcher needs to understand the subject and the conceptions that might exist. This would seem to question whether bracketing out is desirable since the researcher needs to have some understanding of the terminology within the subject area. At the same time, the researcher needs "to be open for ways of understanding it which differ from those generally accepted" (p 66). The researcher needs to be open to learning and exploring terminology which is not recognised or alternative understandings of common terminology.

For the current study, the researcher had an interest in the ways that learners and practitioners express their awareness of object-oriented programming. The researcher's familiarity with the debates within the field of study made him aware of the different approaches used in object-oriented software development and their possible influences on the awareness of the nature of an object-oriented program.

Methodological validity relates to the appropriateness of the methodology and the population for the questions being studied (Booth, 1992, p 66). Booth contends that this validity includes:

- the wise choice of a theoretical sample from the appropriate or suitable population;
- the context of the interviews so that open and deep discourse occurs between the interviewer and the interviewee;
- the structure and content of the interview; and
- the ability of the interviewer to "shift focus between the interview as it is progressing and the interview as it is forming a piece of data, while at the same time being aware that this shift is taking place" (p 66).

Booth interprets Glaser and Strauss' definition of theoretical sample as "a choice made to represent the population in some theoretically appropriate way" (p 58). Glaser and Strauss (1967) describe a theoretical sample as being judged as adequate if there are no obvious exceptions to the theory when assessed by a critical reader. Seale (2003) argues that an "adequate theoretical sample is judged on the basis of how widely and diversely the analyst chose his groups for saturating categories according to the type of theory he wished to develop" (p 230). The issues of the sample for the current study and issues related to the structure and content of the interviews are discussed later in this chapter.

Booth describes communicative validity as being understandable to those in the field and being able to be communicated to the research community (1992, p 65). She describes the internal communicative validity as being understood and recognised by the participants in the study and external communicative validity as being understood by other researchers with similar interests (p 67).

Bowden (2005) talks about validity of the research in terms of the influence of the researcher. His solution revolves around ensuring that there are a few consistent core questions from which the interview is started and that from there on the only themes explored are those introduced by the interviewee. The interviewer's role is simply to draw out the understanding of the interviewee. This approach still relies on the interviewer recognising the key themes that need further explanation.

Patrick (2000) in some respect supports Bowden's perspective when she talks of "the importance of remaining open to the implications of the data" (p 133). She contends that in doing phenomenographic research, it is necessary to become "conscious of our expectations and actively challenging them." An alternative approach that she suggests

is to “aim to adopt a more collaborative approach and to develop a shared understanding through the research process” (p 134). The core theme here is endeavouring to see the phenomenon as the observer or participant sees the phenomenon and for the researcher not to place their own expectations on what the observer is saying.

There is another alternative which is the idea of looking again (Brew, 2001). This involves the researcher having taken a first pass at the data and drawn conclusions, then returning to the data and exploring it again from a different perspective. With a range of interviews, there are a range of experiences expressed. Some are very similar but each has its unique flavour. Having completed a first analysis, a return to the transcripts taking a different perspective can reveal new insights and possibly alternative understandings. Each understanding of the interviews needs to be reviewed and considered in terms of whether it is consistent with the interviewee’s expressed intentions and views.

Is it possible to totally remove the influence of the researcher from the study? The researcher may endeavour to be objective and to put aside his own frame of reference but inevitably, he may miss things that do not fall within the envelope of his awareness or he will struggle to understand things that are peripheral to his current frame of reference.

Marton and Booth describe research as a learning experience (1997, p 129). The researcher should be learning from the experience and his views and understanding of the phenomenon should be changing. It is also likely in the process of a phenomenographic study that the research will have influenced the understanding of the participants as the researcher endeavours to draw out their meta-awareness of the phenomenon.

Sandberg argues that interjudge reliability overlooks “the researcher's intentional relation to the individuals' conceptions of reality” (Sandberg, 1997, p 208). He contends that there is an intentional relationship to “the individuals’ conceptions being investigated.” He then proposes a strategy for dealing with this intentionality which McKenzie summarises as

> "Maintaining interpretative awareness involves five steps (Sandberg, 1997, p 210):
>
> - Remaining "oriented to the phenomenon as and how it appears throughout the research process";
> - Describing experience, rather than explaining it;
> - Treating all aspects of individual's descriptions as equally important;
> - Searching for the meaning structure of the experience, using "free imaginative variation";
> - "Using intentionality as a correlational rule" by focussing on the what and how of the individuals' experience and the internal relations between the what and how.
>
> Rather than focusing on external relations between judges or measures, interpretive awareness therefore focuses on reliability in terms of internal relations between aspects of the phenomenon as it applies to the researcher" (McKenzie, 2003, p 94).

Cope supports the argument by saying that "A researcher is required to be aware of their interpretations during the research process and demonstrate how the interpretation processes have been controlled and checked" (Cope, 2002c, p 87). However, he also argues that interpretative awareness is one aspect of the validity and that there is a need to have some other form of reliability check.

Another alternative is that the researcher expresses their own awareness of the phenomenon so that its influence may be seen and taken into account by the reader of the research results. This should include ways in which the researcher's awareness was challenged and changed during the research process. This would enable the reader of the results to determine whether the researcher was genuinely open to alternative understandings that challenged their own perspective and world view.

Burns (1994) says that the researcher should acknowledge their background. Cope agrees that "the researcher's prior experiences are part of the process" and that the reader is provided with the context for the analysis when the researcher openly acknowledges their scholarly knowledge of a phenomenon (Cope, 2002c).

In hypothesis / theory driven research, the researcher is attempting to prove or disprove the hypothesis. This can blinker the researcher to other data being provided by the research. Kuhn (1996) emphasises this idea as he talks about the way that the current paradigm determines what is valid research. In a data driven approach to research, there is more flexibility to adjust the direction of the analysis based on what the data is providing. In this context, it is expected that the researcher's bias could, and possibly should, change as they explore the different concepts being exposed through the analysis of the data. In the current study, part of the validity of the research relates to the way that the researcher's theories of the outcomes changed over the period of the data gathering and analysis of the data. This openness to being challenged with respect to one's own thinking and understanding is a key element in completing this type of research.

Scott and Usher (1999) also contend that the results expressed in the researcher's report are still the researcher's interpretation and understanding of what the subject of the research is endeavouring to say. In effect, the research expresses the voice of the researcher in interpreting what has been discovered. Phenomenography endeavours to address the possibility of a muted voice by endeavouring to identify different categories of description, but these will still be influenced by the researcher's understanding of what has been said and the framework that has been used for interpreting the subjects' descriptions.

4.3 Research Design

This section describes the design of this particular study into the phenomenon of object-oriented programming. The study primarily focuses on what an object-oriented program is. Initially the overall structure of the study is discussed and the frame of reference for the study. The two phases of the research are then discussed in detail. This is then followed by a discussion on the limitations of the current study.

4.3.1 Overall shape of the study

Where the current study differs from many previous studies using phenomenography is that it seeks to understand the way that practitioners are aware of the phenomenon, in this case, the practitioners' awareness of object-oriented programming, in order to better

understand what variations could be used in teaching. Most phenomenographic studies have explored the learners or teachers perceptions of the phenomenon.

The first phase concentrated on uncovering the perceptions of practitioners with respect to the nature of an object-oriented program, and the second focussed on the approaches to teaching and communicating concepts used in textbooks and whether these techniques were or were not opening up the space of learning with respect to the categories of learning. The second phase was guided by the aspects identified as critical to understanding object-oriented programming as identified in the first phase.

The researcher has developed a habit of using a personal reflection journal as a way of recording critical events related to his teaching, for reflecting on readings, and on each lecture, tutorial, or laboratory session. Although not seen as a primary source of data for the research, the researcher's journal does provide a trace of the developing thinking of the researcher in relation to the research, his reading, and his teaching. Reference to entries in the researcher's journal will be made where there is a need to identify the change in thinking that has occurred as a result of the research.

4.3.2 Influential factors

The researcher entered into the current study having had what was perceived as success in teaching procedural / imperative programming using a strategy based on building a conceptual understanding on the nature of a program (Thompson, 1992). This experience provided a focus for uncovering the awareness of an object-oriented program that was held by practitioners and the possible transitions or variations that existed between the awareness of a novice programmer and those of the expert. This bias is shown in the nature of the questions planned for the research.

Another influence was the researcher's experience in endeavouring to foster object-oriented thinking in third year students whose background experience in programming was primarily procedural / imperative in nature. The object-oriented approach to programming involves becoming aware of, and being able to apply, specific technical terminology and approaches to programming. This led the researcher to seek to understand how this terminology was referred to and how it influenced practitioner awareness of object-oriented programming.

An influence that consistently ran in the background was the ongoing debate between teachers of novice programmers as to where the teaching of programming should start. Should novices be introduced to procedural / imperative programming first and then have object-oriented added on, or should the teaching start with an objects first approach in one of its forms? It seemed to the researcher that the foundation for this debate was confused and it was unclear what the significant conceptions were that made a person a competent object-oriented programmer.

These factors influenced both the approach to the design of the research and the direction of the analysis.

4.3.3 Phase A: Practitioner ways of experiencing object-oriented programming

This section describes and justifies the use of phenomenography to answer research questions 1a and 1b. The research design requires exploration of practitioner's ways of experiencing object-oriented programming through interviews. The selection of a sample of practitioners, construction of the interview instrument, administration of the interview, and data analysis are discussed. Consideration was also given to the issues of conducting interviews with graduate students and university staff members.

1) Sampling

The population for the current study included any practitioner who claimed have some level of awareness of object-oriented programming and preferably was writing programs using this technique. This group included reasonably new practitioners through to those with considerable experience.

Thirty one practitioners were interviewed. The size of the sample was based on sample sizes used in phenomenographic studies reported in the literature. These interviewees included:

1) eight industry practitioners with a range of years of experience including some practitioners who have been considered leaders in the use of the technique,
2) fourteen academic practitioners who have been teaching object-oriented concepts and techniques, and

3) nine postgraduate students who had studied or used object-oriented techniques.

The choice of the subgroups endeavoured to ensure that a range of different expressions of awareness of the phenomenon would be obtained and that these expressions of awareness included those from relatively novice object-oriented programmers to those of very experienced object-oriented programmers. The sub-groups were chosen because of the perceived difference in focus of the practitioners in each. The goal was to obtain at least seven participants for each of the subgroups.

The interviewee selection was based both on a convenience sample and a purposive sample. A convenience sample involves choosing the nearest individuals as respondents, while a purposive sample is based on the researcher handpicking the cases to be included in the study (Cohen & Manion, 1994, p 88-89). The researcher initially selected participants from known contacts and their recommendations. As the data set grew, the selection became much more based on ensuring that a diversity of views were included in the study. This included approaching specific people who clearly voiced particular views in public forums such as online mailing lists and discussion forums.

In all cases, the type of object-oriented systems participants were developing and the design principles that they were applying were determined as part of the initial interview process. In some cases, this discussion was held during the process of arranging an interview. It was desirable not to include people in the sample who were using an object-oriented language to perform procedural programming tasks or people who were aware of the terminology but who had no practical experience or theoretical knowledge of the use of object-oriented programming. The expert practitioners interviewed could be described as the believers in object-oriented approaches to software development in the sense that they were active in promoting this approach to development or believed in an objects-first approach to teaching. Two participants had previously been strong advocates for object-oriented approaches but were now utilising other strategies. It was decided to exclude those who indicated a limited use of object-oriented approaches and argued that the approach had failed to live up to expectations. This was done partially to avoid drawn out debates. However, it is recognised that interviewing such people may have added another dimension to the study.

The post-graduate students selected were drawn from New Zealand tertiary institutions. Emphasis was placed on those students who have completed an object-oriented programming paper, or who were using, or had used, object-oriented techniques in their research.

The academic practitioners selected were drawn primarily drawn from New Zealand tertiary institutions, although some were attached to overseas institutions. Those selected were, or had been, involved in teaching object-oriented programming or software development strategies, or had written object-oriented programming textbooks.

The industry practitioners were those with whom the researcher already had some contact. The bulk of this group were approached directly by the researcher either because they were known to the researcher or the researcher was aware that they held specific views that were not coming through in the other interviews.

2) Interview Schedule

The primary data gathering technique used in phenomenography is the interview although other variations are acknowledged as being appropriate and valid (Marton & Booth, 1997, p 132). For any data gathering in phenomenography, the interviewee has to be reflecting over their experience in a state of "meta-awareness" (p 129), that is, they have to be aware of their awareness of the phenomenon. The interviewer has to work to foster "articulation of the interviewee's reflections on experience that is as complete as possible" (p 130). The interviewer needs to become aware of the phenomenon in a way that is consistent with the interviewee's awareness of the phenomenon.

The types of interviews used for phenomenographic studies are semi-structured. The researcher needs to have "a small number of predetermined questions which deliberately approach the phenomenon from a variety of directions" (Booth, 1992, p 59). The desire is to obtain a full understanding of the interviewee's awareness.

An interview schedule was prepared for the current study (see Appendix 3) and piloted with three participants. The results of the trial suggested that it could be utilised without change.

The focus of the interviews was on understanding the participant's 'way of experiencing' the phenomenon using starting questions such as:

1. How do you assess whether a program has been written using object-oriented techniques and practices?
2. How do you think a person learns object-oriented techniques and practices?
3. How would you explain to a person learning about object-oriented practices, what a 'program' is?

The first question was consistently asked as stated above through all the interviews. The other two were asked but adjusted in form to match the direction in which the interview had taken. The interviewer used the interview schedule as a reference guide.

The interview schedule included some sub-questions designed to ensure that the interviewer followed up on key terminology used. These were of the form "how would you describe X?" This style of question was primarily used when the participant focused on the terminology and gave little explanation of what the terminology meant. Although this question appears to be generic, in preparing the schedule, the researcher had in mind that participants were likely to focus on object-oriented terminology and concepts. This question was a reminder to the interviewer to explore the participants' understanding of the terminology and concepts during the interview. It was envisaged that a second level of analysis might be possible that explored the variation in the understanding of the core terminology and concepts.

The objective of the first question was to encourage participants to identify the features that they would expect to be reflected in an object-oriented program. In preparing the schedule, three possible scenarios were identified and sample programs created. These sample programs were used in the pilot interviews and some of the early interviews but were abandoned because they were not serving the intended purpose. Some participants who were unfamiliar with the programming language used struggled to read the code or were distracted by details or syntactical items. In some cases, this led to the researcher explaining what the code was doing. As a consequence the interview became distracted from its primary intent. For novice programmers, the programs drew the discussion back to the use of language features, and for experts, the programs led to comments on 'appropriateness' of the solution and away from the understanding of object-oriented

programming. The interview schedule was not adjusted but the interviewer utilised the questions related to the code samples as prompts for fostering discussion with the participants.

The interview schedule shows that in the opening section based on this first question, the focus was primarily on the generic concepts of modularisation, and architectural issues. Participants responded with a much wider range than anticipated by the schedule.

The structure of the questions was influenced by the researcher's experience in teaching imperative programming where the concept of a program as data plus algorithm had been used to lay a conceptual framework (Thompson, 1992; Wirth, 1976), and the researcher's awareness of issues in the objects first teaching debate. These objects first teaching issues included awareness of Stein's view of an object-oriented program as being composed of interacting entities (Stein, 1998, 1999, 2003) and the view that in object-oriented programming, algorithms were represented in the interactions between objects (Bergin, Stehlik, Roberts, & Pattis, 2005; Berglund & Lister, 2007). Another factor included the influence of agile development techniques (Beck, 2000; Cockburn, 2002, 2005; Highsmith, 2002; Jeffries, 2004; Jeffries et al., 2000) where the emphasis on refactoring and the elimination of 'code smells' (sic) (Fowler, 1999), although not expressing an emphasis on interactions or algorithms in the code, shows a different understanding of code structuring to those represented by the imperative paradigm and the 'program equals data plus algorithm' style of thinking.

In preparing the interview schedule, the focus was on uncovering aspects related to the internal horizon. This focus can be seen in the section labelled "Object-oriented concepts". The opening question of that section ("What are the concepts that you see as being important when performing object-oriented development?") was designed to elicit the names of particular concepts. It was envisaged that the participants would be asked to describe these and to rate their importance. The concepts of object and class were singled out for particular attention as they were seen as being core concepts by the researcher.

This focus is also reflected in a journal entry written by the researcher, where he says:

> "In a programming context, we can talk of the boundary between a program and the rest of the system. That is if we perceive there is a boundary. Maybe I will see this form of awareness in some of my interviews. This external horizon is more concerned with distinguishing the program from its context.
>
> "The perspective that I was really interested in is the internal horizon. How do my participants understand the way a program is structured? What is their understanding of its structure and parts? As I reflect, I wonder whether I made this clear or whether it was necessary to make this clear." (20 November 2006)

This was written fairly late in the interviewing process but reflects the focus given to the interviews both in preparing the interview schedule and in conducting the interviews.

During the interview, it was important to recognise concepts stated but not described and to encourage description of these concepts and the reasoning for their significance. The interview needed to be focussed while still leaving room for the interviewee to freely express their views on the key themes. It was important that the interviewee's awareness of the core concepts was obtained rather than directing to specific concepts and descriptions. The interviewee may refer to standard definitions from texts and these needed to be further explored. In such cases, the interviewee was asked to explain further.

The second question was primarily aimed at uncovering the practitioner's perceptions of learning object-oriented programming. This focus has not been analysed as part of the current study. In preparing the schedule, it was envisaged that participants would focus on specific object-oriented concepts, techniques, or practices. Those named by the participants would then be explored in greater detail. Although the focus in this section was on the 'how' of learning, it proved to be helpful to uncover the participants' understanding of the nature of an object-oriented program.

The final section that contained the third question was aimed at uncovering the participants' understanding with respect to object-oriented programming of generic concepts used in relation to software development. Focus in this section was on how the participants described what a 'program' is.

3) Administration

All practitioners were approached by the researcher and provided with a copy of the information sheet about the research (see Appendix 4). Before approaching academics and graduate students, written approval was sought from the head of department or school for the researcher to approach the academics and graduate students for the study (see Appendix 4). The academic and graduate student participants were then approached by the researcher and provided with an information sheet. All participants signed a consent form.

A suitable time and neutral venue was then agreed with the participants. Where possible, this was a venue and time proposed by the participant. Most interviews were conducted in environments that were familiar to the participants and without time constraints. Most interviews took an hour to conduct. Interview lengths ranged from thirty minutes to ninety minutes. In the few instances where time was a factor, the researcher was conscious of the restraint that this was placing on the depth of the discourse.

All interviews were conducted during 2006. Although all participants were offered the opportunity to obtain a copy of their transcripts, none requested that they be provided with it.

4) Data analysis

The interviews were transcribed verbatim by the researcher. This provided an opportunity for an overview and initial identification of themes within the interviews. The initial analysis involved reading the transcripts and identifying key themes related to the primary questions. The data from the interviews were analysed using the techniques of phenomenography to identify variations in awareness of the key phenomenon, an object-oriented program, and to identify the “critical aspects” used in describing the phenomenon (Marton, 1981; Marton & Booth, 1997).

For the current study, the focus was on relevant utterances taken in the context of a particular response sequence. This was done because the interviews tended to move in terms of the focal theme. To retain relevance, selection of material was preferred rather than attempting to categorise the interview as a whole. Frequently, the researcher would

revisit analysed interviews based on a perspective picked up from a later interview or from a reading or mailing list conversation. The themes were recognisable primarily by the meaning that they carried with respect to object-oriented programming. This helped identify the categories of description that might be relevant.

A key issue here is viewing the data from different perspectives. What the researcher is endeavouring to do is to uncover variations of experience of the phenomenon. These variations must be observable variations or they will not be discerned (Marton & Booth, 1997, p 134). Each reading of the transcripts (like reading a book) should "keep changing in appearance" (p 134), that is, each reading will bring a new variation and perspective about ways of experiencing. The key issue is to uncover why they saw things or experienced things the way they did. The answer to the experience is secondary to the how and why they arrived at that solution or experience.

The qualitative analysis tool (Atlas.ti) was used to keep track of identified utterances and associated keywords. In reviewing the passages of the interviews, it was identified that certain key phrases or terminology were occurring consistently for different categories. The analysis tool was used to help identify these key phrases and to code passages that contained them.

The initial analysis focused on the three primary questions used in the interview. Since the interviewees often referred to aspects covered by the other questions when answering a question, the complete interview was read for each question and relevant utterances identified and tagged with codes using the analysis tool. This had the effect of pooling the results.

Some emergent categories were clearly identifiable by key terminology. The key phrases and terminology helped identify possible "critical aspects" that were related to each of the categories of description. In most cases, the key phrases and terminology had been picked up during the interview and explored through the sub-questions that were planned as part of the interview schedule. An example is 'interacting entities' where participants would clearly talk about interacting objects or sending messages. Where the term 'sending messages' was used, the surrounding content was read to ensure that the participant was indicating this as the key implementation strategy.

Based on the identified key terminologies, the automatic coding facility (auto coding) within Atlas.ti was used to identify and code additional content that related to the identified keywords. Auto coding needed to be reviewed as the use of a word did not necessarily mean a particular type of awareness. However, it was invaluable in identifying where themes were discussed and in extracting the appropriate portions for more detailed analysis and comparison. The auto coding initially focused on assigning a unique code for these key phrases and terminology. The passages were then reviewed to determine whether they consistently supported a category of description and, if so, the appropriate category code was assigned.

Having coded the interview content, it was possible to build a network view of the codes using the analysis tool and thus establish possible "critical aspects" within the data.

Consideration was given to performing some form of statistical analysis based on the relationship between the codes for passages and codes for critical aspects. However, further analysis revealed that this would have little meaning as it was the underlying meaning that was needed rather than relationships between key words or codes.

5) Ethical Considerations

Although most of the people approached for the interviews were external to the university, some of the graduate students were already known to the researcher. In order to avoid harm to the participants, written informed consent was sought from each participant confirming that anonymity of participants would be maintained in all reporting (see Appendix 4). To avoid possible repercussions for the institutions, written consent was obtained from the heads of departments in institutions where graduate students or lecturers were approached. The researcher was not involved in teaching any postgraduate papers or in supervising graduate students in his own university.

4.3.4 Phase B: Analysis of textbooks

The analysis of textbooks was based on the results of the analysis of the practitioner interviews. Twenty textbooks based on object-oriented programming were used for the analysis (see appendix 5 for a list of the textbooks used). This section describes how the

textbooks were selected, the principles used to guide the analysis, and the approach to analysis used.

1) Sampling

The textbooks were selected based on their claims to be teaching object-oriented programming and the ability of the researcher to obtain copies. In order to restrict the range of books from which to select, the focus was placed on textbooks that taught Java programming or that claimed to offer a unique approach to the teaching of object-oriented programming. As the interview participants included a practitioner with a strong agile software development bias, a book targeting the teaching of agile development practices was drawn into the set of textbooks. This text was not aimed at novice programmers although it was aimed at programmers who were endeavouring to learn a new language.

The initial textbook selection was a convenience sample based on textbooks already obtained by the researcher as possible texts for teaching object-oriented programming in Java. Texts were not considered in the sample if they introduced imperative programming and included a chapter on object-oriented programming simply to round out the text or to ensure that the use of classes and objects were included in the text. The desire was to analyse textbooks that were seen as primary texts for introducing students to object-oriented programming.

2) Instrument construction

In planning the analysis of the textbooks, it was planned to construct a set of lower level questions based on the “critical aspects” arrived at from the interview analysis. These questions became:

1. How do the texts address the nature of an object-oriented program?
    a. How do the texts address flow of control?
    b. How do the texts address the usage of objects?
    c. How do the texts address the nature of the problem solution?
2. How do the texts address the design characteristics of an object-oriented program?
    a. How do the texts address the influence of technology?

b. How do the texts address the program design principles?
c. How do the texts address the cognitive processes?
d. How do the texts address the issue of modelling?

3) Data analysis

Since the textbook analysis followed on from the interview analysis, the textbooks were examined to see whether they supported the outcome spaces obtained from the interview analysis. The initial pass through the textbooks involved identification of key definitions and those portions of the texts where the "critical aspects" were introduced. The way that these aspects were introduced and the examples used was examined to determine the emphasis given by the authors. The emphasis, or lack thereof, on the "critical aspects" helped define the space of learning for each of the textbooks.

Since all except one of the textbooks were in paper form only, the analysis method relied on selecting appropriate portions and reading and rereading. Indexes provided some assistance but if the textbook writer did not provide a section focusing on the research themes, it became more difficult to determine the emphasis on critical aspects.

As well as looking at any declared understandings of what an object-oriented program is, examples and related descriptions were examined to see whether the author consistently supported the declared understanding. Making a simple statement on the nature of an object-oriented program does not mean that this is the message that will be taken away by readers. Variation theory contends that it is the variations of critical aspects that define the space of learning. If the examples in the text do not present variations that open up the space of learning to reinforce the declared understanding, then the learner may recall the definition but take away that programs are not written that way.

One of the difficulties of applying variation theory analysis to the selected textbooks was that often authors introduced a concept with a definition followed by a single example. There was no alternative solution or alternative problem that demonstrated the same concept or contrasted the solution with alternative solutions. This made it difficult to determine what the author regarded as critical with respect to that concept. With respect to the major question of what is an object-oriented program, this difficulty was

compounded because program examples were given to illustrate the use of a construct and not necessarily to illustrate the nature of an object-oriented program. This meant that the analysis involved looking for where concepts were discussed and endeavouring to make the connections throughout the text.

This difficulty may not be so obvious when looking at how particular programming constructs or principles are introduced. Programming constructs or principles are often the focus of the authors and alternative examples are provided by some authors. Guzdial and Ericson (2005) do this when introducing loops. They initially introduce the use of a "while" loop (pp 93-110) using a number of alternative examples. They then introduce an alternative loop form, the 'for' loop (pp 117-124) performing a number of slightly different tasks to the same type of media. A template is then given in the summary that compares the two different loop methods. With these types of examples focused on particular concepts, applying variation theory is easier than when dealing with the more generic concept (a program) that is the focus of the whole text.

In order to make the textbook data easier to work with, an initial pass of the textbooks was conducted which focussed on extracting the definition statements for the "critical aspects". This was predominately the way that the authors introduced the concept of programming and the way that they introduced object-oriented concepts or terminology.

4.4 Limitations of the study

There is always the issue of the researcher's background knowledge of the domain and its influence on the study. During the analysis, it became evident that some of the aspects were not followed up during the interview process. This has an impact on the ability to determine the variations in understanding of critical aspects particularly within the design of program categories. The original design had been based on the critical aspects being the paradigm constructs with possibly some discussion of generic concepts. It was not envisaged that any of the participants would, in discussing the characteristics of an object-oriented program, discuss design qualities or emphasise process characteristics. This was despite the researcher being aware of the process emphasis expressed by the participants. In some cases because of the researcher's interest, some of the aspects surrounding these themes were explored in more detail, but

not for some of the more general aspects like the understanding of reuse or maintenance issues.

In some of the early interviews, primarily with people expressing a ‘sequence of instruction’ view of an object-oriented program, the interviewer questioned the interviewees in an argumentative manner. In one case, this was almost to the point of trying to teach the interviewee about other ways of viewing the nature of an object-oriented program. In the two cases where this is clearly visible in the transcripts, the interviewees showed how strongly they held to their understanding. This actually strengthened the case for this category. However, as the interviewer endeavoured not to enter into debates with such people, there was less exploration of the implications of that understanding or of some of the critical aspects with participants who sought the comfort of book definitions.

Another issue was that some of the interviews became more discussion in nature particularly when the interviewee shared a similar perspective or understanding as that of the interviewee. The easiest interviews to conduct were those with people who held understandings of object-oriented programming that covered aspects in the higher categories. These people appeared to have developed a clearer ability to express their understanding and also had the desire to communicate this to others. At times, the interviewer wondered whether the interview would provide the appropriate data but in the end these interviews proved to be more valuable as the interviewees expressed their thoughts and ideas more fully.

It is also clear from the transcripts that the later interviews were of better quality than the earlier interviews. For this type of study where the interviewer is not familiar with the interviewing techniques, more time needs to be spent early on practicing how to conduct these interviews. Focus should be placed on the interviewer being able to identify, during the interview, possible “critical aspects” raised by the interviewee, and then to seek the interviewee’s understanding of those ‘critical aspects’. This lack of interviewing experience accounted for a number of the difficulties that occurred during the current study.

4.5 Conclusion

The primary approach to gathering data for the current study was the interviewing of practitioners with differing levels of experience of object-oriented programming. The sample was obtained through both convenience and purposive sampling. Three subgroups of participants were targeted with the objective of ensuring a spread of expressions of understanding.

The analysis of sample textbooks was planned based on a convenience sample. The textbooks were analysed based on the critical aspects for the outcome spaces derived from the analysis of the practitioner interviews.

Chapter 5. Practitioner ways of experiencing object-oriented programming

The first question for this research is:

3) How does the awareness of an object-oriented program vary among the 'practitioners'?
    a. What are practitioners' ways of experiencing object-oriented programming?
    b. What are the "critical aspects" that distinguish the different variations in awareness expressed by 'practitioners'?

This question is about what practitioners understand as being important for object-oriented programming and how they are aware of the concepts that they describe as important. In preparing the interview schedule, it was considered that the starting point for uncovering critical aspects should be what the interviewees understood an object-oriented program to be. This is a focus on what they believed they were creating. As well as expressing the 'what' aspect, the practitioners also expressed a 'how' aspect; that is, they talked about issues related to how they would perform object-oriented programming.

In this chapter, the two outcome spaces obtained from the interview transcripts are presented. The "critical aspects" which distinguish each of the outcome spaces and categories are also presented. These are supported by quotes from the interview data. Each quote is identified by a unique code that relates to the particular interviewee.

5.1 What is an object-oriented program?

In all thirty one interviews, the practitioners were initially asked "How do you assess whether a program has been written using object-oriented techniques and practices?" Further questions were then asked based on the interviewee's responses. Closer to the end of the interview, interviewees were also asked a question similar to "How would you describe what a 'program' is in an object-oriented context?" This varied depending on the direction that the interview had taken and whether the interviewee asked for clarification.

During the interview process, it was recognised that two different types of responses were being given in the interviews. The response to the first question tended to focus on design issues of an object-oriented program, and for the second question on the nature or structure of an object-oriented program. At one point in the analysis, these were seen as forming a two dimensional matrix, but further analysis showed that the design characteristics outcome space was not referenced consistently with respect to the categories of the nature of a program outcome space.

The nature of a program outcome space categories are easier to describe than the design characteristics outcome space categories. This is in part because the design characteristics categories involve the identification of aspects that need further investigation and analysis. Another distinction between the two outcomes spaces was that the terminology used to describe the nature of a program clearly used keywords related to the critical aspects while the terminology used to describe design characteristics related to the underlying concepts rather than to the critical aspects.

5.1.1 The nature of an object-oriented program

Interviewees recognised the nature of a computer system and the constraints that this placed on the nature of a program. Some were able to see beyond the technical constraints and visualise an object-oriented program as having a technical structure that was not constrained by the nature of a computer or the operating system or environment in which the program would be executed. This led to five distinct categories (see Table 5.1). The analysis of the interview data uncovered the following technical or structural characteristics of a program. The categories are defined in order of the simultaneous awareness of critical aspects required. These aspects will be discussed later in this section (see page 140).

Category	Referential	Structural
N1	Sequence of instructions	Recognises that all programs have a flow of control that defines the order in which program instructions are executed. The order of execution is seen primarily as sequential in nature
N2	Written using an object-oriented framework	Sees a program as object-oriented if it uses an object-oriented framework or using an object-oriented environment
N3	Based on data types	Sees an object-oriented program as being based on defining new data types or on data analysis or the development of a data-driven model
N4	Based on interacting entities	Sees an object-oriented program as a set of behavioural entities that interact to achieve the required objective
N5	Artificial construct	Sees a program as an artificial construct imposed by operating systems and not applicable to the discussion of object-oriented entities

Table 5.1: Categories of the nature of an object-oriented program

N1) Sequence of instructions

The focus in this category is on what the computer has to do with the program once it is created (flow of control) and on the tools used in its creation (objects from an object-oriented framework). This category shows an understanding of what the computer will do with the program and therefore of attributes of the internal structure of the program. As one participant implied, all programs become executable binaries but that says nothing about how the program is executed or the way it might be constructed. The primary aspect which distinguishes this category is that it recognises that a computer executes a program in a sequential fashion. The supporting aspect is that objects are used from the supporting framework.

Put in the simplest of terms,

> “A program is some instructions to a computer to tell it what you want it to do” (I16).

In this view, the sequential nature of a program is linked with the way a program is loaded and executed within a computer system. This operating system requirement influences the understanding of what a program is. This is expressed as being independent of the programming paradigm.

> “It’s a set of instructions that when it is compiled or executed it does something” and “A program has to start somewhere. Object-orientated or procedural, [...] you’ve got to hit an entry or a main method or run method or an execute method that does something” (I27).

Here, a program may involve sequential execution but it may involve multiple sequential threads of execution. These sequential threads are seen as running across the class structure of an object-oriented program. There is almost a disjoint relationship between the nature of execution and the structure imposed by the use of classes.

> “Its basically one or more threads of execution operating on a set of class definitions. You have what is essentially a sequential CPU stepping through a set of sequential instructions that have been set out in your class definitions and in so doing it’s creating object instances on the heap and local variable instances on the stack and putting the pointers to all these various data structures and execution instructions onto the stacks” (I18).

Participants who expressed this view suggest the sequential nature is not limited to how the program is executed. In designing a program, this sequential execution needs to be considered. How this is expressed in program code is dependent on the choice of programming language but does not remove the sequential nature of the program.

> “Getting across the idea that there are sequence or set of things you are wanting to happen and the way that you do that is write instructions and you are confined in how you write those instructions by the language you are going to use” (I12).

It is possible that this dependency on sequential flow is being de-emphasised with the development of model driven approaches to software development. A participant with a background of having taught object-oriented analysis and design (OOAD) using the Rational Unified Process (RUP) argued that the sequential nature of execution influenced the analysis and design of object-oriented programs. This participant also showed that even where object-oriented analysis and design (OOAD) was being performed, there was an expectation of a need to understand the "sequence of events".

> "Whereas a lot of the original OOAD stuff, you were using an imperative approach […], essentially in manifesting within the requirements, you were making statements about the sequence of events and the order of execution of certain things and how they should occur in terms of steps and alternate flows. And this emphasis on flows is really now sort of moved into the background again" (I26).

The characteristic of this description which distinguishes it from the first is the idea of how a program is executed or structured during creation. The emphasis is upon a program primarily being sequential in nature even though objects are being used. Although quite a number of participants discussed classes as a way of modularising the code, this was not an overriding theme in any of the interviews. The strongest statement to this effect was:

> "the main purpose of object orientation is, in my opinion, very similar to modularisation. And therefore, […] I think the main purpose is to modularise the code so that most of them can be reused without redundancy" (I29).

Being sequential in nature did not lead automatically to the identification of classes as a technique for modularisation. The concept of modularisation is a theme followed through in the design characteristics outcome space. Predominately, this view was expressed from the perspective of the sequential nature of any computer program. As such, it was seen as a characteristic that also applied and needed to be considered in the design of object-oriented programs.

Some of the statements related to this category provided a slight change in emphasis by linking object-oriented with an emphasis on the use of objects, usually in the context of an integrated development environment. The supported language need not be object-oriented but it is assumed to be by those who take this understanding.

In this expression, creating an object-oriented program is all about writing a sequential set of instructions that glues together the use of pre-existing framework objects. This view is expressed by the participant who said:

> "I did learn that you still needed to know some procedural understanding of coding because that is where you go to, to make the methods do things but you can actually do an awful lot of physical things within your program by clicking and dropping into place and then applying a particular attribute or method around that." (I17).

This perspective sees the development environment as defining the object-oriented nature of the program. It relies on the environment providing a way of building the interface objects and adding functionality via a 'drag and drop' style interface. The programmer writes procedural code to glue the whole thing together. This is further emphasised when the participant said:

> "the platforms that I have developed using object-oriented programming have been development programs that have interfaces. Like there is a design interface and code interface. … By actually seeing them within that I would know that it was object-oriented" (I17).

Even when the interviewer questioned further to determine whether there was any understanding of being able to create one's own classes and objects, this participant held strongly to the view that the writing of procedural code was the glue between framework objects. The interviewer was left with the impression that this participant had not created any of their own classes or objects. Any that were created were those that the integrated development environment (IDE) provided for them.

Another participant expressed a similar view although showed some uncertainty with respect to the classification of the outcome as an object-oriented program. The difficulty

may be more one of being able to assess from the end result whether the writer understood what an object-oriented program was.

> "No, I don't think you need to have domain objects ... to be object-oriented. It depends on what... you need to have, ... Maybe, I am being biased by previous debates but ... if you look at the Delphi tool, it's object-oriented but people can use it to create code and RAD, using a RAD Visual Basic style of application building where they are not even aware that they are actually in an object-oriented tool or environment and it's object-oriented" (I18).

A programmer can use the environment to write methods to make the program work without fully understanding the nature of an object-oriented program or having to create any of their own classes or objects. Although this can lead to the rapid development of applications, the participant expressed some unease about the nature of the finished result.

With this category, there is some attempt to be able to identify a distinguishing characteristic of an object-oriented program. The base is seen as being sequential in nature but there is an emphasis on needing to use objects from an object-oriented framework. Some saw this framework being accessed from within an integrated development environment. Using objects in this category is seen more from the perspective of being a requirement of the language or development environment and not as a means for structuring one's own code. As a result, there is no emphasis on creating objects that relate to the particular problem space unless these already exist within the programming language support frameworks or development environment.

N2) Written using an object-oriented framework

This category has a number of forms of expression (subcategories). The primary aspect focuses on the use of classes and objects. This is emphasised in the context of using some form of pre-existing object-oriented framework in the creation of the program. The variations are expressed as an object-oriented program being based on the use of:

A) an object-oriented programming language and specific language features;
B) object-oriented concepts within an object-oriented environment; or
C) object-oriented concepts in a language independent object-oriented framework.

This category makes the move toward characteristics that are specific to object-oriented programs. It places minimal emphasis on the way in which the program is executed. It shifts the focus toward the fundamental aspects of an object-oriented framework or toward the expected features of an object-oriented language or system. Some specific aspects of object-oriented frameworks are named by participants, but the focus is on structural aspects provided by the framework or support system.

The three sub-categories are hierarchical in nature in the degree to which emphasis is placed on the use of object oriented features. However, the emphasis here is not on design characteristics but rather on the context in which these features are used.

N2A) Written based on the use of an object-oriented programming language and specific language features

In this expression, the object-oriented framework is seen to include those language features that are specific to class-based object-oriented languages. An object-oriented language and its supporting framework primarily provide these features.

The simplest expression of this view is that if the program is written in a language that is referred to as being object-oriented then the end program must be object-oriented.

> "if something is written in Java or C#, I would think that was an object-oriented approach of doing programming" (I29).

Another participant expressed it as a view held by students:

> "from the student's perspective they just simply see something as OO if it's done in OO language and that's the perception I get from them" (I28).

A stronger expression of this subcategory is based on the use of specific language features. A participant, who reviewed the code examples, decided that they would be object-oriented because they use classes, methods, and in particular private methods.

> "they are at least using a class, and methods or subs within that, ... They obviously have some private methods, … I'm not quite sure how they call them" (I21).

Some wanted to ensure that specific language features were used in writing the program.

> "There is the special features like I said, inheritance and then there's encapsulation ... it doesn't matter using private which means they are not going to be public to other users." (I31)

Note that the emphasis here on encapsulation is related to the use of language features that restrict access to the internals of an object and not to the more generic understanding expressed in the design characteristics categories. This participant also linked object-oriented with the use of classes and objects.

> "Have a look at the code and see whether there is any class and object being used in that code. Probably from there I can tell whether this is OO programming" (I31).

Some argued that an object-oriented language is a strong indicator but not necessarily sufficient on its own. For these participants, they are expecting the use of some features or some characteristics that would make the program object-oriented. This is reflected in the following statement where there is a question about the size of a class.

> "First, I would have a look at the language that it was written in. That would certainly help. Some people can use object-oriented languages but not necessarily use the object-oriented features. We could use C++ as an example. They might actually code it like C or they might code Java as one big huge class which doesn't actually represent a whole bunch of objects" (I3).

In this expression, language is a major determinant but an OO language imposes certain strengths in the way the program is written. It is expected that these strengths would show through in the structure of the code.

> "If they are using a language like Java, which is predominantly object oriented language, you can almost argue that if its written in Java then they are not really able to write in a non OO language in a OO prog[ram] ... assuming that they don't do something grotesquely wrong like including

> everything in one big method or not splitting things up incorrectly. … But most of the time it ends up being a good OO system because they've written in the OO language and they're going to do it, then it has to adhere to the character strengths the language imposes for that" (I28).

This last participant's response demonstrates the assumption of this subcategory which is that the language imposes a way of writing a program. That imposed structure is using object-oriented features (i.e. classes, objects, inheritance) so if they use the language feature then it must be object-oriented.

N2B) Written based on the use of object-oriented concepts within an object-oriented environment

The expression of this subcategory argues that to be an object-oriented program, a program needs to be written using object-oriented concepts within an object-oriented environment. The environment, by its nature, forces the use of objects and basic object-oriented concepts and provides the mechanisms to create objects, to manipulate them, and to store them. Such environments are based on an object-oriented language and include facilities to write and manage the code. The strongest argument included the need for an object-oriented database within the environment.

> "An OO system is one that the data is stored as objects, the code is written in objects, the principles of encapsulation, polymorphism, data abstraction, generalisation, inheritance, […] The objects should be, stored in an object-oriented database, not a relational database with a proxy layer over the top of it that converts things to objects" (I12).

Notice the emphasis placed on object-oriented concepts and generic concepts. The use of these is seen within the context of the object-oriented environment. This is emphasised further

> "in an object-oriented system that is truly object-oriented having created the object, you've created your procedure, and you've created your database. But it's no longer a database, it's an object base. And the object base contains the process and the data" (I12).

For those holding this view, it may be possible to write object-oriented code outside of an object-oriented environment but it is difficult to do so because of the lack of object base support. This expression promotes the idea that to be truly object-oriented involves more than simply using object-oriented concepts or language features. It requires an object-oriented environment that preferably supports an object base. Others clearly saw an object-oriented environment making it easier to create object-oriented systems but they did not insist that to be truly object-oriented requires an object-oriented environment.

N2C) Written based on the use of object-oriented concepts in a language independent object-oriented framework

This subcategory recognises that the object-oriented frameworks can be implemented in any language. This includes the techniques such as objects, inheritance, and message passing. It might be argued that these are foundational object-oriented techniques, as they are seen as techniques upon which software can be built rather than facilities provided by a specific type of language. One participant stated this view clearly.

> "Alan Fowler writes Fortran programs in any language, ... so I guess you would be looking for solving design problems by constructing objects at run time using dynamic dispatch or whatever you want to call it and message sending using compositional objects to solve those kind of problems. That's what I would see as the core of the OO designs" (I13).

Another participant described a book which provided code that was an example of writing an object-oriented program using a non-object-oriented language.

> "He basically put all his data into structs and he used pointers to reference the structs. And […] I realised that […] he actually had naked objects in a way. Because it was C, he couldn't protect them […] but he was still using the principles of object-oriented development to create partitioned applications with the code flow being determined by what the struct was and what his contents were because he used method pointers as part of the struct" (I18).

Another expressed it slightly differently, but with the same emphasis on language independence. Here there is also an emphasis on the ways in which an object-oriented language can be used to create non-object-oriented programs. The key issue is design.

> "you can write fairly reasonable object-oriented design in C if you separate the interface from the header files to the implementation from the C files and get a reasonable object-oriented design and you can write crappy non-object-oriented code in Java" (I24).

In this subcategory, object-oriented programming is a way to structure a program independent of the programming language. The language does not determine the approach to programming; rather the language makes it easier to follow one approach or another.

This category sees an object-oriented program as using some sort of framework or techniques for programming. The use of a language that includes an object-oriented framework or object-oriented programming techniques makes it easier to do this style of programming, but it is neither a prerequisite nor a guarantee that programs created are object-oriented. For a program to be object-oriented, it needs to use the framework and techniques. For some participants, these techniques need to be used in a way that is consistent with object-oriented design. This distinction is shown in the design characteristics outcome space.

N3) Based on data types

The aspect that distinguishes this category from the other categories is the way in which objects are used within the program. This category highlights a view that sees object-oriented programming as an ability to extend data types provided by the programming language. It represents a data focussed view of how to write programs. There is little concern about the flow of control in describing the characteristics of an object-oriented program.

One participant argued that this is the basis for handling complexity in large programs.

> "Object-oriented design says structure your program around the types of information you've got. For each type, build a module of your program that

> deals with that type of data […] and that turns out to be a very good way of handling complexity of large programs which is the core issue in software engineering." (I22)

The participant further clarified this based on language implementation when he said:

> "Common Lisp object system where objects are basically data types or instances of data types and you have generic methods that will be dispatched and inheritance and stuff. But […] the Smalltalk, C++, Java family, the idea is you can have classes which are types with attached methods" (I22).

In this perspective, the use of classes within the language is linked with the data type system. If the programmer writes a class then they are defining a new data type. This is based on the principle that abstractions within the code are based on data types.

> "How was this program broken up into modules? What are the abstractions? And if the abstractions are based on the data types, on the chunks of information that the thing processes and if that's where the boundaries were drawn around those then I'll say it's object-oriented" (I22).

The concept of abstractions is an issue raised with respect to design characteristics but in this context, abstraction and modularisation are clearly linked with the defining of new data types.

The orientation towards data types is also partially represented in the swing toward model-driven architectures with its emphasis on "information engineering". Here the participant talked about the structure of the model for the generation of the application being based on the data and the relationships in that data.

> "What's manifested across the domains is the data structure or parts of the data structure. So by focusing on the data structure and the structural elements of the design, you are able to generate a manifestation of that data structure into an application tier in the form of Java or into a database tier in form of DDL" (I26).

Notice the emphasis was on the data being common across application or problem domains. This practitioner was very focused on the business application of the code and less on the technical structure but in emphasising the business requirement in terms of information or data, he was placing a structural characteristic on the resulting code. In his emphasis, he devalued the behavioural characteristics as secondary to the data requirements and emphasised that getting the data requirements right is essential to good design.

> "It's a realisation that information engineering techniques are the core to any development activity because essentially the realisation is its data and not behaviour that's the core of the problem. […] If you get your data structures right, it just makes ... so many more things easier" (I26).

In model-driven architecture, the data structures are guiding the development of the application, although not simply in terms of the addition of abstract data types. Having identified the key business objects, the strategies for management are determined using design patterns as a guide.

> "Service objects are very important because that is where you define patterns. Once you have got core data structure you then have to understand how they're created at runtime and how you find them. And then you start employing patterns like factories and abstract factories" (I26).

In this view, it is not sufficient to use object-oriented concepts and techniques. The concepts and techniques are used to extend the data types available in the programming language. The object-oriented techniques make this type of extension easier, or in the case of model-driven architectures, they make it easier to generate the software from the data model.

The model driven approach to software development places an emphasis on the development of a data-driven model by focusing on the data requirements and building a model based on these requirements.

> "Model-driven development is very much focused on data structures and capturing the underlying data structure of the domain" (I26).

This places an emphasis on the process of development (i.e. developing a data model). This model is implemented through abstract data types.

With this category, there is a shift away from a focus on flow of control and more emphasis given to the nature and use of the objects. This emphasis sees the objects as extending the types of the language and as a result, the program representing a data model of the real world or problem domain.

N4) Based on interacting entities

Where the previous category focused on the data aspects as the way to use objects, this category returns to a focus on flow of control and focuses on the use of objects or components in interactions to implement the behaviour of the system. It sees a program or system being based on the interactions between objects or components. Flow of control and the sequence of execution are dependent on the messages passed. There are two types of entities represented in this category and this leads to the two subcategories. These are based on:

A) object interactions; and
B) component interactions.

Component interactions include the concept of object interactions. Component interactions are also closely linked with the understanding of what a system is.

As well as talking about an object-oriented program as interacting objects or components, there is an emphasis on a program being a model of the real world. This is reflected when a participant said:

> "A program is something that […] models some system that did a whole bunch of things to a bunch of objects" (I3).

This is more clearly related by another participant who argued for the use of metaphors to illustrate the interacting objects understanding. This participant recognised the limitations of using metaphors but

> "mapping it onto people in our real world, the metaphor carries very far actually […]. And so starting to think about programming in terms of

> interacting objects is the most important step and everything else can follow from this. So in practice and teaching that means to talk about objects as if they were people. I use a lot of metaphors where people have interacted" (I20).

This is reinforced in terms of modelling when the participant argued, that for object-oriented programming,

> "Programming first and foremost is creating a model of the world and object-orientation is the tool that you have to model the world. And almost as a side effect, almost by accident that model is executable" (I20).

For this same participant, the model was based on the roles and responsibilities that the objects have within the system. Responsibilities in this context relate to the behaviour of the object with specific emphasis on what it is responsible for doing and knowing.

> "From the problem definition, you define a conceptual model that is how you see in abstract terms the object involved in processing this problem. And at that stage, roles and responsibilities get assigned to objects in the stage of the conceptual model. And so then […] you have a model of what objects exist, how they interact, and what their various responsibilities are" (I20).

Another participant was more explicit in terms of what an object is supposed to represent within an object-oriented program. For this participant, there is a strong link between objects in the program and objects in the real world.

> "The class from which the object is derived is the direct modelling of the world so there is a concept like customer and the class is the representation of all customers. And then an object that is instantiated from that class represents a specific customer. So in terms of the domain, the object represents something specific in the domain" (I23).

Another participant also related objects communicating as being a model of the real world. This participant talked about it in the sense of helping someone learn to write object-oriented programs. He said that:

> "finding problems where it's natural for the students to see [objects], helping them to see the benefit there might be in taking that step to have objects communicate rather than to have one big method do the job" (I8).

The emphasis here is on identifying things in the problem domain that are natural to implement as objects that communicate with other objects to achieve what is required. There is an underlying implication of being a model of the real world.

Later in the same interview, the participant talked about eliminating long methods through teaching refactoring practices. In this context, he says:

> "making a kind of object they didn't see in the world so that something could be tested, some strategy object; strategy's something that lives in the head not something that's in the world necessarily" (I8).

The implication here is that although the real world may be modelled to give the initial starting point, it may need to be adjusted to improve the design by introducing behavioural abstractions that have no direct equivalent in the real world. Modelling the real world is a starting point but it shouldn't constrain the objects that are used in the model.

The link with a real world model is stated more clearly by this participant when he said:

> "It's a case of whether you're building a model of part of the world that some of the objects in your system will likely resemble, leak into the model so … in order for your ability to effectively use simulation, then … the object will bear some resemblance. But object-oriented design, good modularity's going to necessitate creating objects that don't exist in the world that are only for the convenience or for somebody's intellectual idea of the world but it's not actually out there" (I8).

As well as acknowledging that not all objects in a program will represent things in the real world, this participant was clear that not everything in the real world will be translated to an object within the program. The program may model the real world, but it shouldn't be constrained to real world entities or necessarily have to reflect the real

world accurately. Part of the selection is based on the behavioural requirements of the system.

The behavioural focus of the model is more explicitly stated by another participant. Here the link with a model isn't explicitly stated but the behavioural focus is quite clear.

> "A program is a description of the objects and their behaviours and also a description of what it is that you want those objects to accomplish" (I14).

This theme of modelling or reflecting real world objects reappeared in the discussions on interacting entities.

N4A) Object interactions

In this sub-category, the emphasis is on the interactions between objects rather than what the objects represent. This could be described as a focus on the behaviour of the objects in the system, or the implementing of algorithms through the interactions between objects. Often this category was stated bluntly, as in:

> "An OO program as a whole is a collection of communicating objects" (I23).

Or in the context of a system as:

> "I talk about this web of interacting objects which is in fact a system" (I14).

This participant expanded on the idea in terms of his objective in writing software.

> "One of my goals when I build a piece of code is to write simple classes with simple methods and have the complexity of the software in the interactions between the objects rather than implemented explicitly in the methods of the objects" (I14).

This can be contrasted with the previous category (based on the use of data types) where a participant talked of dealing with complexity through the use of new data types. Here the focus on dealing with the complexity of the software is on the interactions between

the objects (the way that they work together to achieve the desired result, the behaviour) rather than the data.

Another compared objects and their interactions with those of molecules.

> "objects are a lot of fine grained things interacting with each other generally ... and objects are kind of like molecules" (I18).

Molecules interact to form new substances and build complex systems. This is reinforced by another participant who saw the interactions as critical in achieving anything with object-oriented programming.

> "It is the interaction between objects […]. If this system consists of many different objects then this message passing is the way that all these different objects communicate with each other to achieve the goal of the entire system" (I29).

Others provided additional detail and clarification with a focus on the behavioural aspects that the objects provided.

> "The idea how objects interact to solve a problem is one of the early key ideas. You know that an object is asked for a service and it may ask another object to provide such service, that it's interaction between objects" (I20).

The interaction isn't simply to retrieve data. The interaction is to accomplish something. It delivers a service or some functionality, some desired behaviour.

One of these participants, in talking of the challenges of writing true object-oriented programs, focussed on the idea of the algorithm being represented in the interactions between objects rather than being contained in the methods.

> "Real object programs don't put the algorithms in the methods, they put the algorithms in the interactions between the objects" (I14).

This is reflected in an approach to teaching based on metaphors. In this case, an object-oriented program is seen as similar to the interactions in a restaurant.

> "So in practice and teaching that means often to talk about objects as if they were people. I use a lot of metaphors where people have interacted. Where you're a customer in a restaurant, you talk to the waiter, the waiter talks to the chef" (I20).

Each of the workers has a task to perform and together they provide the services for the customer. This participant saw this as reflecting what objects are doing in an object-oriented program. Each object has a service or behaviour that it provides to the system. This participant went further and argued that the interactions are based on a behavioural-driven model of the problem domain.

> "An object is entirely defined by its behaviour. Data is secondary. Behaviour is what makes an object. Behaviour means which messages it understands or in Java terms which methods it has and how they behave, what they do […] Fields are entirely driven by behaviour. You introduce fields to support the intended behaviour" (I20).

This concept was also linked to the ideas of recursive structures and nested virtual machines.

> "One of the things you have in OO is the sort of recursive idea of an object so your system is recursively composed of objects and the relationship between your system and then the actors […] in the outside world is the same between a particular object in the system and its real object communicate. This is Alan Kaye's idea of an object as a recursion of the computer itself. Object-orientation builds on and amplifies Dykstra's model of nested VMs because you have got that recursion there" (I13).

This view is present in the interpretation of an object as a set of machines.

> "They had well compartmentalised all the behaviour and associated state into objects. [...] Object-oriented programming is making all these objects and then just sort of setting them going and off they go and do their thing. […] Making the object is like making other little machines. All of the classes are like how to make a machine and then in the program, you just make a bunch of all these different type of little machines and you start them

> going by calling a method on them and that calls another method on another one" (I5).

Another approach to understanding this interaction nature of an object-oriented program is drawn from event-driven systems.

> "There's a very clean connection, mapping between OO programming and event-driven input. […] Event driven input typically involves graphical entities on the screen which will respond in different ways" (I22).

This participant expanded on this idea by contrasting the event driven style with functional decomposition of procedural programming.

> "In non-object-oriented, a program was instructions to do something. [...] We rapidly got into functional decomposition into procedures. […] Program is a specification of what the computer should do in response to different situations. […] This leads into the event-driven programming. […]When you define a class, it's a collection of responses. […] A program is a specification of how to respond to different requests" (I22).

Object-oriented programming was seen as a way for decomposing the problem space into smaller problems and recognising the relationships and interactions involved.

> "Object software is software that uses a particular decomposition strategy for breaking up this large problem into sub-problems, and sub-sub-problems and sub-sub-sub-problems based around both the division of data and division of operations or services and the relationships between them and the communication between them, one the relationship defines" (I28).

The description of the interactions and relationships between objects was seen as being core to the development process.

> "The objects have to communicate with each other to achieve a task. […] It's not just a fact they are identifying objects but they're identifying the relationships between those objects" (I28).

In this subcategory, the aspect is that a program is broken up into objects that interact to achieve the desired objective of the program. As one participant put it, the algorithm is represented by the interactions rather than the logic contained in the methods. This is a major difference from the "sequence of instruction" category and also from the "data type" category.

N4B) Component interactions

Some participants were less interested in the object as a unit of interaction. Their focus was on the component. In some cases, this was seen as a collection of objects but this was not necessarily the case. This perspective was more prevalent when the discussion talked about systems rather than programs.

> "To me essentially a component is just a very coarse grained object. Objects are a lot of fine grained things interacting with each other generally and a component is a super, coarse grained wrapper around them that is a whole unit of something or other like a brick wall and objects is kind of like the molecules or whatever" (I18).

Another participant was less interested in objects but was more than willing to discuss components. The emphasis here is not just the interactions; it is the ability to replace a component with another component and still have the system or application work.

> "The people think of components as being any object and I think that's a bad thing because […], the whole goal of component software is that […] you want it to be like hardware components, I can plug in an IC and can pull that IC out and I can plug boards into a computer and different components. I can have different CD-ROM players or DVD players and all these components have clean interfaces and I can substitute one for another in the hardware" (I30).

With this sub-category, there was difficulty distinguishing the difference between a system and a program. Both were seen as interacting components.

> "A system; in terms of software, I'd expect it to be sort of an interacting […] set of units that work together to achieve some result so much like an object-oriented program" (I5)

or

> "A program is a collection of communicating objects and then a system is when you have communication across program boundaries either quite directly or via files or something like that, like shared files. One is writing to a file and another one is picking it up and reading it and so on. All communication across machines that is the intuition, I think, of a system" (I23)

Here the program or system is created through decomposition, not into functional units but into units that will communicate with each other.

> "When I'm doing object-oriented development, the main task that I'm consciously doing is decomposing the system and decomposing it so that operations and attributes are related close by and that I can export or I publicise particular services which are available to other people and I can draw the communications and that. So I guess, the main idea is, you know, decomposing it" (I28).

Interaction is the key aspect here in terms of what an object-oriented program or system is, but there is increasing emphasis that this interaction is not just the exclusive domain of objects. The interaction can be between larger components. Components are larger units but could also be an object. In this sense, this subcategory includes the interacting objects subcategory.

N5) Artificial construct

This category highlights the difficulty with using the program and system terminology. In many interviews these terms were used interchangeably and it was only when interviewees were asked specifically about how they describe a program or a system that the problems were highlighted.

In a very direct response, one participant saw a program as something clearly originating from the requirements of the operating system and not really of interest to the software developer.

> "I don't know what a program is. I mean a program is very much a completely uninteresting idea that is enforced by certain languages and operating systems, like Unix. The Unix operating system, for instance, a program is some thing that can be executed by exec. Other operating systems are just collections of functions and classes and sort of live inside the execution of a program. So the concept of a program in that case doesn't make sense. So I think the concept of program is essentially a ridiculous invention that's necessary by very restrictive languages and operating systems" (I24).

Another participant struggled to distinguish between a program and system but saw the influence of operating systems on determining what a program is.

> "You can have a system made up of programs but... program's more a procedural term and more a unity of purpose probably than a system. A program tends to do one thing more than a system might be suitable for. But a program can do lots of things. […] The program tends to do one job. […] Whereas a system might be made up of more than one program and they might start and stop at different times. […] Tend to think of a program running, starting and stopping. Where a system, I tend to think of as existing and stopping being not something that routinely does." (I19).

A program is something expected by the operating system. The operating system starts and terminates the program, but a program is not seen here as being integral to object-oriented. The operating system requirement is seen as a restriction that has been removed by some environments.

> "Once you have the implementation of your model, you need a way to connect it to your operating system so that you can kick off the execution. You need some construct that connects that implementation to your operating system so that you can do something with it. And the fact that you

> can do only one thing with it and that is start the program and everything else is internal is truism of a long history. And the fact that it is via this "static void main" is even more so an accident of history. That thinking has clearly influenced BlueJ where that restriction is removed, where you interact with your model much more flexibly. I can interact with any object by modelling in any way I like or any way that the object permits. And then we can reduce it down to the traditional view but only because that's what people expect, not because it's logical" (I20).

The artificial nature of a program is related to the history of computing and is seen to have been removed by more flexible environments.

> "A program in C today is a completely artificial invention that was needed because the linker had to do some static linking in order to produce an executable and that became the definition of a program and so have to have a main function so that when the operating system starts this chunk of code, you know where the control goes and stuff like that. If a Common Lisp developer, like I am, my Common Lisp image is just a collection of functions and classes and other interesting objects and I can't point to some thing and say "this belongs to that program". I can run maybe half a dozen different applications in my default image and they share some classes. They might share some objects. I don't know what's a program" (I24).

This participant's development environment removed the artificial boundary of the program giving greater flexibility to reuse and to structure the applications or functionality that he is creating. The contention here is that the notion of an object-oriented program as a container for a set of objects is restrictive. An object or set of objects (i.e. component) should not be constrained by artificial program boundaries. Environments like Common Lisp remove this distinction. The same objects and components could be used to solve a number of different problems and be linked together in different ways. As a consequence the notion of an object-oriented program becomes a restriction on the way of thinking about how components and objects can be used.

This focus on reuse is also reflected in the view that true components should be like integrated circuits and hardware components. The unit can be replaced or upgraded without throwing away all of the system. This is reflected in

> "the whole goal of component software is that […] you want it to be like hardware components, I can plug in an IC and can pull that IC out and I can plug boards into a computer and different components. I can have different CD-ROM players or DVD players and all these components have clean interfaces and I can substitute one for another in the hardware" (I30).

This view promotes the idea that development isn't about programs but rather it is about reusable components. This participant took this further to argue for a new way of defining the interface to a component or object. He called this a collar interface based on his hardware understanding. He said:

> "So the goal is that you should be able to draw a line around the component and understand all the things that are coming in and going out of it. So all the inputs that come in and all the outputs that go out; that's standard hardware. Every pin is either an input or an output or sometimes both and you can exactly define what the inputs and the outputs are. And the curious thing about object-oriented programming is that while they claim that they're really interested in interfaces, they've really only handled half of it. They really only define the interface into an object and while they do have the interface of the result of the object, what they are missing is that any object can arbitrarily call some other object internally and there is never any definition of that interface" (I30).

The objective for this participant was to be able to define reusable units that could be easily extracted and replaced. A program or system was simply composed of reusable components. He emphasised this theme further when he talked about the assumption that a program is started and terminated. He argued that the concept of a program interferes with an understanding of modern system requirements where a system may need to be operating continuously. This requirement for persistence is seen to conflict with the notion of a program that starts and stops.

> "To most software people, a program is an independent entity and you start it up and it runs and it terminates and I think that's a wrong definition of a program and we're actually seeing how that's causing problems because there are a lot of things that are very persistent when you're writing software now" (I30).

The first example given related to the use of a web site where he argued that

> "when you go to a web site, it doesn't start executing then answer and come back. Web sites are very persistent" (I30).

In comparison, he argued that programs and objects have a life cycle. This means that they are not well suited for persistent operations.

> "Classes always have constructors to build your objects and so the life cycle of your objects is always very important and people think the same thing with programs. Programs have a life cycle; they start up, they get the resources they need, they execute, they consume memory while they're executing and then they somehow leave things in some external persistent store… And then they stop executing" (I30).

This participant also cited other examples of persistence, including databases and hi-fi systems. The participant sought instant responses based on being ready to respond at any time, and argued that this is not the case with the concept of a program. He contended:

> "Programming in general has always been this sort of transient thing where you have an instance of a program that exists on disk but that's somehow different from the instance of that program as it exists in memory. So when you instantiate a program, you start executing it, the form of that program changes dramatically and in fact the data structures you use are very different" (I30).

The contention here is that the concept of program is restrictive and not offering what is desired or needed. A new approach is required that is instantly available and ready to respond to the users needs.

Along with the notion that a program is an artificial construct, there is increased emphasis on the model being based on useful abstractions. This shift was seen in discussion in the previous category in relation to models where some real world entities might be excluded from the model and other objects added that enhanced the ability to implement the model. This perspective is also present in this category.

> "You could get that structure but I would expect some of my domain phenomena to be reflected in classes but there would be other classes that are not necessarily representing that. But certainly, I would expect my domain objects to be represented as collections of classes, it might be a hierarchy of classes and even distinct hierarchies of classes if that's justified from an implementation point of view." (I24)

This category sees the notion of a program as having historical roots. It relates to what a computer system has done historically, and not necessarily to what type of system is now desired. A system provides functionality or data at the press of a button. The notion of a program places an artificial barrier to the desired interactions and reuse of the functionality. In essence, there is a desire to have objects or components that can interact across boundaries and be used flexibly to construct systems.

5.1.2 Critical aspects for the nature of an object-oriented program

The discussion of these categories has focused on three distinct aspects. The two most visible aspects are the flow of control and object usage. The flow of control has influenced the meaning associated with two of the categories. The influence of how objects are used helped separate out the remaining three categories. Behind these two categories is the nature of the problem solution. These aspects are represented in the following table (Table 5.2).

The influence of the flow of control aspect was the first aspect picked up during the interview and analysis process. Participants clearly made reference to both the sequential nature of the flow of control or of the interacting entities. An additional emphasis was that some referred to interacting components rather than interacting objects.

Category	Referential	Critical aspects		
		Flow of control	**Object usage**	**Problem solution**
N1	Sequence of instructions	Flow of control is seen as sequential	Objects are used from provided framework	Provides required functionality
N2	Written using an object-oriented framework	Flow of control is still seen as sequential but isn't a major focus	Programmer objects are defined and used but primarily as a requirement of the language or framework	Provides required functionality
N3	Based on data types	Flow of control is not explicitly discussed or considered an issue	Objects are defined based on data requirements	A data model of the problem space (real world model)
N4	Based on interacting entities	Flow of control is defined by the interactions between objects	Objects are defined based on behavioural requirements	A behaviour-driven model reflecting objects in the problem space
N5	Artificial construct	Flow of control is defined by the interactions between objects	Objects are liberated from the confines of program boundaries	A behaviour-driven model using useful abstractions

Table 5.2: Nature of a program critical aspects

The object usage aspect was partially anticipated in planning the interviews. As reflected in the interview schedule, the primary concern was whether the participants focused on the language using objects to modularise the code, or objects as behavioural units. The data versus behaviour distinction became more obvious as the interviews progressed and was often associated with the idea of a program being a model.

The classification with respect to the problem solution is based on the examination of the most prominent theme in the transcript. Two thirds of the participants discussed the concept of an object-oriented program as a model of reality or as a useful abstraction. Participants, who had the view that a program is a model, tended to give the answer of a model in response to the direct question "what is an object-oriented program?" A program as a model of the real world was the response given by the following participant:

> “The standard phrase that everyone would use "models the real world" but we just talked about that.” (I1)

Only one participant, whose transcript had aspects in “sequence of instructions” (N1) and “written using an object-oriented framework” (N2), mentioned the concept of a model but this participant also talked about interacting entities. The mix here was regarded as not significant enough to warrant the model aspect being associated with the lower two categories.

Clearly this critical aspect needed to be seen as part of the nature of what an object-oriented program is. The approach taken was to classify each transcript in terms of the emphasis on the other critical aspects and then to assign the variation in the model critical aspect appropriately. This led to the positioning as shown in the table above.

5.1.3 The design characteristics of an object-oriented program

Interviewees, in response to the initial question on “assess whether a program has been written using object-oriented techniques and practices,” talked more of the design characteristics that would be used in the construction of an object-oriented program. The novice practitioners (post-graduates) focused more on structural descriptions. The experienced practitioners were more abstract and less dependent on specific terminology and concepts. This is reflected in the categories of description obtained from the interview data (see Table 5.3).

In discussing these design characteristics, the concept of an object-oriented program as a model of reality or a useful abstraction was evident. If design characteristics are seen as the ‘how’ of object-oriented programming, then the connection with an object-oriented program as a model forms the linkage to the more ‘what’ focused aspect of the nature of an object-oriented program.

In contrast to the previous categories where participants clearly stated their perspective, these categories had to be drawn from comprehending the emphasis in the interview. It was noted that participants fluctuated during the interview placing emphasis on different characteristics, or made limited use of terminology from a different category. There were also indications that participants may see some of the terminology as paradigm

specific, rather than as a more generic concept that has applicability to other programming paradigms.

A participant may have been reflecting a specific way of looking at object-oriented when they expressed a particular category within this outcome space. For example, a participant said:

> "I mean Alan Fowler writes Fortran programs in any language, so I guess you would be looking for solving design problems by constructing objects at run time using dynamic dispatch or whatever you want to call it and message sending using compositional objects to solve those kind of problems. That's what I would see as the core of the OO designs" (I13).

The focus of this comment is on the coding level and the technical implementation. At this level, it makes sense to talk about "constructing objects" and "dynamic dispatch". Later in the interview, the same participant said:

> "I would emphasise just things like domain modelling. Try to be clear about what it is you are doing, what it is that you are modelling. I would emphasise there are […] generally not any particular solutions to any kind of real world problem. There is really going to be more solutions than there are designers or programmers" (I13).

Here the participant has focussed on design issues; the language has changed to talk of "domain modelling". As a result, the categories in this outcome space may reflect not simply a level of understanding but rather different perspectives or ways of viewing the same phenomenon within the object-oriented community.

To arrive at the design characteristics categories, the analysis looked at the overall emphasis placed on the use of terminology. It is the variation in emphasis and the changing terminology with respect to the design of an object-oriented program that distinguishes these categories. In many cases, the emphasis was clear because of the way the participant talked about the aspects.

These categories use a number of constructs, principles, qualities and thought processes that need further analysis to uncover the understanding that participants have of them.

The naming of the constructs, principles, qualities, and thought processes should not be assumed to have an obvious meaning. In the following discussion, some of the difficulties in the use of terminology related to these concepts are referred to.

Cat	Referential	Structural
D1	Is language dependent	This category represents a view that an object-oriented design is dependent on the use of the class construct in an object-oriented language.
D2	Uses paradigm constructs	Uses specific object-oriented concepts and constructs
D3	Uses generic principles	Uses generic principles and concepts that have wider applicability
D4	Applies generic design objectives	Uses design qualities that are general objectives for all programming
D5	Expression of thought process	Sees the object-oriented program as being the result of a specific thought process that addressed particular design issues and programming problems

Table 5.3: Categories of design characteristics

D1) Is language dependent

This category reflects similar thinking to the second category of the outcome space 'The nature of an object-oriented program' ("Written using an object-oriented framework" (N2)). The emphasis, with respect to this category, is that the use of specific language features made the program object-oriented. At the lowest level, this was simply the use of class or inheritance constructs.

The use of this category as the foundation for this outcome space is to emphasise that some participants focussed primarily on structural components with limited discussion of design characteristics. This focus usually was at a low level of structure such as language dependency rather than the application of object-oriented concepts using the language.

This language focus is reflected in the following participant's comment where the emphasis is on specific keywords rather than the object-oriented concepts implemented by those keywords.

> "If it was a language I was familiar with, I would look for the keywords. The class construct if the language supported it. I don't actually know if I've come across any languages that use objects that don't specify classes. The keyword "this" is usually a good give away. It's a lot of syntax I would look for rather than semantics" (I10).

D2) Uses paradigm constructs

The emphasis in this category is on the use of specific object-oriented constructs such as classification, objects, object identity, state, behaviour, interface, message passing, relationships, inheritance, and polymorphism. There may also be some emphasis on encapsulation as a way of constructing valid classifications for objects through the combination of state and behaviour together to hide the implementation details from potential users. In some cases, the participants discussed these constructs in terms of the technical implementation details.

For some participants, the use of paradigm constructs was reflected in the use of specific keywords or constructs such as "class" that the participant associates with object-oriented programming. However, this was not in terms of language constructs but rather in terms of object-oriented concepts. Here the participant was relating how they might determine whether some fictional program might be object-oriented. In this case, the participant had not been provided with the code examples

> "Class definitions are going to be a bit of a give away that it's object-oriented but if there's no class definition in the code you've given me to look at, I'd be looking for the actual creation of objects to do work or I suppose, if something was being passed into a method and I could see that there were methods being called on that, I would assume that it was object-oriented" (I5).

For this participant, the keywords gave an initial indication but they also wanted to see that the code was actually making use of objects and not simply using the keywords. Although this next participant would be looking for keywords ("tags or labels"), the wording choice of "instantiate, collaborate, realise" suggests a particular style of usage rather than simply usage of those keywords.

> "OO would have tags or labels that I would associate with OO. So words such as class, object, instantiate, collaborate, realise… what other words would I see?" (I12).

Some wanted to be more specific about the concepts that should be associated with object-oriented programming. In the following cases, it is inheritance and relationships.

> "Programs written in it are designed with object-oriented programming in mind in the sense that they have classes, they have inheritance between their classes, there are relationships between data and everything, and there is some behaviour there but it's always declared in terms of functions" (I1).

> "These are the basic kind of building blocks of a good object-oriented program; for inheritance, delegation, interfaces, dynamic binding, encapsulation" (I11).

> "I think the classic object-oriented things as being encapsulation, inheritance, polymorphism" (I19).

The use of object-oriented techniques should be more than simply being used to group things together. There should be explicit usage in the construction of the program that is coherent and meaningful in relation to the programming task being completed.

> "If they're unrelated chunks just grouped in packages then it's not object-oriented. If there is a meaning like you can connect between that class and that class in many places and there is coherent structure rather than an unconnected structure then it's more object-oriented" (I1).

The use of a particular object-oriented technique was seen as a requirement for object-oriented. If it was not used, then the program is likely not to be object-oriented.

> "You can extend that and you can start saying "Is there a sensible use of inheritance amongst those classes?" for example. If they don't use inheritance at all, then I'm starting to think it's been designed with a more procedural approach, more like C or something like that rather than making full use of object oriented ideas" (I6)

or

> "In a class based language, I'd expect to see people inheriting things and using polymorphism" (I7).

Encapsulation was expressed as an object-oriented concept, but in the context that relates it to the creation of objects and the hiding of internal attributes. Encapsulation's broader meaning was not recognised or considered.

> "I'd look for the use of object-oriented concepts particularly things like encapsulation is probably the most important one from my point of view. ... People aren't exposing member variables. The high level access is through accessors" (I19).

The emphasis on the paradigm concepts might be expressed in a way that connects with a higher-level understanding but with very limited focus on that higher-level of understanding. In this case, the participant saw a connection with the real world and of an object-oriented program being a representation of the real-world but there was not a strong connection with the ideas of modelling.

> "If we stick mainly to the general OO paradigm then there would be the whole idea of thinking about the world in terms of objects that carry some state and behaviour and same root. Probably this is the most basic aspect and then the various relationships between such objects and classes. Objects such as association and inheritance and then as far as inheritance is concerned it would be polymorphism, in terms of being able to substitute one class for another thus having flexibility in the design" (I2).

This participant was more direct at other points in the interview making comments such as:

> "In the real world, certain objects are related to other objects such as students are assigned or enrolled in papers so linking this paper object to student object in order to deal with this and we will have to reflect it somehow in our model of the business domain. We just say that all students

> can be associated to paper, can be linked to papers and this is the association." (I2).

Then, more bluntly in relation to the understanding of what an object is, this participant said an object is a "software presentation of real or conceptual objects in the real world" (I2). For this participant, the linkage between the use of object-oriented paradigm constructs and real world entities was the focus of object-oriented design.

Another participant acknowledged that, in creating an object-oriented program, the objective is to model the real world using the paradigm constructs. Notice that there is no specific reference to the quality of that model, although there is an implication here that this is a design process.

> "Start modelling the real world, how will you go around doing that? Without asking a question why would you do that? We would describe classes and instance but then you need to introduce the actual things like inheritance and stuff and that's where it gets tricky. I classify things but why do I need relationships between them. There is no real obvious relationship between many things in the world." (I1)

and

> "Object-oriented program will have a coherent model behind it. That model will have a reasonable classification of things into classes with relationships between them with just a little bit of inheritance of some sort or possibly interface inheritance. That model will have relevance to the problems being addressed" (I1).

In this participant's language, there was a desire to see a close relationship between the use of objects in the program and entities in the real world. Another participant expressed this same thought as:

> "Everything's an object and the objects within the system are a representation of those objects that are outside, that are the real world things." (I12)

Another saw modelling the problem domain as the starting point for developing an object-oriented program and attributes this to being a Scandinavian approach to software development.

> "You go out with the first cut of the domain model. The one thing I guess I believe in is using relatively straight forward modelling notations and techniques and processes because this is to help the people doing the analysis or programming to engage with the domain rather than having to worry much about the semantics of the modelling notation. I think object-orientation has a good a story coming out of Simula and Beta of the Scandinavian school. It's a very nice story about the correspondence between object in the program and object or concepts in the real world." (I13)

The shift that is evident in this category is away from an emphasis on language constructs and keywords to an emphasis on constructs or concepts that are seen as being unique to object-oriented programming. Some interviewees attempted to list these concepts with some degree of closure. The use of the paradigm concepts was based on using them to model the real world.

D3) Uses generic constructs

The shift in emphasis in this category is toward constructs that are more generic and independent of programming paradigm. These constructs include abstraction, composition, delegation, encapsulation, modularisation, reuse, and type. In this category, the object-oriented concepts are seen as tools for implementing the generic concepts; for example, classes and inheritance are used to implement abstraction hierarchies.

Within this category, encapsulation is used as a more generic concept that relates to the enclosing of a piece of code, that implements some concept, within a programming construct to hide the details of its implementation. Encapsulation in this context was not limited to the definition of classes, but could be used in terms of methods, packages, or any construct that enabled the capturing of a piece of functionality and / or data in a way

that enabled reuse. This use of encapsulation is the same as that used by Shalloway and Trott (2005).

Like encapsulation, there were obvious differences in the use of terms such as abstraction, modularisation, type, and design patterns. Design patterns were seen here as tools for constructing object-oriented programs rather than tools for assessing quality of design.

The use of the term ‘type’ presents its own set of problems. This was discussed in a number of interviews with the emphasis on type being defined by a class, through to a type being defined by a set of behaviours that might be defined by an interface or a class, or might be determined dynamically at runtime based on an object’s ability to handle a requested behaviour. The inheritance of type through class inheritance was also an issue in this discussion. This is an area where further analysis is required.

Modularisation was used in a number of different ways but two stood out in particular. One usage was in relation to functional decomposition and imperative programming, and the other in a more generic sense of combining code elements into larger chunks. In the case of object-oriented programming, these larger chunks are classes, packages, applications, or programs.

Abstraction is another area where there was considerable difference in the use of the term in the interviews. Abstraction was used with reference to different levels of the code such as classes, methods, packages; and also used in reference to different levels or categories of proximity to a perceived reality. This also needs further investigation and analysis.

Generic concepts were often discussed in connection with good design and in relation to other generic constructs. They were also linked with paradigm specific constructs primarily in the context of the generic concept being implemented through the use of a paradigm specific concept.

For this participant, the focus was on modularisation through decomposition to objects with the objective of improving reuse.

> "When you break down a system in terms of objects, you're getting parts that are potentially reusable elsewhere so we look at trying to minimise dependencies and we definitely do look at decomposition in the context of objects" (I11).

Encapsulation of data to form new "abstract data types" was also closely linked with modularisation. This was seen as a method of abstraction and as a way of "enforcing design as intent". Although referencing design in this context, this participant was not talking about design objectives, however these may be perceived as being present in the background.

> "When talking about abstract data types that's an opportunity to introduce some examples to show that if you don't encapsulate the data, it's open to abuse by programmers and so on. So it's not just that you're getting abstraction with abstract data types, you're also getting protection. You're enforcing policies in the software. You're enforcing design as intent" (I11).

This emphasis on generic constructs with an emphasis on reuse through design patterns was seen as a way of defining software development as an engineering discipline.

> "It's often difficult to defend software development as being an engineering discipline but the use of design patterns is basically reusing tried and tested design knowledge and that contributes to higher quality and reduced costs of software development and that's basically what engineering is all about" (I11).

Another participant, while focusing on generic principles, focused on modularisation and encapsulation.

> "The biggest thing about OO is the ability to partition applications in such a way that you can actually make them resistant to change in one side destroying something on the other side. The great thing about OO is that you can actually make granularity of those boundaries very small" (I18).

There was a reference to the ability to change the application through the localisation of change through encapsulation. This gives a slightly different focus to the use of

encapsulation compared with the previous participant, who was primarily thinking in terms of data encapsulation to provide protection from unwanted data changes.

> "Both of those two are the big things, the ability to encapsulate your application so that changes in particular parts of it are actually kept local and so you can test them locally as well without destroying the whole application and the other big thing is it allows you to mentally make the model of what the whole application is doing and understand what its doing" (I18).

A third participant also focused on modularisation, but saw object-oriented modularisation as assisting with reuse. However, this participant had not unlinked this thinking from the use of specific programming languages.

> "The main purpose of object orientation is very similar to modularisation. Therefore whether using C to do a really good modularised piece of code or maybe using Java to put them into classes, the main purpose is to modularise the code so that most of them can be reused without redundancy" (I29).

This emphasis on modularisation was seen as a generic practice for all good programming.

> "I would emphasise a lot of good programming practices such as making code modular and try to use as much of the object-oriented features of Java so that things like inheritance achieve the base part of object-orientation to achieve those good programming practices" (I29).

In this category, generic principles were seen as the objective of object-oriented software development. The object-oriented concepts are the tools to implement these generic principles and a domain model. The major focus was on modularisation and code reuse with some emphasis on these providing a base for good programming practice.

D4) Applies generic design objectives

With this category, the emphasis shifted to the application of design objectives. Some of the design objectives referred to relate to design metrics such as cohesion, coupling, and distribution of responsibilities. Others are more general design properties such as communicability, flexibility, maintainability, manageability of complexity, robustness, and simplicity. Each of these needs further analysis to uncover the understandings held by the participants.

One participant although not naming the objectives, alluded to them when they talked about object-oriented programming in terms of the style of the design.

> "To me, it's the style of the design of the solution or in a sense how the solution is expressed" (I4).

He expanded on this line of thinking when he placed emphasis on an object-oriented solution having some design objectives implicit in its design; one of these he stated in terms of ease of maintenance through changes or extension.

> "The first thing would be how hard it is to make changes to it. If I can take a system that's been written and add a piece of functionality [… and] if I can make a logical extension to it then it's done in a minimal amount of work. Its how easy would it be for me to extend or change that program" (I4).

This view was also reflected as being the essence of an object-oriented program. For this participant, a program that is difficult to maintain is not object-oriented.

> "If I'm given a program, if a change to that program is so complex and so large that I have to make all these different changes then I didn't really get anything out of it being object-oriented. An object-oriented program is going to be a program that I can change with a reasonable amount of effort or a small amount of effort ideally" (I4).

Later, he linked this objective with an object-oriented paradigm construct when he said:

> "Polymorphism is actually really important to me in my job because I'm able to plug in and plug out pieces of systems and I get that because of

> polymorphic behaviour. So the changeability of a system is increased because that decision of what module is called is made at runtime" (I4).

This paradigm construct was also seen as helping reduce complexity of the program as well as reducing the maintenance load.

> "So since I have polymorphic behaviour and I can put something different behind an interface and it doesn't affect the system, like the system doesn't care then I'm getting a huge benefit as far as it's not only the functionality of my program but the ability for me not to be hassled" (I4).

For this participant, the paradigm constructs were seen as tools for achieving specific design objectives. In this case, it is to achieve complexity management using polymorphism.

> "Polymorphism […] has something to do with complexity management and in a sense being able to deal with that" (I4).

The whole framing of thinking and focus for this participant was that through using object-oriented concepts, he gains the ability to manage complexity and consequently obtains solutions that are easier to maintain. He therefore saw maintainability as a key characteristic of an object-oriented program.

Another participant simply saw these as generic goals that object-oriented techniques and practices can be used to achieve.

> "If you mention techniques and practices, it is basically not something scientifically defined but something that is generally done. It is a matter of pattern rather than relating some logical expression. So if an application uses certain OO facilities to improve flexibility, maintainability and robustness, we should include design patterns. Of course design patterns would be to a large degree relying on some basic facilities such as polymorphism" (I2).

This participant related the achievement of these goals with the use of design patterns and paradigm principles or concepts. This association with generic constructs and paradigm specific constructs is closely linked.

Other participants saw the design objectives as ways of signalling a need for changes in the structure of the program.

> "You're really just starting to look for places where the program seems unnecessarily complex, where you have bad cohesion in a class, where an object has lots of instance variables, or sending a message to another object that has a lot of information to let it do its job, classes that are too large, methods that are too large. There are a lot of common characteristics of that sort that you can use to judge the quality" (I8).

A design objective is desirable and a program that does not have that quality is revised or "refactored" to achieve the desired objectives. The object-oriented constructs and concepts are simply the tools for improving the design of code and achieving specific design objectives.

Another participant recognised that it is possible to see that something is a bad design even though it may be difficult to provide good reasoning.

> "I looked at what had been produced for this Sokoban program and said "That's a bad design" and redid it as a different design. Why did I think it was a bad design? I remember thinking about it at the time, and I would still struggle to be able to do anything more than say "well, this little detail here is going cause this problem. This little detail here is going to cause this problem and this choice of class has this set of problems with it" So I can't classify it in any real sense other than "wasn't a good object-oriented design"" (I9).

This difficulty of defining a good design is further expressed in a specific example in relation to the best time to introduce a specific class:

> "There are other things that need to go on later on so question like "well what point do we introduce the time zone class?" "When do we get to a

> level where we say “Ok, up until now I haven’t needed one now I do”?” Do we do it as soon as the need for some representation of time zone comes in even all that is needed is we’re in daylight saving, we’re not in daylight saving or do we have to go a bit further down that track, Ok, now we’ve got different time zones in different parts of the world? When do we make that decision? And that’s a hard decision to make and I think it’s that kind of decision that leads to a good design versus a mediocre one and I don’t think that question is easily answered by “what concepts are in the domain thinking process?” (I9).

In this respect, this participant was questioning a simplistic domain modelling approach as leading to a good design. Good design is an attribute of being aware of when a domain construct should be surfaced to a program construct, or when classes should be added or removed to ease maintenance.

Design objectives are desired. What that means in terms of a particular piece of code may be difficult to access but, despite this difficulty, there are characteristics that even this participant was willing to name and attempt to describe.

> “This time example […], there is a point at which we get code associated with time zone ‘mixed in’ in most of the other methods in the time class. When we start seeing that, that’s probably time to simplify everything by extracting all of that out and putting it in a separate class” (I9).

Behind this participant’s discussion were the issues of modelling the problem domain but not precisely, and sometimes using abstractions that are not part of the domain. However, this participant did say that, in teaching, he seeks to use small applications that enable the learner to focus on modelling of domain concepts in those initial programs.

> “For most applications that are likely to be used in early programming courses, they’re sufficiently small that programming the domain models... domain concepts... representing them as classes is the only thing we need to do.” (I9)

Different participants placed emphasis on different design qualities. For this participant, the emphasis was on maintainability and the distribution of responsibilities.

> "That's sort of one of the key components of it, because you're not really using OO if they aren't passing the buck around. If it doesn't say ok, do this for me please and give me the result because a lot of the time if you just copying and pasting code all over the place, it becomes a mess and it's hard to follow and it becomes a nuisance when you're trying to read it and maintain it. But if you break down the responsibility then other people can work on it" (I10).

Maintenance and the distribution of responsibilities work together. Another participant started by emphasising simplicity of classes and methods, and placing of the complexity in the interactions.

> "In the last five or so years, I've also picked up on agile practice and so I try to maintain / build simple software. One of my goals when I build a piece of code is to write simple classes with simple methods and have the complexity of the software in the interactions between the objects rather than implemented explicitly in the methods of the objects" (I14).

However, this emphasis on simplicity is related to ease of maintenance and, in particular, the locality of change. It should be noted that this participant recognised the different types of locality, but does not attempt to argue that one should be used in preference to the other in all circumstances. There is recognition that there are appropriate situations for the use of each type of locality. The participant contended that "the locality of an object program is very different from the locality of a procedural program" (I14). The locality of a procedural program is described as:

> "You're at a situation in a program where there's a variety of things to do and some criteria that you can use to decide which those are. One solution is a switch or a structured if statement of some kind where you evaluate the criteria and based on the solution of that you choose what to do and you do it. So that gives you a certain kind of locality" (I14).

The decision making criteria is placed everywhere there is a need to make that type of decision. The location of the functionality is placed in line with the requesting logic, but this can cause the decision making logic to be duplicated. If there is a change to this decision making logic, then all the places where this decision is made need to be found and modified. The participant said:

> "The problem is that if there's a lot of possibilities, if this occurs a lot in your program then you have a maintenance headache because if you have to change one of these that's usually not the proper change; […] but if you have to add a new possibility, you have to go find all these switch statements because you've got one kind of locality in that solution but there's a different kind of locality that you don't have because theses switch statements are spread around the program and the tools don't help you find the one you want very well" (I14).

The participant contended that, from an object-oriented perspective, polymorphism removes this decision making by placing the logic in classes based on the criteria used in the decision. The participant said:

> "If you take the same program and build it polymorphically with an object-oriented solution then in essence you take that solution and you just cut through it in an orthogonal direction so that you have a method. If you have a triad of subclasses and in each of these classes you have a method for foo [sic] and you have a method for bar and this class structure gets implemented in various ways so that the first choice the foo [sic] gets in one of these classes and the first choice for bar in the same class. The second choice for each gets implemented in another class and the third... So you have a different kind of locality" (I14).

The contention was that the object-oriented solution is easier to maintain and does not increase the amount of code written.

Another participant started from the perspective that programming is about

> "a series of design decisions in some ways. There'll be a certain decision that says that there's a certain amount of coupling between these parts, or

> these are the parts and these are the connections between them. And other parts which will be sort of separated out so there's some sort of conceptual difference between them. And then there's all the mechanics as to how you actually realise that, how you actually code that, what decisions you make to actually get the thing to work. A program is a way to communicate something but at the same time they're also executable. So it's a very stylised form of writing" (I15).

These design decisions were linked with the use of generic and paradigm specific constructs, but the emphasis of the participant's comments remained on the objectives of the design through the separation of concerns or responsibilities. This discussion was part of a discussion on the capturing of events in a user interface. The participant recognised that the events could be all handled by the form, but argued that this is then confusing or mixing up different concerns.

> "It's a convenient name space that they are trying to inherit so that they can define a method to handle events as part of this whole frame rather than have that put off, separating out that concern into a different part of the system. So you get this mixing of concerns. You get a piece of code which has got some logic to do with the application and mixed in amongst it is all this interface logic and all this type of logic to do with laying out things on the screen. When it's all in one class, you know there's something going on that's wrong with that mix because they haven't managed to separate out these concerns and come up with an abstraction" (I15).

However, this separation can go too far so it is not a matter of simply following rules, but rather an issue of understanding the nature of a good design.

> "There is a certain amount of separation that you have to allow in order to allow the information to actually take place and you can make things worse for yourself by trying to artificially separate things that never should have been separated and you create these great complicated interfaces which just get in the way of the design" (I15).

Another participant picked up this theme of the separation of responsibilities when he talked about the specific implementation in Common Lisp. In this case, it is linked with modularity.

> "I can tell you how I would do that with Common Lisp because I can recognise the separation, the modularity issue; for instance which module is concerned with which particular aspect of the functionality. I can see how they separate those concerns between classes, between different methods" (I24).

This participant also linked this separation of responsibilities with the maintenance issue and the difficulty of fully understanding the need for good modularisation.

> "If you are to understand the ideas behind object-orientation, you need to understand the reason why you need APIs and modularity and so forth, and I claim (I've been doing this for about twenty five years now so I know roughly what I'm talking about) that there's nobody who doesn't have at least ten years experience of full time programming in industry could possibly understand the reason why this is important. So our students […] learned enough to repeat it at final exam but they have no profound understanding of why this is important because most of them haven't programmed 200 lines of code. Object-orientation is not needed for 200 lines of code. It's starts becoming interesting when there's ten's of thousands lines of code and any student who hasn't faced ten thousand lines of code of non-modular program doesn't understand why object-orientation is necessary" (I24).

For this participant, the issue is focussed more on the process of arriving at a good design, which he talked about in terms of "program simplicity". Good design is expressed in code through the elimination of duplication.

> "So you have a really dumb design with one or two objects in it probably the first few weeks and then as you go forward, you need to improve that design so that it's sustainable. … to particularly focus on the two rules of simplicity that Kent Beck puts as his definition of program simplicity. The

> two that matter to this conversation. The one being the removal of duplication and the other being expressed in your design ideas. When a design is what we would agree bad, what happens is you tend to have to write the same kind of code over and over again. … Well, that's the code's way of saying, hey, we have a concept here that needs to be expressed in code and so the simplicity rules says if you write the same code two or three times" (I25).

The participant did not focus purely on elimination of duplication. Good design has some characteristics that should be visible in the code. The techniques and practices that arrived at that design may not be visible but the characteristics of a good design should be visible.

> "I would ask myself, is this program well designed in the object-oriented sense? That would be reflected by whether the classes that were there seem complete, cohesive in the technical terms, whether the connections between things were good, whether there seemed to be a lot of duplicated code or near duplicated code that suggested that there were abstractions missing. I tend to be very programmatically focussed on is this program good, is it useful, is it easily maintainable and I'm not either very interested in nor am I very optimistic that you can look at a program and ascertain how it was built. You can look at it and say this is a credible design. It is a design that I could be proud of" (I25).

Another participant placed a lot of emphasis on the connection between one responsibility and one reason for change.

> "At the back of my mind I am always thinking what is the responsibilities of this class. There should basically be one responsibility and one reason to perhaps change that class" (I27).

The participant saw this as guiding the decision to split classes into separate classes.

> "I will think like that in terms of a class where I might say well I think that is doing too much, lets pull some stuff out. I think that class wants to be two

> classes because we have got two clear lines of responsibility, here you are trying to do this and this" (I27).

A participant with a focus on assessment talked of assessing based on the ease of maintenance. This is closely linked with flexibility, modularity, and the localisation of change. However, these characteristics are not the exclusive domain of object-oriented programs so the participant was prepared to accept solutions that are maintainable but may not be considered as object-oriented.

> "I would probably look at how understandable their design is from my perspective so coming in as the marker, look at their program from the perspective of the maintainer who is having to come in after the programs been delivered to some clients and ask the question 'how do I evolve it or maintain it, to either fix this bug or to add this extra piece of functionality'. At that point I would be looking at the structure of the system to see the degree to which it supports say flexibility, modularity, and localisation of change and those kinds of things. So if a student were to come up with a particularly nice design which may not be the classical OO way of doing it, for some definition of classical OO way of doing something, and it was a nice design that you conveyed easily and that lends itself to the maintenance task that I decided as the marker I want to go and do, then I should give them high marks" (I28).

Other characteristics that may lead to ease of maintenance are cohesion and coupling. These were reflected in the way that the program is divided up. However, there was also a warning that this dividing into classes can be taken too far so there is a need to balance things out.

> "A kind of system that obeys the principles of cohesion and coupling may well lend itself to that maintenance far easier than a system that didn't, so I would certainly look at how the system is divided and whether or not the collection of operations and attributes makes sense and the relationship between the classes makes sense and has it been OO'd to death; for example, is there too great a subdivision in classes, and has the student made the right kind of decisions in shifting from the analysis model where

> they're modelling objects in the real world to the design model where they're making informed decisions about which objects to merge, which objects to split, based on what's more pragmatic in the environment they've been developing for" (I28).

This participant also saw that the use of paradigm specific constructs should not be assumed as leading to an object-oriented design. An object-oriented design has attributes that are not necessarily present simply because the constructs are used.

> "They are using OO languages, they are using OO features, they're using polymorphism, they're using inheritance, they're using methods, they're using messages but it's not OO design" (I28).

The link between design qualities and the program as a model cannot be ignored. As was seen earlier, some felt the model did not need to correspond directly. Others still wanted a strong emphasis on being a real world model as represented by:

> "I read somewhere that one of the origins of explanation of what an object-oriented programming is, is that we needed to come up with a way that allows our program structure to be relevant to the real world so that when we talk to a manager and we say we need to change, if our program is very close to the real world and the change to the program is easy, it will be easy to change in the real world and if the change is hard in the real world, it will be hard in the program" (I1).

Here the concern was that if the model is too different from the real world, then the ability to talk about maintenance issues in a way that was realistic to managers or customers would be lost. However, other participants saw this modelling more in terms of choices and being able to decide what to model from the real world and what additional abstractions would be useful. Here the participant referenced earlier refers again to a real world concept, but emphasised that it may not necessarily be the appropriate solution to model that concept, and that there was a point where it becomes appropriate because of the complexity and requirements of the system.

> "This time example […], there is a point at which we get code associated with time zone mixed in in all the other methods, most of the other methods

> in the time class. When we start seeing that, that's probably time to simplify everything by extracting all of that out and putting it in a separate class. But how you decide when you cross that threshold, that's a difficult thing to do" (I9).

Good design involves recognising when it is appropriate to follow a particular design strategy, so in this participant's comments there was a leaning toward a thought process but he held back, seeking to place that thinking within the context of design qualities.

Another was more explicit about the use of useful abstractions when creating a good design. He contended that in writing the program, the programmer is creating a reality.

> "This thing with reality is kind of a funny one, cause you are creating a reality when you're writing a program. You're imposing a reality on the way people are going to interact with this. And I know a lot of the object-oriented people early on were talking, UML people still talk about this, of actually modelling directly your business objects because it was so simple. No it's more that you're coming up with something which is useful, that's a useful abstraction." (I15).

Although he accepted modelling of business objects, he saw it as more of a process of arriving at useful abstractions.

The participant (I28) quoted earlier, saw the analysis process as deriving a model that reflects the real world but when it comes to design and the code, there was a need to evaluate what is workable and effective. He did not feel constrained by the concepts in the problem domain but saw that they could be merged or split depending on the requirements of a good design. He argued that:

> "The way it seems to be described in the literature is that there's a subtly different set of objects you come up with when you describe the problem than there is when you're describing the solution." (I28).

Design objectives are both attributes of an object-oriented program and desirable goals achieved by using object-oriented and generic concepts. Along with this shift toward design objectives, there is a shift in terms of the nature of the model produced from

being a model of reality to a model using useful abstractions. Although it is recognised that the model will reflect the domain of the problem, there are characteristics or attributes of that model that have no relationship to reality or to the problem domain. These abstractions are necessary in order to achieve the required design qualities.

D5) Expression of thought process

This category involves a major shift from the use of paradigm specific or generic constructs to an emphasis on the thought processes and the ways of thinking that are behind the programming paradigm or the software development process. Those participants who expressed this level of understanding tended to show reluctance to discuss paradigm specific terminology or to use some of the generic concepts. Their thought processes focused on design objectives, but in a way that sees them as the inevitable outcome.

This was expressed, in one case, as the paradigm is a "powerful way to think" about solutions to programming problems (I14) and in another as

> "Programming is the nearest you ever get to manipulating pure thought because the dimension of which programming is substantial is not in the physical, it is in the intellectual. It's in the metaphysical. It's a thinking process and building software systems is all about managing your thinking" (I23).

A third participant focussed on a way of working that reflected a "way of thinking to the definition of simple code, no duplication, expressing the whole ideas" (I25). This reflected a way of "working with the material" that said:

> "Ultimately, what matters in many things is what you do. It's all very well to think. I enjoy thinking. I enjoy sitting around in a coffee shop and just arguing how I might do things and all that but ultimately what you do is your guardian point" (I25).

Like experts, these participants recognised patterns of poor or good design but they were not constrained by the rules of the constructs; either expressed generically or for a

specific paradigm. Their definition of an object-oriented program related to the thought patterns it expressed or the thought patterns that brought it into being.

Specific characteristics related to this category are:

1) Relationship to expertise;
2) Reluctance to use and define terminology;
3) Focus on the ways of thinking about solutions;
4) Focus on solutions to problems;
5) Focus on strategies for resolving specific programming design problems (i.e. "ways of solving code duplication" (I25).

The participants in this category were more interested in communicating the thought patterns that led to good design. They saw object-oriented techniques as delivering good designs, but were more flexible about the implications of good design and the need to be able to recite the terminology. The emphasis was on understanding the implications and applications of the techniques rather than knowing what their names are.

A participant who focused on process issues was keen to see more research done that would show whether what he saw as a simple process worked effectively. He talked about writing code based on

> "never putting in a design an element that my tests don't call for or that the hideousness of the code doesn't call for" (I25)

and which offered the challenge to pursue research that focuses on removal of duplication and the expression of every design thought in code. He said:

> "An interesting topic, that I would like to see addressed a little bit in research, primarily because of my experience with it, is if you really focussed on the two notions in your code of removing duplication and expressing every design thought that you had in code, if you say to yourself as you go along here, that's interesting, in this object there's really two things going on" (I25).

Could this be taught to novice programmers? He went on to say:

> "It seems to me that you can take relatively junior people and say 'Look observe duplication. See these lines of code are exactly alike except this guy called it ABC and that guy called it IJK. That's duplication, we must get rid of that duplication. Now how shall we do that?' And then in some ways it just becomes obvious. We need a method that has those two things in it" (I25).

The focus of this discussion was on the implications of focusing on a practice of recognising the patterns of duplication and the techniques for the removal of duplication. If this is done rather than focusing on the terminology and techniques, would we end up with better programmers?

Some saw difficulty with this idea since it seems difficult to articulate exactly what thought processes go behind building a good design.

> "The problem with reading the books is, you look at their examples and it's really obvious and then when you try and apply it yourself, you find I don't know actually how they came up with that. So I don't think they've articulated particularly well. Furthermore, I think it's the kind of thing that it's really hard to articulate and they may have difficulty themselves. They just see the right answer and can't understand why other people didn't see it straight away. But realise that other people don't so they write the book and one reads the book and thinks "yes, that's clearly the right way to do it" but still don't know how you came up with it in the first place. And that's where I see the utility in being able to have a more objective way of characterising the quality of the design" (I9).

This category offers up a number of challenges for research. As the last quote states, there seems to be difficulty in articulating the thought processes that object-oriented experts use and then in passing these on to others. There is also the challenge to validate that the teaching of a process of recognising patterns of duplication and then removing them will bring quality in software design. There is also a need to explore the approaches to teaching these processes to novices.

The emphasis with respect to modelling is more strongly linked with useful abstractions. This participant in some respect reflected the idea of modelling the real world, but recognised the power of being able to think abstractly about what can be modelled.

> "I think of objects as like actors, like people so when I teach I talk about objects being things but I also am careful to point out, because otherwise students will get the wrong idea that things don't have to be concrete physical things. Goodness, truth and beauty are things that we can model in an object system." (I14).

Another participant expressed his concern with seeing an object-oriented program as a model of the real world and rather likes to see it as a simulation. However, he saw that modelling the real world also doesn't go far enough.

> "I got exposed to the notion that the way you do object-oriented programming is by having objects that match things in the real world. I think in trying to simulate the real world that that's not what you want. I think what you should actually think about how software's really designed. It does stuff that's useful in the real world but it does it in ways that makes sense to computers because it's computer programming and the real world doesn't have the database and all the database related stuff. I think that the real world is about employees and cars and whatever. I think that there's some place for that focus on real world objects but as an approach it doesn't go very far." (I25).

These thought processes are based on design objectives, but not necessarily on rigid patterns of implementation. The thought process experts recognise even obscure cases of duplication and poor design. They also recognise the solutions for improving that design. Yet, they are flexible in terms of the solutions arrived at and evaluate each in terms of design objectives and effective solutions (Jeffries, 2004).

5.1.4 Critical aspects for design characteristics

Unlike the nature of an object-oriented program outcome space where the aspects became visible during the process of uncovering the categories, the critical aspects for

the design characteristics outcome space were identified as a second stage in the analysis. The critical aspects were identified by carefully examining the characteristics of each category and its distinguishing features. Each of the identified critical aspects focuses on a particular theme that is evident in the design characteristics outcome space (see Table 5.4).

Cat	Referential	Critical Aspects			
		Technology	**Principles**	**Cognitive process**	**Modelling**
D1	Is language dependent	Constrained by language features	Language features determine principles applied	Need to know the language	
D2	Uses paradigm constructs	Constrained by paradigm constructs	Paradigm constructs determine principles applied	Thinking in terms of objects	Reflects real world objects
D3	Uses generic principles	Implementation tool but paradigm specific	Paradigm independent principles drive design decisions	Thinking in terms of good coding practices	Reflects real world objects
D4	Applies generic design objectives	Implementation tool but some flexibility on paradigm	Design objectives drive design decisions	Thinking in terms of the impact of design decisions	Uses useful abstractions
D5	Expression of thought process	Implementation tool but select paradigm to match	Thought patterns implement design principles	Implicit recognition and correction of poor design	Communicate program design

Table 5.4: Design characteristics critical aspects

The technology aspect is the degree by which the implementation is constrained by the technology. In category “D1: Is language dependant”, the focus is on the language as the constraint to design. The features of the language dictate what is possible and how the program is structured. For category “D2: Uses paradigm constructs”, there is a shift away from the specific language features to paradigm constructs. At the higher levels, both the language features and the paradigm constructs become primarily implementation tools. For the highest category “D5: Expression of thought process”, the

paradigm constructs are seen more as patterns of thought than as technological solutions. For the higher three categories, the technology is the slave of design or the thought processes and not the focus of design. With the shift to the thought process, there is a willingness to mix and match paradigm and technology tools in order to satisfy the design and thought process requirements.

The design principles aspect sees a shift from focusing on the technology restricting the design to a focus on generic principles of design. Design for the “D1: Is language dependant” category is about using language features to modularise or structure the code. The design focus shifts to what the paradigm constructs allow for category “D2: Uses paradigm constructs”. With these first two categories, the technologies drive the design decisions. Although this may allow some code reuse and design for ease of maintenance, these more generic principles or design objectives are seen more as a result of using the technology than as objectives that determine the approach taken in using the technology. The generic principles begin to drive design choices for category “D3: Uses generic principles”. This may see the use of implementation patterns and principles of code reuse dominating the design. Here the implementation patterns may be constrained by the paradigm or the language, but the focus is moving away from paradigm specific concepts and toward principles that apply across paradigms. The implementation patterns are ways of implementing the generic principles for the given paradigm and programming language. For category “D4: Applies generic design objectives”, the focus is on the design objectives. The approach to design is driven by these objectives rather than by any language or paradigm constraints. There is a possibility at this level that the programmer will mix and match languages and paradigms to fit the required design objectives. For the final category “D5: Expression of thought process”, the shift is toward patterns of thought that led to good design. The design principles do not dominate the thinking but they are consistently applied through having become tacit principles for the design of good code.

The third aspect is called cognitive process as the focus is on the thought processes behind the approach to creating an object-oriented program. The distinction between the categories is related to ‘what’ the thought process is focused on, rather than a distinction in terms of the cognitive skills being utilised. At the lowest category, “D1: Is language dependant”, the focus is on knowing the language in order to be able to write programs.

Knowledge of the language is more than simply being able to identify, recall, and apply language constructs; the programmer needs to be aware of the appropriate use of constructs. The cognitive process moves to thinking in terms of object-oriented principles for the second category, “D2: Uses paradigm constructs”. Here the focus is on the use of objects as required by the paradigm. Like the first category, this involves identification, recall, and application of paradigm constructs. With “D3: Uses generic principles”, the focus moves to good coding practices. The technologies are simply tools for the production of good code that satisfy the principles. At this point, the programmer begins to break away from paradigm dependency and have a flexible approach to the use of a range of paradigms. The transition to “D4: Applies generic design objectives” takes the focus to an emphasis on the impact of design decisions. The influence of the paradigm is more from the perspective of how it helps generate a good design. The programmer may mix and match programming paradigms if that delivers a better design. The final category, “D5: Expression of thought process” sees a shift to the point where there is an implicit recognition and correction of poor design. In the preceding two categories, the programmer may be guided by design patterns but with this top category, the programmer works to eliminate code problems through recognition of what are called ‘code smells’ (Fowler, 1999) and applies strategies to eliminate them. The paradigm is irrelevant other than in how it can be used to support the thought processes.

The fourth aspect is a focus on modelling or the production of a model. The lowest category, “D1: Is language dependant” makes no reference to the program as a model or programming as modelling. At the next two categories, “D2: Uses paradigm constructs” and “D3: Uses generic principles”, modelling is emphasised in terms of the program being a model that reflects real world objects. The programmer’s task is that of modelling the real world using object-technology and / or the application of generic programming principles. For the fourth category, “D4: Applies generic design objectives”, while there is still a real world focus in providing a starting point for design or hints of possible objects to include in the design, there is a recognition that not all real world objects will be modelled and that other objects, which are seen as useful abstractions, will be included. The created model has to be seen as delivering a useful solution that satisfies the required design characteristics. For the final category, “D5:

Expression of thought process", the emphasis is still on useful abstraction but with an emphasis on how those abstractions communicate the program design.

5.2 Conclusion

Two outcome spaces, which relate to ways that practitioners express their awareness of an object-oriented program and the design characteristics used in implementation, have been described in this chapter. These have been further analysed to identify the critical aspects that distinguish variations in awareness expressed by the practitioners.

The first outcome space focuses on structural descriptions and is more technical in focus. This is a focus on the 'what' aspect of programming in the sense of what is produced. It describes the characteristics of the program from the perspective of how the program is executed (flow of control critical aspect), the way that the paradigm building blocks (objects) are used (use of objects critical aspect), and the focus on the program as a model derived from either data or behaviour (problem solution critical aspect). In this outcome space, there is a shift from following a set of instructions through interacting entities to the idea that the concept of a program is an artificial construct created for the requirements of operating systems. This latter description frees the developer from the restrictions of program boundary to explore the use of reusable functionality as the units developed in programming. Each of these categories builds on the lower categories, but there is a clear shift in emphasis in the way software is understood from a structural perspective.

The second outcome space focuses on design characteristics and begins to consider the issues of what makes a good object-oriented program or how a program is written. At the lower levels, design quality is less of an issue and the focus is on the technologies (i.e. the language constructs or paradigm specific constructs). The shift to generic constructs suggests that some constructs are applicable regardless of the programming paradigm. The shift to design qualities further moves toward generic characteristics or properties of a program. Finally, the expression of a thought process offers a challenge to further explore the way that expert programmers think, and to explore how novice programmers can learn these thinking processes. These thought processes do not neglect an understanding of design qualities or the paradigm or generic constructs, rather these thought processes leverage the paradigm or generic constructs to produce good designs

that are not constrained by paradigm specific constraints. These thought processes suggest that there may be ‘programming in the small’ patterns that, if used consistently, deliver quality program designs.

There is a difference in the way that the practitioners focus on the critical aspects in the two outcome spaces. For the nature of a program, there are different variations in the expressions of awareness of the critical aspects which provided the differentiation between the categories. With the design characteristics outcome space, there is more of a change in focus with respect to the critical aspects. At the lower categories of the outcome space the focus is on the technology critical aspect. It then moves to the design principles critical aspect through the middle categories and with the highest category, the cognitive process critical aspect is the primary focus. The higher level categories do not ignore the technology critical aspect but see it as taking a subordinate role. The modelling critical aspect reflects how this shift in focus on the other critical aspects influences the approach to producing a program and the nature of the resulting program.

In the following chapter, the critical aspects underlying the outcome spaces will be used as a guide to the analysis of a number of textbooks that introduce object-oriented programming. The objective is to determine whether textbooks utilise similar aspects to those discussed in this chapter and what is the space of learning in these texts.

Chapter 6. Variations in textbooks

Having identified the variations in practitioners' ways of understanding object-oriented programming in the previous chapter, this chapter explores whether these understandings are reflected in textbooks used to teach object-oriented programming. A textbook opens up a space of learning with respect to the subject matter. The space of learning is defined by the variations in the critical aspects presented in the learning material. The focus of this chapter is on the third research question which is:

> What spaces of learning are opened up by introductory programming texts with respect to the 'critical aspects' arrived at from the analysis of the practitioner interviews?

The analysis carried out in this chapter will examine each of the critical aspects and discuss the way the textbooks have dealt with each of the aspects. This will be done by answering the following questions that have been formulated based on the results of the practitioner interview analysis.

3. How do the texts address the nature of an object-oriented program?
    a. How do the texts address flow of control?
    b. How do the texts address the usage of objects?
    c. How do the texts address the nature of the problem solution?
4. How do the texts address the design characteristics of an object-oriented program?
    a. How do the texts address the influence of technology?
    b. How do the texts address the program design principles?
    c. How do the texts address the cognitive processes?
    d. How do the texts address the issue of modelling?

In order to perform this analysis, the second form of variation which defines the space of learning within a learning context has been used with respect to the textbooks. This form of variation endeavours to uncover the variations used to present concepts and to determine the space of learning that is constructed. There was a difficulty in applying this to the text books as many of the authors introduce a concept or an idea and then illustrate it with a single example and limited surrounding discussion. In order to

determine the nature of a program portrayed by a text, it was necessary to look at sample code across a number of examples for different concepts to see the common patterns or variations.

The results of this analysis will provide insight into the emphasis placed on the critical aspects by the selected textbooks. A list of the textbooks used is provided in appendix 5.

6.1 How do textbooks address the nature of an object-oriented program?

In terms of Ramsden's (2003) relational learning theory, how the learner understands the nature of a program is a possible influence on the task representation that the learner has in relation to programming. From the analysis of the practitioner data, three critical aspects were uncovered that related to the nature of a program. These critical aspects are:

1. flow of control,
2. usage of objects, and
3. the nature of the problem solution.

In this section the textbooks have been examined from the perspective of these three critical aspects.

6.1.1 How do textbooks address flow of control?

For the flow of control critical aspect, two variations have been identified from the practitioner interviews that relate to the practitioner's awareness of an object-oriented program. These are that

1. the flow of control is seen as being sequential, and
2. the flow of control being represented in the interactions between objects.

Both variations of flow of control are emphasised in the textbooks.

The sequential flow of control variation is consistently emphasised by Anderson and Franceschi (2005). Emphasis on the sequential nature is declared in the body of the text

and in the bulk of their examples. Some examples are more dependent on interactions but this is not emphasised. Their introduction of the sequential nature of a program occurs in the introductory chapters where the authors discuss a program as consisting "of steps, called **instructions**, which are performed in order" (p. 22). In the discussion around this statement, examples are given which emphasise the sequential nature of a program. The programmer should expect to have to determine the sequence of the steps that will be executed by the computer and create their program so that it clearly lays out the steps to follow. These authors emphasise this when they say:

> "Programming, therefore, requires the programmer to specify the order of instructions, which is called the **flow of control** of the program. There are four different ways that the flow of control can progress through a program: sequential execution, method call, selection, and looping" (p 22).

The discussion then moves on to pseudocode but there is still a strong emphasis on the sequential flow of control. This is emphasised when the authors say "The order of operations is still input, calculate, and output" (p 24). In the examples which follow, they then introduce selections (conditional execution) and loops maintaining the sequential flow of control theme. This leads to defining an algorithm as a "standard method of processing" for a given task (p 25).

Another way in which sequential flow of control is emphasised is through defining the nature of an algorithm (Becker, 2007; Brookshear, 2007; Lewis & Loftus, 2006; Malik, 2006; Soroka, 2006). The common definition in all of these texts is that an algorithm is "an ordered set of unambiguous, executable steps that defines a terminating process" (Brookshear, 2007, p 213) or "A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time" (Malik, 2006, p 15). This is linked with the nature of a program through contending that a program is a "representation of an algorithm" (Brookshear, 2007, p 18). Guzdial and Ericson (2005) make a distinction by linking the concept of a program with the implementation in a programming language (p 6).

Brookshear (2007) does recognise that other programming paradigms are described in terms other than a set of steps or instructions (pp 272-320), but states the algorithm is "the most fundamental concept of computer science" (p 18). If an algorithm is an

ordered set of steps, then logically that is the way that it would be implemented. Malik (2006) also makes this distinction when contrasting structured programming with object-oriented programming by saying that

> "In OOD, the first step in problem-solving process is to identify the components called **objects**, which form the basis of the solution, and determine how these objects interact with one another" and "the final program is a collection of interacting objects" (p 21).

This leads to some confusion with respect to flow of control in an object-oriented program. This confusion around flow of control is also represented by seeing object-oriented programming as "an extension of procedural programming in which you take a slightly different approach to writing computer programs" (Farrell, 2006a, p 4). In statements that precede this quote, Farrell has clearly indicated that procedural programming is based on a set of operations that are executed one after another in sequence (p 3).

But Farrell also implies that an object-oriented program is about the interaction between objects. She says:

> "Thinking in an object-oriented manner envisioning program components as objects that belong to classes and are similar to concrete objects in the real world; then you can manipulate the objects and have them interrelate with each other to achieve a desired result" (p 4)

This isn't an outright declaration of interaction as the flow of control but it does infer interaction between the objects.

This same confusion is present in Morelli and Walde (2006) where they consistently emphasise interactions between objects (pp xv, xvi, 10, 17-18, 24-25) but then confuse the issue when discussing the design of objects, when they say:

> "Object orientation subsumes both procedural abstraction and structured programming concepts from Pascal days" (p xv).

A further way in which confusion with respect to the flow of control is achieved is through the use of "`public static void main(String [] args)`" method

(this will be referred to as the "`main`" method in the following discussion). The first code example is often a short piece of code built around the "`main`" method (e.g. Farrell, 2006a, p 8). Like other texts, Farrell uses code sequences within the "`main`" method to introduce the data concepts and expressions leading to some lengthy single method examples (chapter 2). Methods are introduced by using initially static methods and focusing on parameter passing (pp 84-95) and then return values (pp 95-96). The examples still rely heavily on the use of the "`main`" method even after introducing classes (pp 96-103). Even in chapter 4 (pp 135-147), there is a strong dependence on static methods and the "`main`" method to introduce block and scope concepts. This usage continues into the teaching of logic constructs (chapter 5 and 6) and further through the text. This pattern is also evident in the generic object-oriented text (Farrell, 2006b) although disguised by the limited use of pseudocode examples.

Many of the other textbooks, particularly in early examples, use sequential code in the "`main`" method. In some cases, this was done purely to illustrate the use of predefined objects, with some of the authors emphasising the interactions and message passing (Becker, 2007; Bergin et al., 2005; Guzdial & Ericson, 2005). For others, the use of the "`main`" method is argued as necessary in order to lay a base foundation for the later learning. The same argument was used by some authors for introducing logic constructs and loops early. Malik (2006) argues that this knowledge is necessary before being able to introduce object usage or interactions. The question from the perspective of learning is how the emphasis on procedural logic influences the learner's perception of the nature of a program before the emphasis shifts to interactions.

Becker (2007) and Bergin et al. (2005) initially use a sequence of message calls from the "`main`" method to drive the initial program examples. This is less obvious in Bergin et al. (2005) by the time that they introduce polymorphism. By introducing polymorphism ahead of the logic constructs of conditionals and loops, Bergin et al. are able to talk of the use of the "`if`" statement as ad-hoc polymorphism and to demonstrate how polymorphism can be used in place of conditionals.

Lewis and Loftus (2006) introduce their first Java program as code based on the "`main`" method. This usage of the "`main`" method continues through the introduction of data types, expressions, and the initial code for handling input from the standard

input device (console). With the introduction of applets, the use of the "`main`" method is replaced by the use of the "`paint`" method.

Most of the textbooks in the research sample emphasise interacting objects. These include Barker (2005), Barnes and Kölling (2006), Becker (2007), Bergin (2005), Brookshear (2007), Guzdial and Ericson (2005), Jeffries (2004), Langr (2005), Lewis and Loftus (2006), Morelli and Walde (2006), Sierra and Bates(2005), Soroka (2006), Stein (2003), Wiener (2007). All of these make clear statements that an object-oriented program is based on interacting entities but the approaches to teaching and showing this differ markedly.

Barker (2005) places emphasis on an object-oriented program as a collection of interacting entities is based on the entities responding to external triggers. This is particularly emphasised through the statement:

> "As soon as such a triggering event has been noted by an OO system, the appropriate objects react, performing services themselves and/or requesting services of other objects in chain-reaction fashion, until some overall goal of the application has been accomplished" (p 96).

This leads into a discussion on methods to implement object behaviour. The focus here is on identifying the required interactions and the data that needs to be passed to or returned from a service.

Another example of the emphasis on interaction is shown when she introduces delegation (p 116). Here she says:

> "If a request is made of an object A and, in fulfilling the request, A in turn requests assistance from another object B, this is known as **delegation** by A to B" (p 118).

The language here is not that of calling a method but rather that of communication and interaction. She goes on to equate this with "delegation between people in the real world" (p 118).

Barnes and Kölling (2006) emphasis interaction through arguing for the use of BlueJ in teaching because with BlueJ, a

> "student can create an object and call its methods as the very first activity! Because users can create and interact with objects directly, concepts such as classes, objects, methods, and parameters can easily be discussed in a concrete manner before looking at the first line of Java syntax" (p xx).

They further emphasise this view, when they contend that an "interesting application" can only be constructed by combining objects "so that they cooperate to perform a common task" (p 52). This is further reinforced when they discuss the design of an application. They say

> "When we think about what our program will do, we will think about the object structures it creates, and how these objects interact. Being able to visualize the object structures is essential" (p 58).

They further reinforce the interaction focus when they introduce object-oriented concepts such as interfaces where they emphasise the collaborative nature (p 175). This repeated reinforcement occurs when they discuss coupling (pp 193, 203), the design process (p 394), and design patterns (p 405). The initial interactions are with single objects using the BlueJ object workbench but they move to collaborating objects by chapter three of the text.

Becker (2007) not only emphasises the interactions between objects, he also talks about the interactions between methods. He says:

> "If the program is written in an **object-oriented programming language**, such as Java, the computer program uses cooperating **software objects**" (p 4) and "these objects cooperate to model the problem" (p 13).

This is further reinforced when he introduces how to get robots to perform things through the use of services. Here he emphasises that a message is sent to request that the object performs a service (p 14). The first program examples (pseudocode (p 15) and Java code (p 17-18)) appear sequential in nature but are composed of a series of messages. Creating objects and sending them messages to do things is the primary focus

of the initial examples. This is also picked up in the initial introduction to GUI programming (pp 34-39).

The addition of a service to a class is introduced in chapter 2, with a reminder that "Services are invoked with a message" (p 62). The use of multiple objects is also introduced with interaction between a robot and a lamp (pp 67-78). The use of multiple objects reinforces the idea of sending messages to objects but the use of multiple robots is left until chapter 3 (pp 140-141). A diagram that helped the reader see that multiple objects are being used would have enhanced the message of object interactions but it is not present.

In their preface, Bergin et al. (2005) declare that "In object-oriented programming, a computation is carried out by a set of interacting objects" (p iii) but this is never really enforced other than references to message passing (pp 20, 78, 83) or client server interaction (p 74). Given the robot nature of the examples and the early introduction of polymorphism (chapter 4, p 65), understanding the message passing nature of objects is crucial to being able to work with this text. In introducing polymorphism, they say that "Objects in different classes can behave differently when sent the same messages" (p 65). References to the nature of a good program also show that these authors understand the interacting objects nature of a program when they say "A good program is the simple composition of easily understandable parts" (p 58).

Brookshear (2007) also defines an object-oriented program in terms of being a set of interacting objects. In relation to object-oriented paradigm, Brookshear says

> "Following this paradigm, a software system is viewed as a collection of units, called **objects**, each of which is capable of performing the actions that are immediately related to itself as well as requesting actions of other objects. Together, these objects interact to solve the problem at hand" (p 275)

This is emphasised by the idea that a program unit, an object, would contain the algorithms for maintenance of the data that it contained (p 276) and the idea that objects contain "procedures describing how that object should respond to various stimuli" (p 305). Brookshear constantly contrasts the object-oriented paradigm with the procedural

paradigm to highlight that the paradigm is based on a different assumption about programming. However, he does acknowledge that methods are “small imperative program units” (pp 276, 307).

Guzdial and Ericson (2005), like Stein (2003), use the description of the operation of a restaurant to illustrate this interaction between objects but, like Bergin et al. (2005), their initial examples are about using messages to control and manipulate a pre-existing object (pp 43-50). They strongly promote the idea that “In object-oriented programming we ask an object to do something by sending it a message” (p 48). They also utilise a programming environment, Dr Java, that allows for immediate experimentation by typing in Java statements for immediate execution.

There is a clear emphasis, as in the books that utilise robots (Becker, 2007; Bergin et al., 2005), on using messages to manipulate objects, in this case media objects, but unlike Becker and Bergin et al., Guzdial and Ericson (2005) do not go into all the object-oriented concepts. Although the student is introduced to classes, this is more in the context of basic usage than understanding the possibilities of their usage.

Jeffries (2004) relies on the code to illustrate interacting entities. His approach to programming is based on implementing the functionality within the parameters of the model view controller (MVC) approach to code development. He utilises test-driven development (TDD) to drive the identification and implementation of the application objects. If the problem calls for a procedural solution, then that will be implemented. There is no pressure to produce a particular style of solution simply because the language is object-oriented. In writing his code, Jeffries does not utilise the low level elementary patterns explicitly but they are built into his thinking process. This enables him to see when it is appropriate to utilise interactions to implement the required algorithm and when it is best to utilise procedural logic.

Langr’s (2005) emphasis on an object-oriented program being a set of interacting objects (pp 12-14) is frequently reinforced throughout the text (pp 32, 53, 65, 66). There is, in Langr’s mind, some connection between interacting entities and the ability to use test-driven development as shown when he says:

> "The client that sends a message to an object using a variable of CourseSession type expects it to be interpreted and acted upon in a certain manner. In test-driven development, unit tests define this "manner." A unit test describes the behaviour supported by interacting with a class through its interface. It describes the behaviour by first actually effecting that behaviour, then by ensuring that any number of *postconditions* hold true upon completion" (pp 220-221).

The emphasis on test-driven development also means that Langr avoids the need to introduce the "`main`" method and that all his examples are based on the use of classes that are activated by messages from a test object or other program objects.

Sierra and Bates (2005) have a strong focus on learning with a lot of key issues being introduced through conversational style episodes such as "Chair wars" (pp 28-32, 166-167). These stories quickly introduce ideas that challenge the reader about the differences between procedural programming and object-oriented programming. Through these stories, the reader experiences the ideas of interacting entities. But Sierra and Bates are also explicit at times and say:

> "As long as you're in main() [sic], you're not really in Objectville. It's fine for a test program to run within the main method, but in a true OO application, you need objects talking to other objects" (p 38)

and

> "A real Java application is nothing but objects talking to other objects. In this case, *talking* means objects calling methods on one another" (p 38)

Once they get passed the initial introductory chapter and into "Objectville", their examples reinforce the idea that an object-oriented program is made up of interacting objects.

From the modelling emphasis in Wiener (2007), the nature of a program would seem to involve object interactions. The models introduced even in chapter one involve multiple classes and the code both defines and uses objects although still primarily driven from the "`main`" method. There is little emphasis placed on this method and it seems to be

used simply as a requirement for doing anything. Another indication of the possible emphasis on object interaction is that he moves quickly to discussing interaction diagrams and sequence diagrams within his models (pp 69).

Stein (2003) makes a clear effort to contrast the two styles of programming when she examines the procedural flow in making a peanut butter sandwich with the interactions in the operation of a restaurant. The core emphasis of the text is that a program as an "interactive computational community" (chapter 1). She discusses the entities that make up this community as being in a constant loop waiting for the next request. When talking about individual instruction followers, she does talk about a sequence of instructions but these are constructed using methods and smaller chunks of code (Chapter 1). When she looks at the development process, her emphasis is on the overall "behaviour of the program" and the identification of "members of the community" (Chapter 2). This is followed through by considering how the community members interact and what each member does. Stein introduces these ideas from real world situations highlighting both the instruction following nature of a program and the interactive community of instruction followers. This emphasis on interaction is also reflected in the choice of topics covered in the text. Stein introduces multi-tasking and event-driven programming as techniques that depend on interacting entities.

The bulk of the textbooks examined in this research place some degree of emphasis on interacting objects as the control flow for an object-oriented program. These include Barker (2005), Barnes and Kölling (2006), Becker (2007), Bergin et al. (2005), Brookshear (2007), Guzdial and Ericson (2005), Jeffries (2004), Langr (2005), Lewis and Loftus (2006), Morelli andWalde (2006), Sierra and Bates (2005), Soroka (2006), Stein (2003), Wiener (2007). Only Anderson and Franceschi (2005) focus more on a sequence of instructions. Bronson (2006), Farrell (2006a; 2006b), and Malik (2006) introduce the idea of interacting entities but don't seem to follow it through in the text. Felleisen et al. (2001) make no specific references and it is difficult to identify the flow of control concepts promoted by the texts.

Stein (2003) makes a definite effort to illustrate the difference between the two variations of flow of control. Sierra and Bates' (2005) use of a dialogue between a procedural programmer and an object-oriented programmer also compares the two approaches and they use this to discuss the relative advantages and disadvantages.

The approach predominately used in the textbooks to emphasis the flow of control is one of telling and retelling the reader what the nature of an object-oriented program is. The overall impression from the analysis of these texts is that there appears to be some difficulty in presenting the interaction variation through example code. This is despite two of the texts endeavouring to contrast the two variations.

6.1.2 How do the texts address the usage of objects?

With respect to object usage critical aspect, five variations were identified from the practitioner interviews. These variations are that:

1. Objects are used from the provided framework;
2. Programmer objects are defined and used but primarily as a requirement of the language or framework;
3. Objects are defined based on the data requirements;
4. Objects are defined based on behavioural requirements; and
5. Objects are liberated from the confines of program boundaries.

All of the texts discuss and provide illustrations of the use of objects from the provided framework. In the case of Becker (2007) and Bergin et al. (2005), this includes a framework that implements robots.

The analysis with respect to the use of objects encountered a problem related to the use of concept of ‘type’ in object-oriented languages. The focus of the meaning of type is related to data in the sense that authors refer to abstract data types. For example, Malik talks of a class being “used to allow users to create their own data types” (p 76). In contrast, Stein (2003) states

> “A type tells you what kind of behaviour you can expect from an object” (Chap. 3) and “A type specifies what a Thing can do (or what you can do with a Thing). Types are like contracts that tell you what kinds of interactions you can have with Things. What an object's structure is -- and what that object can do, or what you can do with it -- is fully determined by its type, i.e., what kind of object it is.” (Chap. 3)

The emphasis for Stein is behavioural in focus. Type is about what an object can do. It isn't about the data that it holds although that might be an attribute of the behaviour. The use of type becomes unclear when the author doesn't declare the way that they use the terminology. For example, Bronson talks of object types but it isn't clear what makes something a type other than it is related to a class.

> "Although the terms *objects* and *object types* that we have been using make sense from an intuitive viewpoint, we need to refine our understanding of these terms for our programming. To make this clearer, and as an example, consider any automobile that you have recently driven. From an object viewpoint, a specific car is a particular instance, or object, of a more general class of car. … The plan for building a particular car is held by the respective manufacturers. Only when such a plan is put into action and a car is actually built does a specific object come into existence. … In programming terminology, a class is simply what we have been referring to as an object type. It is from the object type, or more accurately, from a class, that any one specific object is created" (p 14).

Object type seems to being used as some type of classification mechanism that allows the creation of classes or plans for construction. The nature of how that classification is performed other than by identifying things that have obvious differences is unclear.

Anderson and Franceschi (2005), Barker (2005), Brookshear (2007), Farrell (2006a; 2006b), Felleisen et al. (2001), and Malik (2006) lean toward objects being defined based on data requirements.

Brookshear (2007) emphasises "the similarities between classes and data types" (p 306) and then argues that "classes are actually descriptions of abstract data types" (p 395). At the end of the discussion, he concludes that "the concepts of classes and objects represent another step in the evolution of techniques for representing data abstractions in programs" (p 396).

In a similar way, Malik (2006) explicitly states that a class is a user defined data type (p 76). He emphasises that the data is what is used to categorise into types. However, he

recognises that in order to utilise the data, there needs to be operations that can be performed on it when he says:

> "**Data type:** A set of values together with a set of operations" (p 31)

Barker (2005) presents a data-driven approach by arguing that you should design the "application's **data structure** first, and its functions second" (p 67) and that in the modelling process, you should "Handle the data side of the application by identifying the different classes of "real-world" objects" ahead of the "functional side" (p 337).

Barker also relates objects to "physical things" (p 68) and recognises the link between data and behaviour when she says:

> "From a software perspective, a (software) **object** is a software construct/module that bundles together **state** (**data**) and **behaviour** (**functions**) which, taken together, represent an abstraction of a real-world (physical or conceptual) object" (p 68)

Barker refers to services as an alternative to behaviour (p 92) and uses this terminology when talking about information hiding and the way that objects expose services to other objects.

The emphasis in Felleisen et al. (2001) is on the development of classes based on data requirements and the building of class hierarchies. This emphasis starts early when they say:

> "students should also understand that a program may rely on an entire system of interconnected data definitions (family trees, files and folders)" (p 5)

The data emphasis is reinforced by the idea that "a programmer's first action is to describe the classes of *all* possible input and output data" (p 7) and that "For object-oriented languages such as C# and Java, the data language is the language of classes" (p 7). Object usage in Felleisen et al.'s understanding is based on data requirements. The rest of the program is about manipulating that data.

The data requirements are initially emphasised by Farrell (2006a) through an emphasis on data modelling (p 7). The data requirements drive the decisions on objects and their use but when introducing inheritance and interfaces, the focus switches to behaviour. She says "**inheritance** is a mechanism that enables one class to inherit, or assume, both the behaviour and the attributes of another class" (p 510) and "An interface is analogous to a protocol, which is an agreed-on behaviour" (p 537). It would appear to be a mixed message although from the perspective of a class, the emphasis is on the data requirements.

Anderson and Franceschi (2005) are less direct in their emphasis on objects based on data requirements when they refer to a class enabling "the programmer to encapsulate data and the functions needed to manipulate the data into one package" (p 20). They do however state:

> "Classes provide services to the program. … The program that uses a class is called the **client** of the class" (p 92)

If services equates to behaviour, then they are aware of the behavioural nature of classes but these perspectives are not followed through into the body of the text.

Bronson (2006) appears to emphasise both the data and behavioural requirements of objects. When Bronson compares a class with a plan and a recipe (pp 14-18), he recognises both the data and behavioural aspects of an object. The objects are identified first followed by "identifying the objects' attributes and behaviour" (p 116). However, in the discussion on simulating a coin toss (pp 115-116), Bronson considers possible attributes that could be used for describing the coin but concludes "in terms of modelling a coin for performing a coin toss, the only attributes that we need to consider initially is what side is visible after the coin is tossed" (p 116). The influence in deciding what data is required is based on the behavioural requirements of the system and not the range of possible attributes that could be used for describing a coin.

Most of the textbooks emphasise defining objects based on behavioural requirements. These include Barnes and Kölling (2006), Becker (2007), Bergin et al. (2005), Guzdial and Ericson (2005), Jeffries (2002), Langr (2005), Lewis and Loftus (2006), Morelli and

Walde (2006), Sierra and Bates (2005), Stein (2003), and Wiener (2007). None totally ignore the data requirements.

For the identification of objects, Barnes and Kölling (2006) focus on entities within the problem domain rather than either the data or the behaviour. In defining a type, the emphasis appears to be on the data when they say:

> "The type of a field specifies what kind of values can be stored in the field. If the type is a class, the field can hold objects of that class" (p 55).

The emphasis on behaviour becomes more visible when examining one of their code examples where they label the section "Exploring the behaviour of a naïve ticket machine" (p 18) but the examination includes examining the data as well as the methods. The behaviour focus is picked up when they talk about well-behaved objects (Chapter 6, pp 158-188) and when they introduce designing classes, they talk briefly about responsibility-driven design (pp 204-206). The responsibilities are linked with a class being responsible for maintaining its own data (p 204) although defined more generally as classes being assigned "well-defined responsibilities" (p 204). When they begin to talk about static and dynamic types, they place an emphasis on behaviour when by saying:

> "We call the type of the object stored in a variable the *dynamic type*, because it depends on assignments at runtime - the dynamic behaviour of the program" (p 261)

The behaviour focus becomes clearer when they use noun/verb analysis to identify classes and responsibilities when designing applications (Chapter 13, pp 393-401). This includes the use of the class/responsibilities/collaborators (CRC) card, a focus on scenarios, and a focus on the interface for the classes. This is the last chapter of the book. This text is clearly sending a mixed message but the emphasis in the design of classes is on behaviour or responsibilities.

Becker's (2007) focus on behaviour is introduced through the idea that a class provides services. Although he defines the meaning of a class in terms of the attributes and services that it provides (p 6), he places more emphasis on the need to know the services when he says:

> "The important part of a class is the services its objects provide - the things they can do. It is appropriate to say that the programmer implementing the class needs to know the attributes, but it's no one else's business how the object works internally" (p 13)

Through this argument, Becker is placing the emphasis on the services a class offers to other parts of the program or users rather than the attributes. He argues that access to the attributes is provided through services (p 4). This emphasis on services is also reflected in the way that he introduces coding where he looks at controlling a robot.

The behavioural focus in Bergin et al. (2005) is shown through an emphasis on robot capabilities (p 3). These capabilities include what the robot can do and what it can remember. This follows through to the idea that robots can be asked to do things and to recall things. This message is repeated through the text and is further emphasised by encouraging the view that robots be considered as servers providing services to its clients (p 74). When referring to delegation, it is a task that is delegated to another object (p 78) and polymorphism is talked about in terms of defining different sets of behaviour (pp 83-89). Although it may be inferred from the early references to remembering that there will be data in the objects, the emphasis is consistently on what an object can do and the distribution of behaviours to objects.

For Guzdial and Ericson (2005), the initial emphasis is on what objects can do and know. They introduce this by saying:

> "Each object of the same class will have the same skills or operations (things it can do) and data or variables (things it knows about)" (p 15)

This has been introduced in the context of a restaurant illustration where the emphasis has been placed on the tasks that need to be performed. It is then stated that:

> "Restaurants break tasks into different jobs so that they can be efficient and effective. Object-oriented programs also try to distribute the tasks to be done so that no one object does all the work" (p 16)

This theme of behaviour is also emphasised when describing how message passing is used to ask objects to do things (p 48). The emphasis through these sections is on the

behaviour of objects and this is also fostered by using framework media objects to manipulate photos, sounds, and videos.

The focus of test-driven development is on the features of the system (Jeffries, 2002; Langr, 2005). To ensure the integrity of the software and the ongoing ability to improve the design, tests are used. Jeffries describes this as:

> "XP teams deliver running, tested software, building features – "stories," as we call them" (p 2).

This emphasis on functionality flows through the text but in a vein of learning the way of thinking about software development. Constantly there is a focus on shipping the required functionality as early as possible. This is seen when Jeffries says:

> "The system should have in it just the functionality that has been delivered to the customer and the simplest well-designed code needed to support that functionality" (p 33).

Extraction of objects within the test-driven approach is targeted at improving the ability to test the functionality and to ensure a clean and communicative design.

Langr (2005) is more explicit with respect to tests defining the behaviour of the unit against which the test will run. He says:

> "A unit test describes the behaviour supported by interacting with a class through its interface. It describes the behaviour by first actually effecting that behaviour, then by ensuring that any number of *postconditions* hold true upon completion" (p 220).

Langr is also very specific about object-oriented systems being primarily about behaviour and that messages are sent in order to achieve behaviour (p 14). A message sent to an object tells it to do something (p 14). He reminds his readers later that:

> "object-oriented systems are about modelling behaviour. That behaviour is effected by sending messages to objects to get them to do something - *not* to ask them for data" (p 65)

But he recognises that there is also data involved when he says:

> "Objects are a combination of behaviour (implemented in Java in term of methods) and attributes (implemented in Java in terms of fields)" (p 133).

To drive home his emphasis, he argues that if a system is only about storing and retrieving data then it wouldn't be particularly useful and it wouldn't be object-oriented (p 65).

Lewis and Loftus (2006) declare that classes "represent objects with well-defined state and behaviour" (p 162). In this context, they are linking both the behaviour and the data together. However, they argue that:

> "Using classes and objects for the services they provide is a fundamental part of object-oriented software, and sets the stage for writing classes of our own" (p 113).

They remind their readers that "a class represents a group of objects with similar behaviour" (p 296). This behavioural emphasis is also represented when they define encapsulation in terms of the sets of services provided by the object (pp 168-169). This is recognised in their discussion on polymorphism where they say that

> "Each class knows best how it should handle a specific behaviour" (p 492)

A stronger emphasis on behaviour is seen when they talk about identifying classes and they say:

> "Part of the process of identifying the classes needed in a program is the process of assigning responsibilities to each class. Each class represents an object with certain behaviours that are defined by the methods of the class. An activity that the program must accomplish must be represented somewhere in the behaviours of the classes. That is, each class is responsible for carrying out certain activities, and those responsibilities must be assigned as part of designing a program" (pp 296- 297)

There is no emphasis on the data required. All of the emphasis is placed on the behaviours expected of the objects of that class.

Morelli and Walde (2006) place data requirements as secondary to behavioural requirements when they define the encapsulation principle as:

> "The goal here is to encapsulate within each object the expertise needed to carry out its role in the program. Each object is self-contained module with a clear responsibility and the tools (attributes and actions) necessary to carry out its role" (p 16).

The same emphasis is picked up in their discussion on class definition when they ask "What role will the object perform in the program?" and "What action will it take?" (p 73). The data requirement is recognised when they ask "What data or information will it need?" and "What information will it hide from other objects?" (p 73). The behaviour takes precedence when they refer to polymorphism which they see as meaning that calls to methods involve "different behaviours when invoked on different objects" (p 136).

In contrasting the procedural approach to solving a problem and the object-oriented approach, Sierra and Bates (2005) begin to emphasise issues of behaviour. They have their object-oriented programmer saying:

> "The rest of the program really doesn't know or care *how* the triangle does it. And when you need to add something new to the program, you just write a new class for the new object type, so the **new objects will have their own behaviour**" (pp 31-32)

The context is the use of polymorphism but it illustrates the emphasis on behaviour and on extending behaviour. In discussing objects they say that objects have state and behaviour but it is written in the language of an object knowing and doing. They contend that:

> "Remember: a class describes what an object knows and what an object does" (p 72)

They are clearly linking the relationship between data and behaviour to what an object knows and does. This behavioural focus is picked up in the discussion of inheritance when they argue that inheritance should be used when there is a need to share behaviour between multiple classes but this is tempered with an emphasis on the inheritance being

a “is-a” relationship that is a type inheritance (p 181). The nature of type in terms of being data or behaviour focused is unclear but with the strength of the behavioural focus on inheritance, it would appear that behaviour takes precedence in determining the type. Polymorphism is talked about in terms of types and in relation to the roles of objects (pp 183-215).

Wiener (2007) emphasises the modelling behaviour. He says:

> “We model behaviour of entities in the domain of software development using the concept of a **class**. These entities are the objects in the problem domain, and their behaviour is the actions that may be taken on these objects. A class is a model of the behaviour set of objects or instances of a class” (p 4).

The discussion of information modelling does confuse the issue but the emphasis on the data model includes mechanisms for querying and updating the state. He does argue that:

> “The behaviour of a class is defined by the information model that characterizes the state of class instances (objects), the methods that access this information (query methods), the methods that modify this information (command methods), and the constructor methods that allow the user to set the initial state of an object as it is being created” (p 5).

Through this statement he is endeavouring to place greater emphasis on behaviour rather over the data / information model.

When he discusses refactoring, he emphasises that refactoring is about modifying the structure of the program without modifying the external behaviour. He suggests that one of these refactorings is extracting common behaviour to a super class (p 206).

The bulk of the references place greater emphasis on behavioural requirements rather than data requirements; none ignore the need for data. Those with this emphasis include Barnes and Kölling (2006), Becker (2007), Bergin et al. (2005), Guzdial and Ericson (2005), Jeffries (2002), Langr (2005), Lewis and Loftus (2006), Morelli and Walde (2006), Sierra and Bates (2005), Stein (2003), and Wiener (2007). Bronson (2006) has a

mixed message. The data requirements emphasis is presented by Anderson and Franceschi (2005), Barker (2005), Brookshear (2007), Farrell (2006a; 2006b), Felleisen et al. (2001), and Malik (2006).

The texts that place more emphasis on modelling (Barker, 2005; Wiener, 2007), place greater emphasis on identifying entities from the real world or problem domain. To determine whether there is a data or behaviour focus, the reader has to look at what the authors regard as more important when defining the characteristics that belong to a class or object. Barker places emphasis on data modelling while Wiener emphasises behaviour. The real difference in terms of the end result is unclear since the foundation in modelling appears to have the same origin.

There were no comparisons of using a data-driven approach versus behaviour-driven approach. Nearly all the textbooks followed only one path to the design of the illustrated solutions and never provided the learner with an alternative. The exceptions were Sierra and Bates (2005) with their contrast between a procedural solution and an object-oriented solution and Jeffries (2002) with his willingness to show that developing a design means a willingness to follow false paths and to explore solutions. Neither of these actually compared a data-driven focus to a behavioural-driven focus.

None of the texts placed an emphasis using objects as a requirement of the language or framework. This category may come through in texts more closely aligned with teaching the use of a drag and drop type development environment such as Visual Studio. Focusing primarily on Java texts meant that these texts were not looked at in this study.

The other variation that did not seem to appear in the texts is the category where objects are liberated from the confines of program boundaries. The nature of Java type development environments doesn't lead to this type of thinking. Other environments such as Smalltalk, Common Lisp, and Jade where the concept of a program is less evident may lead more naturally to the presentation of this variation of object usage.

6.1.3 How do the texts address the nature of the problem solution?

With respect to the nature of the problem solution, the interview analysis revealed four variations. These variations are:

1. Provides required functionality
2. A data model of the problem space based (a real world model)
3. A behaviour-driven model reflecting objects in the problem space
4. A behaviour-driven model using useful abstractions

A number of texts make no reference to modelling nor place much emphasis on the nature of the solution. These include Anderson and Franceschi (2005), Bergin et al. (2005), Brookshear (2007), Jeffries (2004), Malik (2006), and Sierra and Bates (2005).

A strong push for a program as providing the required functionality is presented by Anderson and Franceschi (2005) with their view that "programming is like solving a puzzle. You have a task to perform and you know the operations that a computer can perform" (p 22). This theme is present as they help the learner to understand what the programming language can do. They make a reference to an object being a template or model and representing "something in real life" (p 20) but this isn't picked up again when talking about user-defined classes (Chapter 7) with the possible exception of statements like "Let's define a class that represents an automobile" (p 368). The examples appear to demonstrate that classes are based on something in the real world but there is no reinforcing of this theme throughout the text. No reference is made to any aspect of developing a model, either in the context of planning the writing of programs or in terms of the code being a model of something.

Malik (2006) contends that "Software consists of programs written to perform specific tasks" (p 6). He does talk about abstract data types in terms of real world entities (pp 492-493) but this is not an explicit definition or requirement.

Brookshear (2007) defines a program in terms of being an implementation of an algorithm, where an algorithm is the steps for accomplishing a specific task. He makes no specific reference to a program as a model although, like many texts, he uses examples where objects represent entities from the problem domain. Because this text addresses a range of programming paradigms, he uses generic definitions and then talks about how each paradigm addresses those issues.

Sierra and Bates (2005) place no emphasis on a program as a model. Their discussions of classes are as a template for objects and objects as entities that contain data and

behaviour relevant to the application (p 35). In this sense the focus is on providing the required functionality.

Bergin et al. (2005) makes no explicit reference to the program being a solution to a problem or to it being a model of anything. This is in part because the focus of this text is on introducing object-oriented techniques within an environment where robots do things. This focus will leave the learner with the impression that programming is about providing the required functionality to solve programming problems.

Jeffries (2004) places no emphasis on modelling but in writing the code, feels no constraint to remain with objects that reflect entities in the real world. He is prepared to use abstractions that serve a useful purpose and communicate the intent of the code.

Although Guzdial and Ericson (2005) talk of identifying classes by underlining "the nouns in the problem description" (p 344), there is no talk of a program as a model of the problem domain. Even when discussing naming and emphasising that the computer "doesn't *care* about names" and that "Names are for the humans" (p 39), there is no emphasis on being a model of the problem domain understandable to humans. The emphasis is on communication and the way that computers represent and refer to things.

Stein (2003) doesn't discuss modelling but here examples use classes that reflect things in the problem domain and she says in relation to identifying classes:

> "One way to figure out what things your program needs is to look for nouns and verbs. That is, in your description of the system, you will talk about the things that are a part of your program and the actions that they perform or are performed on them. The things -- the nouns -- are objects or entities that you will likely need to create. The actions are recipes that these things will follow. In a library, typical nouns include book, library card, and catalog; typical verbs include check in, look up, etc. We'll begin with the nouns." (Chap. 2)

The focus of this process suggestion is on modelling objects named in the domain description or in the real world.

The remainder of the texts refer to modelling of the real world but it is not always linked with being data driven or behaviour driven. In most cases, the data-driven or behaviour-driven focus is drawn from the usage of objects critical aspect analysis thus reinforcing the linkage between these two critical aspects. Those with a data driven focus include Barker (2005), Farrell (2006a; 2006b), and Felleisen et al. (2005).

For Barker (2005), the model is a data-driven abstraction of the domain or real world (p 9, 67, 337). She has examples of abstraction hierarchies based on natural objects (pp 4-8) and focuses on abstraction and its implementation as a hierarchical model (chapter 1). When introducing objects, Barker emphasises that an object becomes an "abstraction of a real world (physical or conceptual) object" (p 68). She reinforces this relationship between the code and the model of the real world when she talks of a method as defining "the business logic" (p 107). Even when talking about the interaction between objects, she contends that these are related to the real world interactions (p 122) and with respect to delegation she equates it with "delegation between people in the real world" (p 118).

Farrell places emphasis on "concrete objects in the real world" (2006a, p 4) and claims that the "object-oriented approach is said to be "natural" because it is more natural to think of the world as consisting of objects" (2006b, p 6). In providing examples for object-oriented thinking, Farrell refers to real world objects. Some examples are:

> "**Objects** both in the real world and in object-oriented programming are made up of attributes and methods" (2006a, pp 4-5)

and

> "Real world objects often employ encapsulation and information hiding. … When the details are hidden, programmers can focus on the functionality and the interface, as people do with real-life objects" (p 34)

In discussing the identification of objects, Farrell refers to it as data modelling (2006b, p7) thus emphasising a data driven approach to real world modelling.

With the emphasis on defining data types and class hierarchies, Felleisen et al. (2005) have argued for the use of a data-driven representation of "information in the problem domain" (p 15). Their concern is that the program represents "domain knowledge" (p

49) which incorporates both the data and the processing logic. The learner is repeatedly reminded of this relationship as portrayed in “Recall this kind of knowledge is domain knowledge. It is best to acquire as much basic domain knowledge” (p 105)

Although both Bronson (2006) and Soroka (2006) emphasise modelling of the real world, they make no reference to the approach taken.

Bronson (2006) contends that modelling relates to simulation when he says “A central feature of simulation languages is that they model real-life situations as objects” (p 10). This is then followed through by seeing the world as “full of objects” (p 13), modelling the behaviour of a coin (p 115), and chapters 5 and 6 including material on modelling.

Soroka (2006) introduces classes and objects based on modelling exercises (chapter 2, pp 32-46). He is consistent in mentioning this relationship between the real world and the program code. The initial emphasis is stated as:

> “The goal of designing with objects is to make the program correspond more visibly to the real world” (p 22).

He introduces the modelling aspects independent of any program code and places the focus on modelling specific real world entities (pp 32-37).

Those text with an emphasis on a behavioural-driven approach to modelling include Barnes and Kölling (2006), Becker (2007), Langr (2005), Lewis and Loftus (2006), and Morelli and Walde (2006).

Langr (2005) has a behaviour-driven focus and contends that objects are abstractions of real world entities (pp 12-14). This is not an emphasis that is strongly promoted in the text.

Lewis and Loftus (2006) state that

> “One of the most attractive characteristics of the object-oriented approach is the fact that objects can be used quite effectively to represent real-world entities. … In this way, object-oriented programming allows us to map our programs to the real situations that the programs represent” (p 42).

It could be argued that the examples illustrate this point and that they make the occasional reminder that there is this relationship between objects and real world entities (pp 43-44, 296, and 462-463) but again it is more like a background theme.

Like others, Morelli and Walde (2006) place some emphasis on the parallel between program objects and real world objects with the emphasis being on the program being a model of "a collection of real-world objects" (p 10). This emphasis is not strongly promoted through the main body of the text.

Barnes and Kölling (2006) promote behavioural modelling of the real world as the core of object-oriented program design. This focus on being a real world model starts in the opening chapter when they say "If you write a computer program in an object-oriented language, you are creating, in your computer, a model of some part of the world" (p 3). This is repeatedly revisited in different contexts. This emphasis is particularly noticeable when they talk about the cohesion of classes and say:

> "The rule of cohesion of classes states that each class should represent one single, well-defined entity in the problem domain" (p 212)

Here it isn't simply in identifying classes where they emphasise the use of the noun / verb analysis approach to identify things in the real world (p 394), it is in the definition of a design quality that they reinforce this link with the problem domain.

Wiener (2007) focuses on modelling in his text and says:

> "Modelling has been the basis of many scientific disciplines. A model of a system represents an abstraction that captures the essence of the system's behaviour" (p 4).

He reinforces this emphasis on the relationship between the model and the problem domain when he says, "The application domain dictates the model that is appropriate" (p 8). However, he does recognise that the more abstract entities are required when he says, "In the world of software development, we often are required to model more abstract entities - things that are not physical objects" (p 8). This modelling emphasis is carried throughout the text.

Becker (2007) emphasises behaviour-driven modelling with a hint toward these being real world models. This emphasis is shown through the opening section "Modelling with Objects" in chapter 1. In his introduction to the chapter, he says:

> "A computer program usually models something. … Whatever that something is, a computer program abstracts the relevant features into a

> model, and then uses the model to help make decisions, predict the future, answer questions, or build a picture of an imaginary world" (p 1).

Here he is talking about the computer program as an "imaginary world". In that sense, it is a model of the problem space. This theme is used throughout the text. An example is when talking about an example of collaborative classes where he talks of "modelling a person" (p 392).

He emphasises a behaviour-driven approach through the types of questions that he asks, such as "What is the date of the concert?", "How many tickets are still unsold?" and "What is the total value of all the tickets sold to date?" (p 2). Becker is expecting the system to be able to report things even though some of these may be simple data entities. Other questions are dependent on the results of calculations.

Providing the required functionality is a common theme through all the texts. It is the primary theme for Anderson and Franceschi (2005), Bergin et al. (2005), Brookshear (2007), Jeffries (2004), Malik (2006), and Sierra and Bates (2005). Guzdial and Ericson (2005), and Stein (2003) talk of using the noun / verb analysis to identify classes but they don't talk about this as modelling the real world or the problem domain. A data-driven real world model is emphasised by Barker (2005), Farrell (2006a; 2006b), and Felleisen et al. (2005). A behaviour-driven model is the focus of Barnes and Kölling (2006), Becker (2007), Langr (2005), Lewis and Loftus (2006), and Morelli and Walde (2006). Bronson (2006), and Soroka (2006) place an emphasis on modelling but are unclear as to whether it is data or behaviour driven. For Langr (2005), Lewis and Loftus (2006), and Morelli and Walde (2006) the emphasis on a real world model is not very strong. The possibility of using useful abstractions is present in Wiener (2007) with the idea that objects may need to be introduced that are not in the real world or problem domain, and in Becker (2007) where he talks of the program as a model of an imaginary world. Jeffries (2004) illustrates the use of useful abstractions but makes no reference to the program as a model.

6.2 How do the texts address the design characteristics of an object-oriented program?

The design characteristics reflect the influences on the way that the software is designed. For the design characteristics four critical aspects were identified that impacted the approach to design. These critical aspects are:

1. technology,
2. program design principles,
3. cognitive processes, and
4. modelling.

A difficulty with completing this analysis is related to the way that this material is distributed through the textbooks. All deal with technology issues and program design principles. Some textbooks discuss modelling but few deal explicitly with the cognitive processes.

Another difficulty is dealing with the variations in interpretation of the programming related concepts that comprise these critical aspects. This depth of analysis was not performed on the interview data and is not intended to be the focus in this analysis. As a consequence, it is sometimes difficult to identify how these critical aspects are handled within these texts.

In this section the textbooks are examined from the perspective of these critical aspects to determine the emphasis placed on each within the texts.

6.2.1 How do the texts address the influence of technology?

This critical aspect places the influence on the design in relationship to the technologies being used. The variations identified are:

1. Constrained by language features
2. Constrained by paradigm constructs
3. Technology is seen as an implementation tool

All but one of the textbooks examined in this study used a single programming language. In most cases, the language was Java. Only two of these textbooks emphasised the language as a constraint to design.

Malik (2006) portrays a strong emphasis on knowing the "the basic components of a programming language to be an effective programmer" (p 22). This theme is again emphasised when talking about the need to understand the syntax and semantic rules of the programming language (p 29). Malik reverses the perspective to a paradigm focus when he says "A programming language that implements OOD is called an **object-oriented programming (OOP)** language" (p 21). The implication is that there are object-oriented concepts which can be discussed independent of a language. Malik's approach in the text is to focus on the language thus giving the impression that object-oriented programming requires an object-oriented language.

Anderson and Franceschi (2005), although stating that the text "has been written to provide introductory computer science students with a comprehensive overview of the fundamentals of programming in Java" (p v), endeavour to place emphasis on the use of object-oriented or paradigm constructs. They state that "Smalltalk provided the programmer with a new tool: classes and objects of those classes" (p 20), there is an implication that what makes a program object-oriented is its use of classes (p 90) and the use of paradigm constructs such as inheritance and polymorphism (p 681).

Bergin et al.'s text (2005) reflects an emphasis on paradigm constructs with their emphasis on polymorphism as the cornerstone of object-oriented. They want to emphasis the characteristics that make a program object-oriented. This is clearly reflected in the discussion on the advantages of using polymorphism over the conditional 'if' construct (p 127). However, there is also a clear message that design objectives play a role in the choice of polymorphism over the conditional 'if' construct:

> "When a program contains a large number of IF statements and only some of them need to be changed for the new problem, it can be a major task keeping the updates consistent" (p 127)

On the surface, Soroka (2006) appears to focus on the programming language. However, the author initially endeavours to give some idea of what computer science is. He argues that

> "my goal is to teach you the fundamentals of computer programming that will apply to any language you have been asked to use. I hope to teach you skills that will be useful in learning new programming languages" (pp vii-viii).

However, the focus is then on the programming paradigm which he says

> "is a generally accepted approach to programming. It's not a specific language, but rather a general approach as to how we will program and what kinds of languages will be accepted" (p 20)

The paradigm concepts then take centre stage with the occasional reference to generic constructs and design objectives. This same focus on paradigm constructs is shown in a number of other texts (Bronson, 2006; Farrell, 2006a, 2006b). Lewis and Loftus (2006), although claiming in the introduction to focus on problem solving techniques, focus on the use of paradigm constructs in the design of their solutions. Wiener (2007) demonstrates this in his text when he focuses on the paradigm constructs and how these are modelled.

The remaining texts do cover language and paradigm issues but in the context of generic programming principles or the thought process.

What is illustrated in all of the texts is the difficulty of giving appropriate weight to technical issues as well as to the more generic program design issues and the thought processes. The exception is Jeffries (2004) who only introduces the language constructs and paradigm concepts if they are required to solve his programming problems and help illustrate the thought processes behind his approach to programming. Langr (2005) shares some of Jeffries' emphasis when he declares:

> "Agile java is not an exhaustive treatise on every aspect of the Java language. It instead provides an agile approach to learning the language" (p 4)

However, Langr does place more emphasis on the language and paradigm constructs than Jeffries.

Anderson and Franceschi (2005) and Malik (2006) place greater emphasis on the language features. Bergin et al (2005), Bronson (2006), Farrell (2006a; 2006b), Lewis and Loftus (2006), Soroka (2006), and Wiener (2007) place emphasis on paradigm constructs. The remainder cover language features and paradigm constructs as implementation tools.

6.2.2 How do the texts address the program design principles?

With respect to this critical aspect, five variations were identified. These five variations are:

1. decomposition into modules
2. sensible usage of paradigm constructs
3. applying or reusing generic principles and implementation patterns
4. designing guided by design objective principles
5. Implementing design principles as an outcome of the thought process

Because the program design principles are often distributed through the texts, it is easier in this analysis to discuss the way in which the texts place emphasis on the different variations rather than link texts with specific variations.

Anderson and Franceschi (2005) provide an example of an emphasis of the paradigm influence on design when they discuss decomposition as being the job of a programmer.

> "As a programmer, your job is to decompose a task into individual, ordered steps of inputting, calculating, comparing, rearranging, and outputting" (p 22)

The paradigm focus is picked up when they had previously talked about the class as being the mechanism for encapsulation and thus leading to being the mechanism to enable the programmer to use decomposition.

> "A class enables the programmer to encapsulate data and the functions needed to manipulate the data into one package" (p 20).

This emphasis is paradigm specific rather than emphasising encapsulation as a principle that applies not only to classes but also to any enclosing of code to enhance reuse and understanding.

As well as encapsulation, they discuss other generic principles. In this case, it is reuse.

> "The most important advantage to inheritance is that in a hierarchy of classes, we write the common code only once. After the common code has been tested, we can reuse it with confidence by inheriting it into the subclasses. And when that common code needs revision, we need to revise the code in only one place" (p 640)

Again, the introduction of this generic principle is based on a paradigm specific facility. Code reuse is seen as a by-product of using inheritance rather than inheritance as a mechanism for enhancing code reuse. The objective is not to understand reuse but rather to understand inheritance and its implications.

Bergin et al. (2005) place emphasis on the communication of paradigm concepts ahead of the generic principles and the design objectives. This is built on a desire to ensure that the object-oriented concepts are understood early. This does not mean that they do not see design objectives influencing the approach to design. They clearly state what they believe makes a good program when they say:

> "Writing understandable programs is as important as writing correct ones; some say that it is even more important. They argue that most programs initially have a few errors, and understandable programs are easier to debug. Good programmers are distinguished from bad ones by their ability to write clear and concise programs that someone else can read and quickly understand" (p 58).

However, this is not the dominate theme that runs through the text. It is more likely that a design objective will be mentioned as part of a discussion of some paradigm construct as in:

> "When a program contains a large number of IF statements and only some of them need to be changed for the new problem, it can be a major task keeping the updates consistent" (p 127).

Here the concern for maintenance is discussed in relation to a preference for using polymorphism rather than the "if" statement. Although this discussion isn't driven from the perspective of the design objective of maintenance, it is this objective that is illustrated as providing the base for the decision. The authors clearly see the design objectives as drivers but the text is not structured to place the emphasis on these objectives.

Morelli, and Walde (2006) emphasis the influence of the generic principles when they state that

> "Object Orientation (OO) is a fundamental problem-solving and design concept, not just another language detail that should be relegated to the middle or the end of the book (or course)" (p xv).

They outline the "principles of object-oriented design" but in a way that is dominated by generic principles (p 17-18). There are design strategies included in the list but they only dominate it because they are listed first. A design process, based around the principles of "divide-and-conquer" or decomposition, is also introduced but focuses on object concepts rather than the objectives of good design (p 25). They do revisit their strategy in case studies throughout the book (e.g. pp 78-86). To reinforce the generic principles, they return to discussions of these in the chapters (e.g. pp 89-90).

Barker (2005) emphasises generic principles as the drivers for design with a specific focus on abstraction as it relates to modelling starting in chapter 1 and this is followed through in chapters 8 through 12 and chapter 14. She lays out her modelling theory built on the generic principle of abstraction and the use of object-oriented concepts of classification, generalisation, and inheritance as the tools for implementing that abstraction. This is particularly clear through chapter 1 where she has section headings "Simplification through abstraction" (p 3), "Generalization through abstraction" (p 4), "Organizing abstractions into classification hierarchies" (p 5), and "reuse of abstractions" (p 9). Abstraction is seen as the "basis for software development" (p 8) and as the tool to "simplify our view of the world" (p 3).

Although she recognises that “there are an unlimited number of possibilities” for developing an object model for any particular problem (p 364), there is little done to discuss what might be characteristics of a good design. The modelling examples tend to work toward a predefined solution without consideration of variations. Even where she eliminates the use of two action listener event handlers by using an “if” statement, there is little discussion of the value of removing the second event listener other than eliminating redundancy (p 746).

Barnes and Kölling (2006) place a lot of emphasis on design objectives such as maintainability, and simplicity. These design objectives are linked with generic concepts such as encapsulation, modularity, decomposition, abstraction, and code reuse. They declare in their introduction that

> “Instead of introducing a new concept and then providing an exercise to apply this construct to solve a task, we first provide a goal and a problem. Analyzing the problem at hand determines what kinds of solutions we need. As a consequence, language constructs are introduced as they are needed to solve the problem before us” (p xxi).

This approach leads to linkages between the language constructs, the object-oriented concepts, the generic principles, and the design objectives.

In the discussion of design characteristics, there is limited use of contrasting examples or variations. For example when talking of cohesion, there is not a range of examples with varying degrees of cohesion visible. The example provided shows good cohesion only for two methods. This could easily have been taken up as a refactoring exercise in the following section to demonstrate how a poor design could be improved through refactoring for better cohesion.

This style of example is used when introducing inheritance (Chapter 6). Here a solution without inheritance is shown and then the revised version using inheritance demonstrates how inheritance can be used to remove duplication and to reduce the effort of later changes. Between the two examples, there is a discussion on the design of the application emphasizing the duplication that exists. Following the inheritance implementation there is a discussion on the advantages of inheritance couched in terms of design objectives. The emphasis is placed on avoiding duplication, code reuse, maintenance, and extendibility (pp 246-247).

Becker (2007) initially uses a robot environment to provide a problem space for learning the concepts that he wishes to communicate and he summarises the concepts as patterns at the end of each chapter. However, the problem context is not consistently used to introduce his main themes.

He places emphasis on maintenance and ease of understanding of the program as design objectives with statements like:

> "We will find that extending the language makes it easier to modify the program. With the intent more clearly communicated, it is easier to find the places in the program that need modification" (p 56).

These objectives do not drive the learning of the language but they do have a high visibility within the text. An example of this is when in relation to the issue of extension versus modification, Becker says:

> "When a class is open for extension it can be modified via subclasses without fear of introducing bugs into the original class or introducing features that aren't generally needed" (p 67).

Later he has a chapter on "Building Quality Software" (Chapter 11), in which he deals with quality objectives including communication of the code, program testability, and maintenance. He also has a section on managing complexity in which he discusses encapsulation, cohesion, information hiding, and coupling. Other principles such as the substitution principle (pp 645-646) are introduced in other places in the text.

Guzdial and Ericson (2005) promote design objectives through the introduction of "creating methods" early (pp 50-59), but the intent is aimed at developing an understanding of "how" to manipulate media content and at being able to communicate that understanding to others. Introducing the use of methods early allows them to add structure to their code through the use of meaningful names. They contend, "Much of Programming is About Naming" (p 38). This promotes their theme of "teaching people to program in order to communicate" (p xxi).

As a consequence, Guzdial and Ericson not only present different paradigm concepts and language features as solutions to particular problems, but they revisit the language

features to manipulate different media or to achieve alternative tasks with the same media. This revisiting reinforces the learning and encourages the development of a conceptual technical understanding of what is involved in manipulating media. Underlying this style of teaching is the theme that the application of design objectives is an integral part of the process.

Felleisen et al. (2005) argue from the perspective that there is a need to be able to reason about the design of a program (see next critical aspect, p 213). However, the nature of the text doesn't foster the development of these thought processes. The nature of the examples and related discussion focuses on good design of class hierarchies from a data or information analysis perspective. The text lacks alternative solutions that can be used to discuss the relative merits of design approaches and the learner may be left with an impression that there is some obvious way that class hierarchies are designed. As a consequence the learner, although picking up habits of design objectives, may not comprehend the reasoning behind those objectives.

Stein (2003) shows an emphasis on generic principles guiding design by using the generic principles to provide structure to the text. Stein talks about dispatch when discussing procedural conditional statements and then refers to object-oriented dispatch when moving to polymorphism. Although the paradigm specific constructs are present and there are references to some design objectives, the overall emphasis is on generic principles and how they are implemented with paradigm specific constructs. In the chapter on the design process, she does emphasise the issue of maintenance but this isn't a theme picked up consistently throughout the text.

Sierra and Bates (2005) place their emphasis on design objectives primarily through dialogues about why you would use particular object-oriented concepts or the advantages of using an object-oriented approach verses an procedural approach. Examples of this are:

> "Brad smiled, sipped his margarita, and *wrote one new class*. Sometimes the thing he loved most about OO was that he didn't have to touch code he'd already tested and delivered. "Flexibility, extensibility…" he mused, reflecting on the benefits of OO" (p 29).

> "Even though the methods don't really add new functionality, the cool thing is that you can change your mind later. You can come back and make a method safer, faster, better" (p 82).
>
> "You get a lot of OO mileage by designing with inheritance. You can get rid of duplicate code by abstracting out the behaviour common to a group of classes, and sticking that code in a superclass. That way, when you need to modify it, you have to only one place to update, and *the change is magically reflected in all the classes that inherit that behaviour*" (p 182).

Here Sierra and Bates, through the telling of a story, emphasise the design advantages of using objects both in terms of protecting code from change and also in making it easier to change code later if there is a requirement. The stories are short but to the point and of course include a little humour.

Langr (2005) illustrates how thought processes can implement design objectives when he places emphasis on communication. He argues that for the code to communicate, it must be readable and understandable. He emphasises these design objectives when he says: "ensure code is clean and expressive, clearly stating the intent of the code" (p 53), "understand what a class does and how it does it" (p 155), "view the names of tests to understand all the functionality that a given class supports" (p 155), and "Code tests as comprehensive specifications that others can understand" (p 155). Maintenance is emphasised through references to ease of change (p 181, 183, 188), cost of maintaining code (p 53), importance of fixing code problems now" (p 95), one reason to change (p 112), minimise impact on other parts of the code (p 168), simple design (p 147), testability (p 155), extension and not modification (p 183). With this emphasis, Langr clearly shows that the thought process that leads to expression in code is built on understanding the significance of design objectives.

Jeffries (2004) introduces design objectives or principles but not in the form of "design for maintenance". These principles are instead expounded as ways of thinking and working on software in order to maintain the desired qualities. This is illustrated in a lesson on design experiments when Jeffries says:

> "My view today is that it's better to start delivering useful software with a simple, clear design and to evolve the design as I go forward" (p 8).

Design objectives are something that is part of the thought process, underlining the activities of the master. They are also open to change as the master learns more of the language, paradigm, and solutions that work or fail within the given context.

This section has provided examples of how texts have endeavoured to introduce program design objectives. Anderson and Franceschi (2005), Bergin et al. (2005), Morelli and Walde (2006) contain examples of sensible use of paradigms. Barker (2005), Barnes and Kölling (2006) illustrate the use of generic principles while Stein (2003) uses the generic principles to structure her text. Becker (2007) provides an example where the design objectives are specifically dealt with in a separate section, while Guzdial and Ericson (2005), and Felleisen et al. (2005) integrate these ideas into the flow of the text. Sierra and Bates (2005) use dialogues to draw out the design objectives. Langr (2005) and Jeffries (2004) provide an emphasis on the influence of thought processes on the use of design principles.

6.2.3 How do the texts address the cognitive processes?

In relation to this critical aspect, the focus is on the thought processes that are used in the development of software or in working with the paradigm specific concepts and the generic principles. The variations in this critical aspect are thinking in terms of:

1. knowledge of the language,
2. objects,
3. good practices,
4. the impact of design decisions, and
5. implicit recognition and correction of poor design.

Some of the key characteristics of these thought processes are communicating the intent of the code, identifying duplication and eliminating it, iterative design, and a focus on continual learning. The authors with this focus talk about rules or principles that guide their work but these are applied thoughtfully and not as rigid rules that must be followed.

The emphasis on the thought processes is not present in most of the textbooks. Some like Becker (2007), Lewis and Loftus (2006), Malik (2006), and Morelli and Walde (2006) discuss programming as a problem solving process. Lewis and Loftus (2006),

and Malik (2006) even describe problem solving steps but the emphasis is not on this as a cognitive activity. A similar argument applies to texts that focus on design objectives or paradigm concepts. Fostering a thought process does not feature.

Felleisen et al. (2005) imply that there is reasoning about programming required when they say:

> "In general, students must learn that programming is *not* just the act of writing down classes and methods that work, but that programming includes reasoning about programs, editing them, and improving them as needed" (p 175).

From this statement, it would appear that Felleisen et al. believe that it isn't simply a case of choosing appropriate classes to include in the model. There is a process of reasoning which will involve attempts at different variations to resolve. They later contend that

> "Programming is about subtleties and expressing them is important. Others will read your program later and modify it. The clearer you can communicate to them what your program is about, the higher the chances are that they don't mess up these carefully thought-through relationships" (p 190).

This indicates that it is more than just reasoning about programs that is important but it is also important to express that reasoning in code so that others can understand the reasoning from the code.

Guzdial and Ericson (2005) are not explicit about the cognitive activity but with their emphasis in naming and computational media, they place an emphasis on communication. They argue that:

> "If you can only manipulate media with software that someone else made for you, you are limiting your ability to communicate" (p 10).

It is also their contention that through learning how to work with media through programming, the learner will learn:

> "*how* the tools work" and as a consequence have "a core understanding that can transfer from tool to tool. You can think about your media work in terms of the *algorithms*, not the *tools*" (p 11).

The algorithms provide a cognitive framework for understanding how tools approach the task. This approach also echoes the thinking of diSessa (2000) who promotes programming as a way of developing computational literacy or deeper understanding of how things work. Although this is developing a way of thinking about a specific type of computation, it is not a thinking process that is aimed at producing programs. The programs are like essays designed to express an understanding of what it means to manipulate media.

Langr (2005) emphasises the use of the test-driven development (TDD) process but does so in a way that emphasises the thought processes involved. He initially hints at this when he says "Programming involves thinking and problem solving" (p 4). The thought process is not always clear in the text.

Langr further emphasises the thought process when he says:

> "Part of crafting software is understanding that code is a very malleable form. The best thing you can do is get into the mindset that you are a sculptor of code, shaping and moulding it into a better form at all times. Once in a while, you'll add a flourish to make something in the code stand out. Later, you may find that the modification is really sticking out like a sore thumb, asking for someone to soothe it with a better solution. You will learn to recognise these trouble spots in your code" (p 94)

and

> "You will also learn that it is cheaper to fix code problems now rather than wait until the code is too intertwined with the rest of the system. It doesn't take long!" (p 95).

Langr seems to be aware that the code structure can communicate with the programmer and that this is a skill which the programmer will develop over time. He also indicates that he understands the principle of crafting the code to avoid costs later on. There are ways of thinking underlying the basic skills which help ensure a quality of code and a reduction in the later maintenance costs. This is also reflected in the statement:

> "The more you can continually craft the code as you go, the less likely you are to hit a brick wall of code so difficult that you cannot fix it cheaply" (p 53)

This carries the idea that the way of thinking about your code helps produce quality code. These statements are used to reinforce practices but reflect an attitude toward the way code is produced.

Jeffries (2004) emphasises the cognitive activities as he endeavours to show the thought processes that he utilises as he develops a small system. As he works through the design of the system he discusses design objectives and the ability to communicate the program's intent through the code. There are false development paths as he endeavours to learn the language and the ways of doing things in that language. Behind his thought process are the values and practices of extreme programming but these are seen as needing to be kept in balance. This has been reflected in a recent discussion on the YAGNI (you aren't gonna need it) rule in the extreme programming mailing list (Beck, 2008; Jeffries, 2008). Both Beck and Jeffries have emphasised that the rule should not be applied without thought. It is not a rule that means that we should totally ignore all possible future requirements. Rather it is a rule that helps the programmer realise that they should not spend too much time on things that are not part of the current development iteration. Other rules, such as the need for feedback, removal of duplication, and using the simplest design that will work, have to be taken into consideration and to be held in balance.

Jeffries reveals a lot about his thought processes and the false starts that are part of the process. He is not trying to idealise the process, but to provide a realistic description of what is occurring.

In the "Foreword" of the book, Robert Martin says "You won't just learn facts about C# - you'll learn a master's principles, patterns, and practices" (p xvii). The book provides an insight into the operations of a master programmer. As Martin puts it, "You'll gain insights into the thought processes, personal values, and subtle gestures of a master" (pp xvii-xviii)

Jeffries also emphases this idea in his introduction when he says

> "For the past few years, I've been experimenting with these ideas. I start with a very simple design idea and implement features in a customer-driven order, using refactoring to keep the design just barely sufficient to the moment. One question I've been interested in was whether the code necessarily becomes brittle, hard to change, and unreliable or whether more up-front planning and design would give better results, in term of substantially less rework, faster progress, or a better resulting design" (p xxvii).

This book could be said to show that the master bears his soul so that the apprentice may see the real workings of how to approach a programming task.

As might be expected, the book contains a lot of design objectives or principles but not in the form of "design for maintenance". These principles are instead expounded as ways of thinking and working on software in order to maintain the desired qualities. This is illustrated in a lesson on design experiments when Jeffries says:

> "My view today is that it's better to start delivering useful software with a simple, clear design and to evolve the design as I go forward" (p 8).

Design objectives are something that is part of the thought process, underlining the activities of the master.

Again with respect to testable software, the emphasis is on the advantages of using tests to drive the writing process and the code quality. Testability is not a goal, it is part of the writing process and it fosters design objectives that aid the overall development process (p 17). Later as a reminder of the value of tests, he expresses how "automated tests record our intelligence for use later" (p 65) and strongly states "If you can't test it, it's wrong" (p 86).

Jeffries places some emphasis on an attitude toward continual improvement with statements like:

> "And yet, in this chapter we've recommended putting in code that's clearly not up to our standards of craftsmanship. What's up with that? Are we being inconsistent, or are we taking a risk that things won't get cleaned up? We're sure we're not inconsistent: We're intentionally building from rough toward

> smooth, as you've already seen. We put the code in the Form, and now we're moving it to a better place. We'll keep doing that going forward. … What I hope you'll see is that everything we *do* visit we make a little better each time we pass through" (p 47)

Jeffries sees the master programmer as continually improving code but not necessarily putting in the best code possible at the first attempt. The master recognises that there are times when it needs to be made to work before being put right. As he says:

> "There's an old saying in programming: "Make it work, make it right, make it fast." That's what's happening here. We have made it work. Now we're in the process of making it right, and if we need to, later, we'll make it fast." (p 55)

And above all, the master has an attitude of learning at every opportunity. Jeffries says:

> "One more thing I've noticed that might help you. Because we know that every day's work has to be written up for the Web site, we make a point of doing little retrospectives all the time, basically going over what has happened and what we have learned. I think it's helping us learn faster and remember more. You might want to try that technique yourself" (p 113)

and

> "The lesson should be: we should learn as much as we can, all the time. And we should be careful to keep our learning focused on safely delivering the customer features he needs" (p 200)

This culminates in the attitude "I plan to be smarter later than I am now" (p 217).

All of these themes are revisited repeatedly throughout the book reminding the reader that it is not the language, paradigm concepts, or generic principles that drive our programming. Instead, it is design objectives wrapped in an attitude toward software development that delivers quality software and user functionality.

Jeffries (2004) writes his text from the perspective of the thought processes of a master programmer while Langr (2005) gives emphasis to the thought processes within the particular development approach that he emphasises in his text. Guzdial and Ericson (2005) promote the idea of communicating understanding in the code and Felleisen et al. (2005) promote reasoning about the code without giving it major emphasis. Becker

(2007), Lewis and Loftus (2006), Malik (2006), and Morelli and Walde (2006) talk of problem solving but give little emphasis to it throughout the text.

6.2.4 How do the texts address the issue of modelling?

With this critical aspect, design is seen as modelling. Three variations were found with respect to understanding the objective of modelling. These are:

1. Reflects real world objects
2. Uses useful abstractions
3. Communicates program design

Eight of the texts make no reference to the concept of a model or modelling. Four of these also make no reference to an approach to identifying classes (J. Anderson & Franceschi, 2005; Bergin et al., 2005; Brookshear, 2007; Guzdial & Ericson, 2005). Guzdial and Ericson (2005), with the emphasis on media literacy, is less concerned about how a program is developed. Two of the texts use modelling as their primary theme and work extensively with the unified modelling language (UML) to represent their models (Barker, 2005; Wiener, 2007). The remainder of the texts which refer to an object-oriented program as a model do so in the context of it being a model of the real world but do not place modelling as a central theme. The material related to a program as a model (see section 6.1.3, page 196) has already discussed the emphasis on how the authors have related a program to a real world model and use useful abstractions.

For Barker (2005), the emphasis is on modelling the domain or real world through the use of abstraction (p 9). The modelling focus in her text is shown by the sections dedicated to object modelling (Part 2, Chapters 8 through 12). She uses the “noun phrase analysis” technique to identify possible objects in the problem domain further emphasising the link between entities in the domain and the object model. She later discusses how the created model is transformed into Java code (Chapter 14).

Wiener (2007), like Barker, introduces object-oriented modelling concepts in his opening chapter. He says:

> “Software systems are often constructed by examining models of the problem being solved” (p 4).

Like Barker he emphasises modelling of the problem domain but he acknowledges that more abstract entities may be required (p 8). He also provides advice on translating the model into code.

Although modelling may not be the central focus of the remainder of the texts that talk about a program as a real world model, a number use some form of modelling notation in order to promote understanding of the concepts.

Felleisen et al. (2005) have a strong modelling emphasis but refer to it as designing classes and class hierarchies. They use a modified UML type notation to discuss the models that are used as the basis for writing code. Barnes and Kölling (2006) represent their models using a notation that is an adaption of UML and is supported by the BlueJ environment. Lewis and Loftus (2006) introduce the UML notation (pp 167-168) following the introduction of classes and objects in the preceding chapter. Soroka (2006) introduces UML notation alongside the concepts of class and object (p 31).

Farrell makes no reference to UML in her Java text (2006a) but her third last chapter (chapter 10) in the generic object-oriented introduces modelling using UML (2006b). Becker (2007) introduces modelling with object (p2) and class diagrams (p 7) and uses the notation throughout the book but not with a frequency or in a manner that emphasises modelling. Bronson (2006) introduces a UML type diagram as an object description (p 116) but leaves it until later in the text to introduce UML for program design (pp 271-278). Langr (2005) introduces UML early referring to it as being used to model object-oriented systems (p 16-18) and then uses it to help explore the developing designs throughout the text. Morelli and Walde (2006) introduce UML as a notation for representing object-oriented programs (p 11) and then use it to illustrate program designs.

Of the texts that make no mention of modelling or models, two use a diagramming notation. Malik (2006) introduces UML class diagrams about half way through his book (pp 450-451) and then uses them in discussing specific program designs. Sierra and Bates (2005) use a UML type notation to illustrate and discuss program designs but make no introduction to the notation nor refer to it as modelling.

Jeffries (2004) does not refer to the program as a model nor talk about modelling but does show that he is prepared to use useful abstractions if they help communicate the programs intent. He says:

> "Look at the code; can you see where it does those seven things? You'll be able to figure it out. But why should we have to figure out what code does? Why doesn't the code *say* what it does? We call that expressing intention" (p 42).

This is further emphasised when talking about "program by intention" (p 87) where he says "Programming by intention goes a step further than expressing intention after the fact. When we program by intention, we reflect on what we intend to do and we write the simple method first" (p 88). Expressing intent and making code readable is part of the master's thought processes. The value of programming by intent is further expressed when he says "First, we don't have to remember so much. As soon as we know what we intend to do, we write it down. Then we just refine it until the code is there. Second, the code comes out well-factored without as much refactoring. We get better code, and it takes less time!" (p 88).

Readability of the code leads to another attitude and this is that the code can tell us things about when it needs to be changed or enhanced. Jeffries says

> "When you find yourself writing procedural methods and utility methods and railing at how the objects aren't helping you, this too, is telling you that there's a missing object." (p 86)

And later

> "when you're looking at part of the system that doesn't seem just right, trust that feeling. If you know what to do to improve it, do so the next time you're working in that area. And if you *don't* know what to do, take it as a signal that it's time to do some research, time to add a few things to your bag of tricks" (p 183)

The code is an expression of the programmer's thinking. If that expression is not clear then the code needs changing. It is telling us something and the master programmer can recognise these signs and continually improve the code.

Of the texts that talk of a real world model, Barker (2005) and Wiener (2007) focus on modelling. Felleisen et al. (2005) emphasise class design using a modelling approach. The major group of texts, including Barnes and Kölling (2006), Becker (2007), Bronson (2006), Langr (2005), Lewis and Loftus (2006), Morelli and Walde (2006), and Soroka (2006), make use of some form of modelling notation to support their discussion of program design without emphasising modelling. Farrell uses a modelling notation in her generic book (2006b) but not in her Java specific book (2006a). Two texts, Malik (2006) and Sierra and Bates (2005), make no mention of real world models but utilise a modelling notation to discuss program designs. Jeffries (2004) emphasises the codes ability to communicate without making any reference to modelling or models.

6.3 Conclusion

The analysis of the textbooks proved to be easier in relation to the nature of an object-oriented program than in relation to the design characteristics. The space of learning in relation to this outcome space was more visible in the texts with a clear emphasis on defining the authors' perspectives early in the texts.

Seventeen of the twenty texts promote the idea that an object-oriented program is made up of interacting objects through definitions of what an object-oriented program is. Four of these fail to maintain the emphasis through the text. Procedural logic constructs were frequently dealt with ahead of the creation of objects. Some attempted to overcome this by introducing the use of objects early and emphasising the message passing aspect of method calls. Only two texts endeavoured to provide a contrast between procedural approach and an object-oriented approach that highlighted the difference between a sequential flow of control and an interacting entities flow of control. The discussion of algorithms consistently refereed to a sequential flow of control even when the text was endeavouring to promote interacting entities. Only one text focused purely on a sequential flow of control and the remaining text made no reference to flow of control.

With respect to object usage, the focus in eleven of the texts was on the behavioural requirements for the objects rather than the data requirements. All acknowledged the importance of data but this was seen as serving the behavioural requirements for many writers. Seven focused on objects representing the data requirements through being abstract data types. The remaining text (Bronson, 2006) gave equal priority to both data

and behavioural requirements. The two books with a modelling emphasis (Barker, 2005; Wiener, 2007) focused on identifying objects in the problem domain and then either their data requirements or their behavioural requirements. None of the texts emphasised the idea that objects could be liberated from the constraints of program boundaries.

With respect to the nature of the problem solution, eight of the texts primarily focused on providing the required functionality although two did refer to the use of noun / verb analysis for the identification of objects. This was an implied focus for all except Anderson and Franceschi (2005). The remaining eleven texts talk of objects representing entities in the real world. Four of these eleven preferred a data driven approach to identifying the entities and two made no specific preference. The remaining five texts promoted a behavioural approach to identifying the entities. Only two texts acknowledged the use of useful abstractions although this was not a major focus. Another text preferred to talk of the program as a model of an imaginary world rather than the real world.

The design characteristics outcome space was more difficult to identify on a text by text basis. It was more visible when stepping back and looking at the overall focus of the writer.

With respect to the influence of technology critical aspect, all the texts placed some emphasis on language features but two had this as their primary focus. Seven focused more on the paradigm constructs while the remaining ten saw the technology more as an implementation tool.

Of all the critical aspects, there was greater diversity in the identification of variations with respect to the influence on program design principles. Examples of the different variations were identified rather than identifying the primary focus of the text. Three texts illustrated the influence of paradigm constructs on design and three illustrated the influence of generic principles. One text was structured around the generic principles. The influence of design objectives was covered in a separate section in one text, and integrated into the flow of the text by two others. One used specific dialogues throughout the text to give emphasis to the design principles. Two provided an emphasis for the influence of thought processes on the use of design principles.

The cognitive process critical aspect was difficult to identify in all but two of the texts. These two texts placed the emphasis on a process from developing software rather than on the programming constructs or design principles. One text used as its major theme the writing of code to communicate understanding of the required processing. Another promoted reasoning about the code but not as an overriding theme. Four gave emphasis to problem solving strategies but although two described the steps of such strategies, none promoted these in the text. Of the critical aspects identified in this research, this was the critical aspect that gained the least attention in the textbooks.

Two of the texts used the issue of modelling as the primary theme for the writing of their texts while eight made no reference to modelling. A third text appears to have a modelling emphasis but the authors refer to it as the design of classes and class hierarchies. Of the remaining texts that emphasised a program as a model of the real world, seven made use of a modelling notation for discussing their code examples but placed limited emphasis on modelling. Two of the texts that made no reference to models or modelling made use of a modelling notation.

The difference in the way that the critical aspects have been identified in their usage within the textbooks demonstrates the core difference between the two outcome spaces. The nature of a program outcome space (the ‘what’ aspect) is difficult to ignore and all the critical aspects are given some emphasis in all of the texts. The design characteristics (the ‘how’ aspect) sees much more variation in the emphasis given to each of the critical aspects with the cognitive process critical aspect being given very little emphasis.

Chapter 7. The implications of the critical aspects

The objective of this research was to find a way of improving the learning of object-oriented programming. The initial question that started the journey for the current study was “How do we improve learning of computer software development?” Through exploration of the literature and relational learning theory, the emphasis on improvement was replaced by a focus on understanding the range of ways in which practitioners are aware of object-oriented programming. This led to the following questions.

1. How does the awareness of an object-oriented program vary among the ‘practitioners’?

    c. What are practitioners’ ways of experiencing object-oriented programming?
    d. What are the “critical aspects” that distinguish the different variations in awareness expressed by ‘practitioners’?

2. What spaces of learning are opened up by introductory programming texts with respect to the ‘critical aspects’ arrived at from the previous questions?

This chapter provides a summary of the results of the current study and then examines these results in the light of the literature. It discusses areas where the literature and the results of this study are in agreement and reviews areas of disagreement. It raises some further questions that arise from this analysis.

7.1 Summary of results

The current study explored the nature of an object-oriented program as expressed by practitioners and applied the critical aspects derived from the analysis of practitioner interviews to the analysis of textbooks. The objective was to uncover ways of understanding that might assist the learning of object-oriented programming and to see whether these were emphasised in a range of textbooks.

7.1.1 Practitioner outcome spaces

Through the analysis of practitioner interviews, two outcome spaces that relate to the nature of an object-oriented program and to object-oriented program design were identified. The first outcome space relates to the nature of an object-oriented program (the 'what' aspect), and the second to the design characteristics (a 'how' aspect). The emphasis in the 'how' aspect is related to the primary influence on how an object-oriented program is designed.

1) The nature of an object-oriented program

The nature of an object-oriented program is the equivalent of the direct object of learning or the 'what' aspect. This is how the practitioners express their understanding of what they are attempting to produce as a result of writing an object-oriented program. The analysis resulted in five categories that are expressions of variations in three critical aspects (see Table 7.1).

A category represents an expression of understanding of the nature of an object-oriented program. The structural aspect describes the key characteristics of the category and the critical aspect descriptions describe the variations in those aspects that are held in simultaneous awareness. The higher level understandings build on lower levels and involve the simultaneous awareness of a greater number of aspects.

The detailed results are described in section 5.1.1 The nature of an object-oriented program, page 114.

2) Design characteristics of an object-oriented program

The design characteristics outcome space reflects the primary influences on 'how' a program is implemented. The focus in the current study is object-oriented programming but the practitioners showed a transition from emphasising object-oriented characteristics to characteristics that are independent of paradigm although they may be implemented within the object-oriented paradigm in specific ways. For the design characteristics outcome space, five categories were identified that are expressions of variations across four critical aspects (see Table 7.2).

Category	Referential	Structural	Critical aspects		
			Flow of control	**Object usage**	**Problem solution**
N1	Sequence of instructions	Recognises that all programs have a flow of control that defines the order in which program instructions are executed. The order of execution is seen primarily as sequential in nature	Flow of control is seen as sequential	Objects are used from provided framework	Provides required functionality
N2	Written using an object-oriented framework	Sees a program as object-oriented if it uses an object-oriented framework or using an object-oriented environment	Flow of control is still seen as sequential but is not a major focus	Programmer objects are defined and used but primarily as a requirement of the language or framework	Provides required functionality
N3	Based on data types	Sees an object-oriented program as being based on defining new data types or on data analysis or the development of a data-driven model	Flow of control is not explicitly discussed or considered an issue	Objects are defined based on data requirements	A data model of the problem space (real world model)
N4	Based on interacting entities	Sees an object-oriented program as a set of behavioural entities that interact to achieve the required objective	Flow of control is defined by the interactions between objects	Objects are defined based on behavioural requirements	A behaviour-driven model reflecting objects in the problem space
N5	Artificial construct	Sees a program as an artificial construct imposed by operating systems and not applicable to the discussion of object-oriented entities	Flow of control is defined by the interactions between objects	Objects are liberated from the confines of program boundaries	A behaviour-driven model using useful abstractions

Table 7.1: Categories of the nature of an object-oriented program

Table 7.2: Categories of design characteristics

Cat	Referential	Structural	Critical Aspects			
			Technology	**Principles**	**Cognitive process**	**Modelling**
D1	Is language dependent	An object-oriented design is dependent on the use of the class construct in an object-oriented language.	Constrained by language features	Language features determine principles applied	Need to know the language	
D2	Uses paradigm constructs	Uses specific object-oriented concepts and constructs	Constrained by paradigm constructs	Paradigm constructs determine principles applied	Thinking in terms of objects	Reflects real world objects
D3	Uses generic principles	Uses generic principles and concepts that have wider applicability	Implementation tool but paradigm specific	Paradigm independent principles drive design decisions	Thinking in terms of good coding practices	Reflects real world objects
D4	Applies generic design objectives	Uses design qualities that are general objectives for all programming	Implementation tool but some flexibility on paradigm	Design objectives drive design decisions	Thinking in terms of the impact of design decisions	Uses useful abstractions
D5	Expression of thought process	Sees the object-oriented program as being the result of a specific thought process that addressed design issues and programming problems	Implementation tool but select paradigm to match	Thought patterns implement design principles	Implicit recognition and correction of poor design	Communicate program design

The detailed results are described in section 5.1.3 The design characteristics of an object-oriented program, page 142.

7.1.2 Textbook analysis

The textbook analysis showed that the textbooks dealt with the identified critical aspects in different ways. The difference in the way that the critical aspects have been identified in their usage within the textbooks demonstrates the core difference between the two outcome spaces. The nature of an object-oriented program outcome space (the 'what' aspect) is difficult to ignore and all the critical aspects are given some emphasis in all of the texts. The design characteristics (the 'how' aspect) sees much more variation in the emphasis given to each of the critical aspects with the cognitive process critical aspect being given very little emphasis. A summary of the textbook analysis results is described in section 6.3 textbook analysis conclusion, page 222.

7.2 Implications for research on the nature of object-oriented programming

Sajaniemi and Kuittinen (2007), in reviewing the research on programming, made a call for research on the mental representations of object-oriented programmers. They contended that the notional machine for object-oriented appeared to be more complex and that the program designers were more problem domain focused. Much of the research on programming expertise has been conducted based on the foundation built by du Boulay (1989) and Pennington (1987a; 1987b). Du Boulay emphasised issues such as what programs are for and what they can do, the notional machine on which programs run, the language notation, algorithms, and skills or pragmatics. Pennington's framework included operations, flow of control, data flow, and function. Sajaniemi and Kuttinen (2007) in their call are recognising that some of the models and solutions coming from the nature of programming research seem to have difficulties when applied in the object-oriented context.

In the research into the naturalness of object-oriented programming, Wiedenbeck et al. (1999) used the sequential variation of flow of control as one of their measures of naturalness. Détienne (1997) also assumed that the nature of the flow of control does not change between procedural programming and object-oriented programming.

Détienne argued that the difference is seen in the way that objects bring together data and the algorithms that manipulate that data. The closest that either of these two studies come to a notion of interacting entities as described in the current study is the concept that the flow of control is distributed within an object-oriented program. The flow of control aspect from the current study is seen in their research and although they recognise object-oriented programming as being different from procedural programming, they do not recognise the variation in the flow of control. It would seem that for their studies the underlying assumption is that flow of control is sequential in nature for all programming.

The alternative to flow of control used in their research into object-oriented programming is data flow. Data flow relates to how data is transformed during processing (Corritore & Wiedenbeck, 1999; Garner et al., 2005; Robins, Rountree, & Rountree, 2001; Wiedenbeck et al., 1999). In Sajaniemi and colleagues' work on roles of variables, they do not explicitly refer to flow of control or data flow but both of these aspects underpin the role of a variable which they define as "stereotypic usages of variables that occur in programs over and over again" (Kuittinen & Sajaniemi, 2003, p 1). Sajaniemi (2002) is more explicit when he refers to a role being "characterized by the sequence of successive values of a variable and how it depends on other variables but not by the way the variable is used." This would relate closely to the notion of data flow. The concept of data flow was not visible from the analysis of the practitioner interview data in the current study, although the concepts of object state and changes in state were discussed in relationship to the behaviour of objects.

The sequence of instruction variation of the flow of control critical aspect identified in the current study is discussed in both the literature on programming paradigms and in research into novice learning of object-oriented programming. However, not all of the authors or researchers recognise the variations in the flow of control critical aspect as a sequence of instructions and as an algorithm implemented in the interactions between objects. The issue of data flow as an alternative to flow of control also needs further consideration. Since it was not uncovered in the current study, it may be an indication that data flow is not seen as critical to the understanding of object-oriented programming. The flow of control critical aspect appears to be supported by the research but the current study also challenges some of the research to look at flow of

control from a different perspective particularly when it comes to exploring the naturalness of object-oriented programming.

The way objects are used is not raised as an issue in the research on the naturalness of object-oriented programming (Burkhardt, Détienne, & Wiedenbeck, 2002; Corritore & Wiedenbeck, 1999, 2001; Détienne, 1997; Wiedenbeck et al., 1999). Other than reference to objects integrating data and functions (Corritore & Wiedenbeck, 1999; Détienne, 1997), there is no specific reference to the way that objects are used in the construction of an object-oriented program. It is unclear from this field of research what distinguishes the object-oriented paradigm from the procedural paradigm. Although the wider research on the naturalness of object-oriented programming examined data flow, it did not examine the state changes in objects. These were visible in the practitioner comments in the current study where state changes were linked to the behaviour of an object. Instead, the wider research discusses state in terms of changes in program state of specific variables (Wiedenbeck & Ramalingam, 1999, p 81). The understanding of objects as holding state which is modified through behaviour seems to have been ignored in the naturalness of object-oriented programming studies. The current study therefore questions the assumptions used in the naturalness of object-oriented programming studies with respect to examining programmer understanding of object-oriented programs.

With the nature of an object-oriented program outcome space from the current study, one of the identified critical aspects is flow of control but it is the change in understanding of the flow of control that identifies the shift from the lower level to the higher level categories. The second aspect of object usage is not reflected in the research literature on naturalness of object-oriented programming. There is an element of problem solution in the research literature but not in the sense of the program being a model of the problem domain or utilising useful abstractions in relation to behavioural requirements.

Some of this nature of programming research has focused on mental representations used by expert programmers (Burkhardt et al., 1997; Corritore & Wiedenbeck, 1999). The types of mental representations investigated were again based on Pennington's work (1987a; 1987b). Pennington argued for a program model and a domain model where the program model was assessed based on flow of control and operations and the

domain model based on data flow and functions. Burkhardt et al. (1997) adapted Pennington's models. These used a situation model which included data flow, static information about objects, and goals where goals related to aspects of the problem solution, and an adaption of Pennington's program model in which they examined the micro-level (control flow), and the macro-level (functions corresponding to units in the program structure). Corritore and Wiedenbeck (1999) used Pennington's model with adaption for function on a local level versus function on a global level.

It is difficult to see how the model critical aspect of the design characteristics outcome space in the current study relates to those used in the studies of mental representation. It is also unclear how the cognitive aspect relates since the emphasis in the mental representation research tends to be on features that correspond more with the nature of an object-oriented program outcome space as identified in the current study.

Control flow is an aspect that is considered as part of the research into the nature of object-oriented programming. The methods of assessing a programmer's understanding tend to focus on a sequential understanding of control flow. The results of the current study suggest that there is also a need to assess the understanding based on the interaction between objects. The data flow characteristic used in the nature of programming research was not explicitly mentioned in the current study although it may be partially present in the idea of object state. In this respect, data flow may be seen as reflecting an understanding of object usage. The final critical aspect of the nature of an object-oriented program identified in the current study is problem solution and although the nature of object-oriented programming research examines the domain model the methods do not relate to the way that the programmer understands objects as reflecting entities in the problem domain.

With respect to the design characteristics outcome space of the current study, the nature of object-oriented programming research seems to place minimal emphasis on these aspects.

7.3 Implications for conceptions of "object" and "class"

Eckerdal and Thuné's (2005) conceptions of objects and classes identified three categories. Their conceptions of objects do not talk explicitly about object usage but in

their lowest two conceptions of object, there is an emphasis on the way objects are seen within a program. At the lowest level in their outcome space, objects are experienced as a piece of code. This is partially the emphasis of the lower two levels of the object usage critical aspect in the current study, although the emphasis in the current study is more on the code as a user of existing objects. Eckerdal and Thuné's second category of an object being "experienced as something that is active in the program" is reflected in the understanding of a class in their study as providing a description of properties and behaviour. In their descriptions, there is no separation of objects with a data emphasis versus a focus on the behavioural aspects. Both appear to be seen as part of the defining characteristics for a class.

Eckerdal and Thuné's (2005) conceptions of objects and classes is drawn from those learning object-oriented programming and the emphasis on useful abstractions as identified in the current study is not represented. Their highest category recognises the relationship with a real-world phenomenon while their lower categories emphasise the technology. In this respect, their outcome space may be seen as a subset of the outcome spaces uncovered in the current study. There is also another major difference between the focus of their study and the focus of the current study that helps explain the difference in the outcome spaces. Eckerdal and Thuné focused on the object-oriented concepts of object and class while the current study focused on an object-oriented program, that is the context in which objects and classes are used. As a consequence, one might expect to see a wider range of issues raised in the current study while still incorporating the characteristics shown in their responses. The results of the current study do expand on the categories of description uncovered in their study with respect to the use of objects and in particular the aspect of the problem solution.

Participants in the current study talked about objects and classes. The responses, particularly to the nature of an object, are reflected in the object usage critical aspect of the nature of an object-oriented program. When talking about a class, the most frequent response for those who talked about object usage in the highest three levels of the object usage aspect was that a class was a template for creating objects. These participants did not link the class with the modelling of real world entities other than as a description for creating objects.

From the perspective of defining a paradigm, Stolin and Hazzan (2007) describe object-oriented as using objects as the building blocks but then simply refer to inheritance in relationship to their usage. There is no reference to the use of objects as abstract data types or as a method to encapsulating behaviour. Although their classification of building blocks and usage may help distinguish between the paradigms, it is not clear whether their attributes are adequate descriptors for each of the paradigms.

Diversity in the way objects are used is reflected in the literature with emphasis on data abstraction and behavioural abstraction. Less emphasis is placed on the idea of objects as entities freed from the constraints of programming boundaries. This may be in part due to the operating system environments that dominate current computing rather than the abilities of the technology.

Apart from the split between data type based objects and behavioural objects, the object usage critical aspect as identified in the current study seems to have had minimal focus within the literature. Although the current study identified the lower two levels, it did not investigate the reasons why this level of understanding was held. The highest level object usage as identified in the current study also needs further investigation since it does not appear to be represented within the literature.

7.4 Implications with respect to the "objects-first" debate

A question with respect to teaching objects-first is whether object-oriented programming is a new paradigm or an extension of the existing procedural paradigm. The literature on this subject had a range of different views but none were conclusive in their argument (see section 2.1.1, page 6). Lister et al. (2006a), in examining the issues in the object-first debate, highlight the diversity of the dialogue and the lack of an evidential basis for many of the arguments.

Reviewing the nature of object-oriented program outcome space from the current study, the two critical aspects that are reflected in this debate are the flow of control and object usage. Those who hold the view that object-oriented is an extension of procedural programming (Farrell, 2006a, 2006b; Hu, 2005, 2008; Vilner et al., 2007; Wirth, 2007) tend to see the flow of control as unchanged from procedural programming, and object-oriented programming as adding the ability to create abstract data types. The flow of

control remains sequential in nature. This view fits within the third category in the nature of an object-oriented program outcome space identified within the current study where the flow of control may be seen as sequential and the usage of objects is seen as based on data requirements. These authors may or may not see the solution in terms of being a model of the real world although frequently the emphasis is on finding the required types through finding the nouns used within the domain.

Berglund and Lister (2007) identified the same variation with respect to the flow of control in their analysis of the debate on teaching objects first. In their research, 'algorithms being implemented as object interactions' is one of the distinguishing characteristics for those who argue that object-oriented programming is conceptually different from procedural programming. They contend that those who see object-oriented programming as different from procedural programming saw objects as active entities with object interactions providing the algorithm and polymorphism being seen in the interaction between objects. It is unclear from their analysis whether these people also saw objects as being behavioural based although that may be the meaning behind active entities.

Taking a wider view of the literature related to object-oriented programming, object usage discussions tend to start at the third level of the nature of an object-oriented program outcome space as identified in the current study where objects are used as abstract data types (Coplien, 1992; Liskov & Guttag, 2000; Tucker & Noonan, 2007; van Roy & Haridi, 2004; Vilner et al., 2007; Wirth, 2007). The ability to define classes that encapsulate the data along with the behaviours for the manipulation of that data is seen as a core attribute of using an object-oriented language and as the basis for abstraction in object-oriented programming. Coplien (2000) contends that the organisation of systems around "abstractions based on commonality in structure and behaviour" is a core principle of the object-oriented paradigm. This reflects level three as described by the object usage critical aspect of the nature of an object-oriented program as identified in the current study.

Implementing data structures using objects could be argued as using objects to implement data abstractions but this is not the emphasis illustrated in examples by Nguyen and Wong (1998; 1999). In their examples, they utilise design patterns to enable a separation of the variable portion of algorithms from the structure and static

portions of algorithms. In later works, they focus more on using design patterns for the implementation of algorithms or the encapsulation of behaviour (Nguyen & Ricken, 2006; Nguyen et al., 2005; Nguyen & Wong, 2002a; Nguyen & Wong, 2002b). The perspective that objects are used for more than implementing abstract data types is also picked up in the design patterns literature. Gamma et al (1995), document a range of pattern types which they have classified as constructional, structural, and behavioural. They argue that design patterns do not follow an abstract data types approach but focus on the nature of the problem to be solved and provide appropriate patterns that can be adapted to create the required solutions. The implication of the design pattern classifications is that there are different reasons for using objects. Some design patterns may be seen as supporting an abstract data type approach to using objects while others are more in line with a behavioural understanding of object usage.

Part of the historical foundation for object-oriented programming is in the development of simulations (Nygaard, 1986). In his paper, Nygaard says that in Simula, '"classes" are named "activities" and "objects" named "processes"' (p 129). This naming convention shows a focus on behaviour in the thinking of the paradigm developers at least from a simulation perspective. This theme is also picked up by Wegner (1990) in reviewing the concepts and paradigms of object-oriented programming. Wegner places emphasis on the 'behaviour attributes' of the entities within an application domain but recognises that the representation of these as classes "allows behaviour to be managed and manipulated as though it were data" (p 82). This focus supports the two critical aspects of object usage and the focus on problem solution in terms of simulating 'entities within the problem domain as identified within the current study.' There is also a hint of a change in the flow of control in the sense that simulated objects interact but it is not a strong emphasis.

Another historical path is from the work on abstract data types (Liskov, 1988; Liskov & Zilles, 1974). Liskov and Zilles (1974) describe this as allowing the programmer to work with abstractions that are relevant to the problem domain, thus allowing the programmer to focus on solving problems relevant to the domain. They see these abstractions being represented as abstract data types that are defined by their characterizing operations. Although it is called an abstract data type, the behavioural characteristics define the type. This understanding does not place emphasis on the

interactions but rather on extending a structured programming language through these new abstractions. Again this is an emphasis on object usage and on the problem solution through modelling of domain entities as identified in the current study.

It would seem that the distinction based on object usage has historical roots as does the idea of modelling real world entities. Berglund and Lister's (2007) paper places more emphasis on the flow of control critical aspect. The nature of an object-oriented program outcome space identified in the current study provides a basis for discussing the issues that surround the objects first debate and the nature of an object-oriented program.

Much of the historical and current material on object-oriented programming would appear to fall into the top three categories of the nature of an object-oriented program outcome space as identified in the current study. This raises the question of where the lower two categories have their origin. The only possibilities found in the current study relate to introductory programming texts that use languages with object-oriented support frameworks but teach procedural programming (Bradley & Millspaugh, 2003). These texts have their origin in the teaching of procedural programming and although possibly adding a chapter on programming with objects are still primarily based in procedural programming. This same comment may apply to texts which introduce object-oriented programming techniques later in the text. In the case of the Bradley and Millspaugh's (2003) text, their utilisation of the Visual Studio's drag and drop facility further hides the nature of the higher level category understanding of object-oriented programming as identified in the current study. Further exploration of references related to the .NET framework and programming with Visual Studio may provide additional evidence to support these two lower categories and their variations in object usage and problem solution.

The two extremes of the nature of an object-oriented program outcome space would seem to indicate that there is a difference between procedural programming and object-oriented programming. This difference is supported by the analysis of the debate on teaching objects-first. However, this outcome space may also provide a base for further dialogue on these issues.

7.5 Relationship to learning to program

The problem solution critical aspect reflects the understanding of the objective of programming. Booth (1992) conducted a phenomenographic study of learners who were learning a functional programming language and a procedural programming language. Her outcome space on the nature of programming also contains an emphasis on the nature of the problem solution. Although the lower two categories in the problem solution critical aspect of the current study are stated as providing the required functionality, the structure of the solution as a sequence of instructions reflects an understanding of the way that a computer operates and requires instructions. The required functionality is implemented in a way that is understood and required by the computer or is a computer oriented activity (Booth, 1992, p 95). As the focus moves to a solution that models reality within the problem solution critical aspect of the current study, the focus is moving toward programming as a problem oriented activity. For Booth, the emphasis on a problem oriented activity is on solving the problem rather than on the computer itself (p 94). What is not so clear in the current study's outcome space is the relationship to a product oriented activity. This may be reflected in the principles critical aspect of the design characteristics outcome space where the emphasis moves to a focus on maintenance and reuse. It may also be behind the focus on a model but it is not made clear through the focus on the nature of an object-oriented program as opposed to Booth's focus on programming. Although there may be parallels between Booth's "conception of the nature of programming" and the design characteristics outcome space of the current study, there is essentially a difference in focus in terms of the analysis. It may be that a researcher with a similar focus to Booth rather than the more technical focus of the current researcher may come to similar conclusions to Booth from the data obtained in the current study.

Booth's research also includes a "conceptions of the nature of programming languages" outcome space. Although this was not a focus of the current study, it is interesting to see that in her outcome space, the second level (programming languages as code) reflects the technology focus that is seen in the current study in the technology critical aspect of the design characteristics outcome space. The top two levels in Booth's conceptions of the nature of programming languages emphasise a means of communication and a medium of expression that is reflected in the design characteristics outcome space in the

cognitive process critical aspects top levels of the current study. In both Booth's results and in the results of the current study the communication is seen as not just being with the computer but with other programmers and the ultimate users of the program. Again, the difference in focus of the analysis may account for the different outcome spaces developed in the two studies.

Booth's final outcome space relates to conceptions of learning to program (p 119). The data in the current study was not examined from this perspective but there are parallels between her outcome space and the outcome spaces in the current study. Booth's categories of "learning a programming language" and "learning to write programs in a programming language" are reflected in the technology critical aspect of the design characteristics outcome space of the current study. Booth's "solving problems in the form of a program" is reflected in the problem solution aspect of the nature of an object-oriented program outcome space and the model critical aspect of the design characteristics outcome space of the current study. "Becoming part of the programming community" from Booth's outcome space may lie behind the emphasis on design objectives and cognitive process categories of the design characteristics outcome space of the current study. There is a focus in the principles critical aspect of the current study that reflects a desire for others to be able to understand and maintain the code base created. Again the focus in analysis between the two studies is quite different and it may be possible that a researcher focusing on the data from the same perspective as Booth may draw similar conclusions to those of Booth's study.

Although there are marked differences between Booth's outcome spaces and those produced by the current study, there are some areas where Booth's study is reflected in the outcome spaces of the current study. The current study neither acts as a confirmation of Booth's study nor as a contradiction, rather, the two studies reflect the difference that occurs when data is approached from different perspectives.

Eckerdal and Berglund's (2005) phenomenographic study on how students understood what learning to program meant reflects an emphasis on the thought processes required. Their outcomes space contained five categories. The lowest level of these is stated as "Learning to program is experienced as to understand some programming language, and to use it for the writing program texts" (p 141). This is similar to the knowledge of the programming language variation at the lowest level of the cognitive process critical

aspect of the design characteristics outcome space identified in the current study. Their second level emphasises the learning of a way of thinking that aligns with the programming language and acknowledges that is experienced as ‘difficult to capture’. The alignment with the programming language may be an indication that the programming language embodies constructs that reflect a programming paradigm however, this is not explicitly identified. It is difficult to tell whether this carries the same meaning as the thinking in terms of objects variation of the cognitive process critical aspect of the current study but there are similarities in the emphasis. The fourth category of their study with its emphasis on “learning a way of thinking which enables problem solving” and is a “method” of thinking has parallels with the highest variation of cognitive process category of the design characteristics outcome space identified in the current study.

Eckerdal and Berglund’s third category of gaining an “understanding of computer programs as they appear in everyday life” and their fifth category of “learning a skill that can be used outside the programming course” have no parallels in either outcome space of the current study.

There are differences in the terminology used and this may reflect the difference in focus of the current study. Both their study and the current study were completed in the context of object-oriented programming but Eckerdal and Berglund’s focus was on ‘programming thinking’ and in this sense was exploring the cognitive process aspect where for the current study, the cognitive process arose from an analysis of the data relating to how the practitioners understood the characteristics of an object-oriented program. The current study placed no emphasis on ‘programming thinking’ however it was a characteristic that became evident from analysis of the interviews.

7.6 Programming as a multi-programming paradigm activity

The relationship to other programming paradigms was not considered a focus of the current study but frequently the more experienced practitioners mentioned advantages of other paradigms. In the analysis, it became clear that underlying the design characteristics outcome space is a shift from a focus on a specific language or programming paradigm to an ability to think through the design issues to build an

application. This shift is not present in the nature of an object-oriented program outcome space. Awareness of this shift in emphasis meant that the literature on programming paradigms was revisited to identify the aspects dealt with in this context.

The issue of flow of control and data flow is discussed in the context of being the distinction between programming paradigms (Floyd, 1979, p 456). Floyd argues that some paradigms favour data flow while others favour flow of control. Other writers on programming paradigms also raise the issue of flow of control as a defining aspect (Ambler et al., 1992; Stolin & Hazzan, 2007; Tucker & Noonan, 2007; van Roy & Haridi, 2004). Ambler et al. (1992) with their classification of the paradigms as either operational or definitional, see data flow as a particular paradigm category but flow of control is discussed in reference to operational paradigms and as being defined by the rules engine for declarative paradigms. In the case of Stolin and Hazzan (2007), this is implicit in their building blocks aspect and the relationships between building blocks. Procedural programming is defined from the perspective of building blocks as using procedures or sequences of commands. For the functional programming paradigm, the building blocks are simply referred to as functions and as classes for the object-oriented paradigm. For these later two paradigms, the flow of control issue is not discussed. Tucker and Noonan (2007) reference flow of control with respect to the procedural paradigm but do not refer to it in relation to other paradigms. Van Roy and Haridi (2004) talk less of flow of control but place emphasis on the control abstractions used by the programming paradigms.

None of these authors specifically discuss the paradigms in terms of design characteristics, although Brookshear (2007) does discuss design independently of the discussion of the programming paradigms. Van Roy and Haridi (2004) talk of the paradigms as a way of reasoning about programming. This would seem to reinforce the idea that there are design objectives which are not programming paradigm specific and there are thought processes that, while being influenced by the programming paradigms, may encompass all paradigms enabling greater flexibility in the production of program solutions.

The notion of exploring the nature of an object-oriented program in a multi-programming paradigm context was not considered for the current study and may limit the applicability of the results.

Exploring the tools in use by programmers developing web applications reveals an interesting mix of tools. With a tool such as Fit (Mugridge & Cunningham, 2005), the developer creates a set of tables that provide a set of inputs and expected outputs. With a testing framework such as JUnit (Beck, 2003) or NUnit (Newkirk & Vorontsov, 2004), the programmer writes tests in a declarative manner so that the testing framework can determine how to run the tests against the application. Many of the supporting build tools utilise declarative techniques (Ferguson Smart, 2008). Even programming languages such as Ruby (D. Thomas, 2005) support multiple programming paradigms and when combined with application frameworks such as Rails (D. Thomas & Hansson, 2005) use programming conventions that are multi-paradigm in nature. Languages for scripting such as JavaScript provide a procedural base for the production of program segments that utilise the resources of the containing environment (i.e. a web browser) (Loui, 2008). As well as using an object-based framework, such as would be reflected in the second level of the nature of an object-oriented program outcome space identified in the current study, to access the containing environment, these languages may provide facilities for the programmer to create their own objects (Morrison, 2007). The combination of languages in use in projects may include a main programming language such as Java but also incorporate XML or other language technologies drawn from a different programming paradigm.

Although the design characteristics outcome space from the current study has the higher level of thought process, the current study did not explore or seek an understanding of what programs are becoming within the current software development context. So although there is a move in the outcome space toward paradigm independent concepts, design qualities and thought processes, there still needs to be further investigation to uncover the way that practitioners working in this environment think about the nature of an object-oriented program or system and how they understand design within this broader context.

7.7 Relationship to model-driven approaches to software development

The lower two levels of the nature of an object-oriented program outcome space from the current study reflect a problem solving approach to programming with respect to the

problem solution critical aspect. This variation of the problem solution aspect is reflected in a number of introductory programming texts (Lewis & Loftus, 2006; Malik, 2006; Morelli & Walde, 2006). There is no clarity in the texts with respect to the nature of the solution other than it provides the required functionality. At the higher levels in the current study's outcome space, the emphasis is placed on the program being a model that represents the solution. At level three, this model is based on the data requirements of the problem space. This perspective is reflected in the domain-driven modelling (Evans, 2003) and model driven architecture (Kleppe et al., 2003; Mellor et al., 2003; Miller & Mukerji, 2001, 2003) approaches to software development. The starting point in these approaches is the data requirements of the domain but neither approach is restricted to a pure data-driven model. Both approaches implement behavioural objects to provide the services for the application domain.

It is this emphasis on identifying objects in the problem domain that leads researchers to explore the naturalness of object-oriented programming. Their view is represented by the statement: "The claims about program comprehension are based on the belief that OO programs more closely reflect the real-world problem with which they deal, thereby facilitating comprehension" (Corritore & Wiedenbeck, 1999). This same idea is picked up when arguing proximity to the real world by the argument that "When using object-orientation the structure of the software system mirrors that portion of the real world that the software is meant to facilitate" (Sprague & Schahczenski, 2002).

Détienne (1997) claims that Meyer contends that "the world can be naturally structured in terms of objects" (p 49). Meyer (1997) does emphasis that objects can be directly picked from physical reality and modelled in software but Meyer's emphasis is on the way that software addresses the needs of the problem domain through the development of a model that describes that domain. As such, the model becomes "a clear correspondence (mapping) between the structure of the solution, as provided by the software, and the structure of the problem, as described by the model" (p 47). For Meyer, the language becomes a way of thinking about the problem domain. This idea is picked up by Quatrani who contends that "Visual modelling is a way of thinking about problems using models organized around real-world ideas" (2000, p 3).

Although some may argue that this means that object-oriented is a "more natural way" of programming and therefore easier to think at higher levels of abstraction (Neubauer

& Strong, 2002), the real emphasis is on the software being a model and tool for communication with others in the development team. As a model, it incorporates entities from the problem domain. Evans (2003) contends that the model becomes a ubiquitous language for discussing the problems and the nature of the solution. As such it provides a tool for communicating between the customer, those for whom the software is being written, and the software developer. It helps ensure that both understand what is expected of the system. The value of the model is in its ability to communicate and to represent the knowledge acquired from the domain (Armour, 2000a, 2003b; Robillard, 1999).

One view of the model with respect to model driven architecture is that the model is a higher level of abstraction (Mellor, Scott, Uhl, & Weise, 2004). It moves the developer another level away from the requirements of the computer and the programming languages. It is argued that by using the model and a model compiler, it should be possible to work with entities that reflect the problem domain and are therefore easier to reuse. With an emphasis on developing a platform independent model and using transformation tools to obtain platform specific models and the code, there is an increase in productivity, portability, and interoperability (Kleppe et al., 2003). The platform independent model (PIM) is a higher level of abstraction than the code and is therefore easier to maintain. For model driven architecture (MDA), the "model is a description of (part of) a system written in a well-defined language" (Kleppe et al., 2003, p 16). It is an abstraction of something that exists in reality (p 15) and it can be used as the basis for generating code. With respect to the current study, this view of a model is reflected in the nature of an object-oriented program outcome space's problem solution critical aspect and in the model critical aspect of the design characteristics outcome space. There seems to be some agreement between object-oriented programmers and model driven architects that what is being produced is an abstraction of reality. The emphasis in model driven architecture on reuse and maintenance is also reflected in the principles critical aspect of the design characteristics outcome space.

There is a strong emphasis on the relationship with the real world represented in the third and fourth levels of the problem solution critical aspect. This emphasis has both data and behavioural aspects although the distinction in the literature is not as clear as that of the practitioners in the current study. The reading of the literature, although

emphasising the relationship with the problem domain, also allows the freedom of expression of ideas that leads to the use of useful abstractions rather than a simple correlation. This later emphasis is reflected in the view that an object may represent either a real-world entity or a conceptual entity (Quatrani, 2000, p 43).

The nature of the model in terms of its approximation to reality or as an abstraction that addresses issues within the problem domain also reflects the concepts being discussed in respect to abstraction and its role in computer science (Kramer, 2007). The degree to which an object in the model reflects the key characteristics of the real world phenomenon is an assessment of a level of abstraction within Hazzan's relationship between the object of thought and the thinking person (Hazzan, 2002). The more a solution resembles the problem domain, the less abstract is the solution. Yet, even when useful abstractions are used in a solution, they need to carry some meaning that is of value to the participants in the development project. There is a balance between providing an abstract solution that appears to have little resemblance to the problem domain and using useful abstractions that enhance the understanding of the problem domain and the ability to produce meaningful solutions.

A model built on data abstractions of real world entities may require less ability to abstract from reality than a solution built on behavioural requirements. The ability to see how a behavioural abstraction might be created that appears to have no direct equivalent in reality requires a higher level of abstraction than identifying the data characteristics required for an entity.

7.8 Levels of concepts

In relation to the current study, within the design characteristics outcome space, variations have been identified that relate to technology and design principles. Both of these aspects utilise concepts and terminology that can be analysed to understand the variations in understanding of these concepts. Sufficient analysis of these concepts was performed to identify some of the issues that arise but the utilised concepts need further investigation.

Understanding the variations in the concepts and terminology that are contained within the technology and design principle critical aspects is essential for planning a teaching

strategy based on the variations in these critical aspects. Planning the presentation of variations in these critical aspects for teaching requires being able to use appropriate variations related to the underlying concepts that comprise these critical aspects.

Some of the themes for which variations were noted in the interview data include:

a. the relationship between type, class, interface, inheritance, and reuse
b. the understanding of encapsulation
c. the understanding of abstraction

These issues were also identified in the analysis of the textbooks especially the issues surrounding type and classes.

7.8.1 Type, inheritance and reuse

Some of the specific issues surrounding the understanding of type are influenced by the issues raised by static typing versus dynamic typing. Some interviewees talked about type inheritance when inheriting from a base class and a number of the textbooks talked about inheritance in terms of it being an IS-A relationship. The difficulty is that classes also incorporate implementation (code) that forms part of a code reuse strategy. Reuse is not necessarily dependent on type. The concept of type hierarchies and code reuse at times appear to be in conflict. This seems to be particularly true with respect to statically typed languages such as Java. One possible proposal to solve this problem is to utilise Java interfaces to define types and to use inheritance for inheriting implementation (VanDrunen, 2006). An interface defines the behaviours expected of a type but includes no implementation.

This issue is not a problem in dynamically typed languages where the type is determined by the messages that an object responds to. For a language like Ruby, this is referred to as duck typing in the sense that if the object quacks like a duck and waddles like a duck then it must be a duck (D. Thomas, 2005). Ruby, Smalltalk, and other dynamically typed languages still use classes but primarily as a mechanism for inheriting implementation.

In contrast to these languages are the languages that allow the creation of objects through cloning (Black, Hutchinson, Jul, & Levy, 2007; Ungar & Smith, 2007). Some

interviewees argued that these languages were not object-oriented, while the thinking of others on object-oriented programming had clearly been influenced by such languages.

Although these issues surrounding the understanding of type, inheritance, and code reuse were noted in the current study, further investigation is required in order to fully understand the differences in the ways that these concepts are experienced by practitioners and are communicated to novice programmers through programming texts. In planning a teaching strategy, there also needs to be clarity in terms of the variations presented to learners.

7.8.2 Encapsulation

In the analysis, this concept was discussed as being used in two distinct ways. The first was in relation to the object-oriented paradigm with the meaning of grouping together data and the methods that manipulated that data within a class. This usage was also prevalent in the textbooks (J. Anderson & Franceschi, 2005; Barker, 2005; Becker, 2007; Farrell, 2006a, 2006b; Lewis & Loftus, 2006; Morelli & Walde, 2006). The second usage of encapsulation recognises that any concealing of the details is a form of encapsulation. In this context, procedures can be thought of as encapsulating processing logic (Shalloway & Trott, 2005; Stein, 2003). Stein links encapsulation with abstraction techniques including procedural and data abstraction.

A form of encapsulation used within the electronics community is the integrated circuit. The integrated circuit is a component that performs a specified function based on a set of defined inputs and produces a specified set of outputs. One of the participants in this study suggested that the programming community should borrow from the integrated circuit design approach and define collar interfaces for objects with the objects encapsulating the functionality required. The collar interface defines both the expected inputs and outputs for the object and the dependencies that the object has on other objects. As an encapsulated object based on a collar interface the object could be reused in another context more easily and freed from the constraints of the program context in which it was developed.

Even from a brief and cursory analysis of the materials, it is evident that there are different ways of explaining or comprehending what encapsulation means within a

programming context. Although the interview data obtained from the current study does have some of these variations present, the exploration of the understanding of encapsulation was not a focus of the current study.

7.8.3 Abstraction

Abstraction is a term used frequently in relation to programming and computer science. The term was also used by participants in this study and with distinctly different ways of expressing their experience of it. Kramer (2007) has argued that there are different variations or levels of performance with respect to abstraction, and Hazzan (2002; 2003; 2008) has explored how students endeavour to reduce the level of abstraction. Bennedsen and Caspersen (2006; 2008) have endeavoured to assess the level of abstraction as an indicator of success in learning Computer Science concepts.

In some contexts authors qualify their use of the term abstraction linking it with procedural abstraction when seeking to talk about implementing behaviour in a method and as a consequence giving it a name, and with data abstraction when seeking to talk about creating new data types (van Roy & Haridi, 2004). Liskov and Guttag (2000) define a third type of abstraction as iteration abstraction which relates to iterating over objects in a collection. Van Roy and Haridi (2004) would possibly classify this as a control abstraction since it introduces a control structure for working with the logic to process a collection. Van Roy and Haridi talk of creating linguistic abstractions when new language constructs are introduced that extend the language to simplify forms of processing. These authors are being specific about their use of abstraction in relation to the different programming techniques and issues.

A more generic use of the term abstraction is referred to in the context of modelling when talking of a model being an abstraction of the real world (Barker, 2005; Evans, 2003). In this context, abstraction is used in an all encompassing manner to refer to eliminating unnecessary detail but in the process of modelling, data and procedural abstractions may be utilised.

No attempt was made to analyse the way that the term abstraction was used in the current study other than to determine whether the participant was talking in a paradigm specific or generic sense. In most cases, abstraction was treated as a generic concept or

principle within the design characteristics outcome space. Like other generic concepts such as type, discussed above, the way that the participants talked about abstraction needs further investigation and a more complete study of this theme needs to be completed. For the current study, concepts were grouped together based on their primary emphasis; that is, they were grouped as paradigm concepts if that was their primary usage and as generic concepts if they tended to be used free of the constraints of the paradigm.

7.9 Implications in relation to textbooks

The analysis of textbooks performed in the current study is based on the outcome spaces from the interview analysis. It also has in mind the principles of variation theory (Marton & Tsui, 2003) and identifying the space of learning enacted in the text with respect to the critical aspects uncovered in the interview studies. The task of identifying the space of learning proved more difficult than anticipated especially in relationship to the design characteristics critical aspects. An analysis aimed at uncovering the space of learning may have been better started with a blank slate rather than a set of already identified critical aspects.

Previous analyses of programming textbooks have endeavoured to determine whether specific computer science themes were covered and what emphasis was placed on those themes (Czerwinski, 1985; Means, 1988; Ross, 2005; VanDrunen, 2006). Means (1988) conducted a content analysis of textbooks to determine whether the content had changed over a ten year period and to identify trends in the content of textbooks. The analysis was performed using content headings and index topics and was based on the number of pages assigned to each theme. Means compared different editions of the same text to determine the extent to which the coverage had changed. There is no judgement made on the effectiveness of the presentation. Czerwinski (1985) endeavoured to make a judgement on coverage and the effectiveness of that coverage of specific computer science themes drawn from the Computer Science Curriculum 1978 (Austing, Barnes, Bonnette, Engel, & Stokes, 1979). The method of identifying coverage was based on analysis of the author's preface, table of contents, and key words in the index. Sections on the themes as identified were read in depth to gauge depth of cover. There appears to be no metrics identified for assessing the depth of cover.

Lin and Wu (2007) reviewed thirty two textbooks used in Taiwan over a four year period. They identified common problems with the textbooks based on responses from users. The problems identified included a bias towards Windows / Intel products, too much emphasis on software application tools, too little emphasis of computer science concepts, too many technical terms, meaningless analogies and examples, lack of supplementary materials, and inadequate treatment of programming-related content. There seems to be no identification of what the desirable computer science concepts should be or the emphasis that the users felt should be present although there is a reference to a curriculum that parallels the ACM model curriculum of 1993.

Ross (2005) performed an analysis of the emphasis placed on polymorphism by 10 texts on object-oriented programming. The texts used Java, C++ or VB.Net as their teaching language. The analysis looked at the number of pages used to cover these themes in relation to the total size of the texts and the types of explanations used to describe these techniques by the authors. Ross regards polymorphism as one of three core techniques for object-oriented programming and is concerned that there may be a decline of emphasis on the topic. It is unfortunate that Ross' study makes no conclusion to support the hypothesis that there is a decline of emphasis on polymorphism. For the current study, polymorphism is included as one of the themes relevant to paradigm concepts.

VanDrunen (2006) also follows a theme closely related to polymorphism and that is the use of the Java Interface as the foundation for analysis of textbooks. VanDrunen contends that Java interfaces enable subtype polymorphism and that class based inheritance provides for code reuse. He further contends that Gamma et al. (1995) argue "that in an object-oriented setting, programming should be done to an interface, not to an implementation, that class inheritance is a mechanism for code reuse rather than for subtyping and polymorphism, and that even composition should be favoured over class inheritance" (p 875). VanDrunen argues that Java interfaces should therefore be addressed ahead of inheritance but in examining twenty seven textbooks, his study shows only five introduce interfaces ahead of sub-classing (inheritance), and three of these introduce it in the context of using polymorphism.

None of these textbook analyses are focused on the space of learning opened up by the text or explored the variations used in teaching the concepts under examination. The themes explored in these studies, possibly with the exception of Lin and Wu (2007), are

relevant to the current study although their conclusions are difficult to compare with the current study's results. VanDrunen's argument with respect to interfaces and polymorphism in relation to reuse and inheritance, however, is an issue that was seen as confused by the current study's participants and also an issue in the analysis of textbooks with respect to the emphasis on paradigm concepts and the generic principle of reuse. This difficulty has been noted elsewhere in this thesis.

What all of these studies show is that there is a wide variation in the emphasis between textbooks, in terms of the amount of content dedicated to a particular theme and the placement of the theme relative to other themes. The current study reveals that there are other variations in focus of textbooks with respect to the understanding of the nature of an object-oriented program and the emphasis given to different design characteristics. The theme covered with the greatest consistency appears to be in relation to the flow of control critical aspect where all except two of the texts placed some emphasis on an object-oriented program being comprised of interacting entities. However, there are questions with respect to just how consistently that theme is supported throughout the textbooks with many textbooks having examples showing a sequential flow of control while the text emphasised that the requesting of object behaviour through calls on class methods really represented message passing and interaction between objects. During the analysis, the researcher struggled with how an example would illustrate the object interactions and concluded that none of the texts provided a convincing argument, although Stein (2003) came close with the comparison between the making of a peanut butter sandwich and the operation of a restaurant. Others used the restaurant theme to promote interacting entities but did not contrast it with any alternative or did not carry it through to program code. This difficulty in representing interacting entities is an issue requiring further investigation.

There were two difficulties in investigating the space of learning opened up by the textbooks. These were the limited use of multiple examples to illustrate concepts or themes and the tendency to work to a single solution to the programming problem. The textbooks tended to focus on defining a concept and then illustrating that concept with an example before setting exercises for the learner to complete. There were few cases where alternative solutions to the same problem were explored in order to draw the learner to a position of either contrasting the technique being learnt with other possible

approaches or drawing out the full meaning of the technique or concept through variations of application. The notable exceptions to this were Steins (2003) contrasting procedural coding with interacting entities, Seirra and Bates (2005) dialog on the differences between an object-oriented solution and a procedural solution to a programming problem, Bergin et al.'s (2005) discussion of the use of an if statement versus the use of polymorphism, and Jeffries (2004) willingness to explore false paths before coming to a solution with which he was more comfortable. There seems to be greater comfort in working toward single solutions or presenting desired coding qualities without providing contrasts that would help open up the space of learning.

When planning the current study, the researcher was thinking in terms of variations in object-oriented concepts emphasised and planning an approach to teaching these concepts, and it appeared as though emphasis should be placed on a range of object-oriented concepts versus a range of types of problems addressed by those concepts. Although this model is fairly restrictive in its application, it would appear that textbook writers do need to spend more time thinking about variations of problem situations for the concepts being presented or variations in ways of illustrating concepts to ensure that the appropriate space of learning is opened up for the concept. The current analysis placed a lot of emphasis on what the authors said rather than the illustrations or application situations presented by the authors. The current study has been unable to identify the space of learning opened up by the textbooks and this was primarily caused by the lack of variations used in presenting concepts and themes by the textbook authors. This may in part be caused by a focus on the quantity of material to be covered in the textbooks rather than the quality of presentation. Here Sierra and Bates (2005) with their emphasis on keeping the attention of the brain through the use of different presentation techniques possibly come closest to consistently using variations to open up a space of learning.

The final issue in the analysis of textbooks relates to the decision to focus on Java textbooks. The exceptions were the Jeffries (2004) and Wiener (2007) texts that used C#, and the Brookshear (2007) text that introduced multiple paradigms and languages. A second text by Farrell (2006b) was analysed alongside her Java text (Farrell, 2006a) but these two books were so similar in content, even to the point of using exactly the same wording, that the language independent text really provided no contrast. Although

some of the texts recommended particular programming environments (Barnes and Kölling (2006) – BlueJ, Guzdial and Ericson (2005) – Dr Java, Felleisen et al. (2005) – Professor J), most texts endeavoured to be programming environment neutral. Jeffries (2004) text is based on the use of Microsoft®'s Visual Studio but his emphasis on using test-driven development lessens the reference to the programming environment. In contrast, introductory texts on other .Net languages place more emphasis on the drag and drop features of the development environment (Bradley & Millspaugh, 2000, 2003) and may lead to an understanding of object-oriented programming as being sequential in nature and using objects as provided by the supported frameworks. Another development environment that may promote this style of thinking is Jade (Clarke, 2000; Post, 2000) where the emphasis is not on drag and drop but the integrated database and framework. This means that there is a strong emphasis on a particular style of program design. Broadening the study to include a wider range of texts may help reveal the difference in emphasis caused by the programming language and the development environment.

Variations in understanding caused by the use of different environments or object-oriented languages by the participants in the interviews clearly showed up in the variations in the ways that they experienced object-oriented programming. This was most noticeable where the participants had used languages that used object prototyping and not classes (i.e. Self (Ungar & Smith, 1987, 2007)) and also those who had used other paradigms (i.e. functional programming). Other interviewees with background in .Net languages or Jade also presented different variations but it was difficult in one of these cases to determine whether it was the limited experience of object-oriented programming or the environment that caused the particular views to be expressed. It would seem that the analysis of textbooks should have included some of the texts based on other languages and environments, and that there may be scope for exploring the influence of programming environment and language on the expressed understandings of participants.

7.10 Conforming to phenomenographic outcome spaces

From the current study the linkage between the design characteristics ('how') and the nature of an object-oriented program ('what') appears to be the notion of a model. In

this context, the nature of an object-oriented program ('what') is describing the characteristics of the resulting model that might be the result of applying the design characteristics ('how').

7.10.1 The relationship between 'how' and 'what'

Marton and Booth (1997) presented the 'how' and the 'what' aspects in a manner whereby these aspects can be represented through different outcome spaces with respect to learning. However, Marton, Dall'Alba, and Beaty (1993), in presenting their findings on the conceptions of learning, closely linked the 'how' and the 'what' aspects within one outcome space. In the current study, the nature of an object-oriented program has been described as a 'what' aspect and the design characteristics as a 'how' aspect of object-oriented programming. However, both outcome spaces include a reference to model or modelling as part of their critical aspects.

7.10.2 Model as link between the two outcome spaces

The outcome spaces recognise that there is an element of the object-oriented programming community who may not refer to models at all. In the current study, a third of the participants made no reference to there being a model or to the process of programming as being an act of modelling. This also applied to the textbooks where just under half made no reference to modelling. Yet the current study appears to show that modelling or model is a link between the two outcome spaces and is present in the upper categories of both outcome spaces.

One participant, in emphasising the role of modelling, did contrast the difference as that of focusing on modelling the real world versus managing source code. His contention was that the predominant approach in Scandinavian countries is to focus on programming as being the modelling of the real world. He emphasised this by saying:

> "In all the Scandinavians' view, in Norway and Denmark especially, programming is modelling. There is no distinction. Programming first and foremost is creating a model of the world and object-orientation is the tool that you have to model the world. And almost as a side effect, almost by accident that model is executable. But the purpose of object-orientation is to

> allow you to create a good model of your problem domain and then you can execute it" (I20)

In his approach, programming and modelling become synonymous. The program is created by building a model of the problem domain and that model is executable (i.e. a program).

He contrasts this view with what he calls the American approach which he saw as focusing on managing the source code. He expressed this idea as:

> "The motivation of the American school is quite different where object-orientation is seen as the tool to manage source code and they approach this from right at the other end where the thing is you have these large amounts of source code and it becomes too complex. […] At the educators' symposium, Robert Martin said that as well. He said separation of concerns of encapsulation is the defining factor of object-orientation because it is all about how I have this source code and it's all a big mess. There are too many dependencies. I can't maintain it. I can't modify it easily. The purpose of object-orientation is to cut up my source code and make my source code manageable" (I20)

Although this participant contends that the modelling perspective is a Scandinavian perspective, the outcome spaces arrived at in the current study show that behind even the source code maintenance perspective there is some awareness of the concept of building a model. Those who were focused on design qualities and thought processes emphasised the "program as a model" aspect. The separation would appear to be less distinct and may be more of a focus with respect to the approach to programming.

With the concept of a "program as a model" being a critical aspect of the nature of an object-oriented program outcome space, and modelling appearing as part of the problem solution aspect of the design characteristics outcome space, this seems to provide a linkage between the 'how' and the 'what' aspects of object-oriented programming.

The real world nature of a model or the "program as model" based on useful abstractions provides a 'what' aspect to the design characteristics outcome space that is primarily a focus on the design of an object-oriented program (a 'how' aspect). This

does not directly correspond with the data-driven versus behaviour-driven model focus of the nature of an object-oriented program outcome space although these are linked with real world data or behavioural requirements, thus providing a linkage between the two outcome spaces. But the model emphasis in the nature of an object-oriented program outcome space carries with it a ‘how’ aspect. In seeing the model as being data or behaviour driven, the participants are describing an approach to arriving at the model. It should be noted that the current study did not attempt to explore the ‘what’ differences between a data-driven model and a behaviour-driven model. The end result of these approaches to modelling may arrive at very similar solutions.

Marton and Booth (1997) talk about the ‘how’ and the ‘what’ aspects being coupled and this seems to be reflected in the results of this study. Although there are two outcome spaces, there is a common linkage that draws those outcome spaces together. That linkage seems to draw the top three levels of the nature of an object-oriented program outcome space and align them with all but the lowest level of the design characteristics outcome space. The lowest level of the design characteristics outcome space would appear to align with the lowest two levels of the nature of an object-oriented program outcome space. Accordingly, someone who is focused on knowing and using language constructs (design characteristics) is more likely to see an object-oriented program in terms of language features or in terms of using framework objects.

This linkage was not a specific focus of the current study but became apparent as the critical aspects were identified for each of the outcome spaces. Further research and analysis would be required to explore how the other aspects of the nature of an object-oriented program impact the design characteristics thinking that influences the ‘how’ of programming.

7.10.3 Relationships between critical aspects

In examining the nature of an object-oriented program outcome space and the critical aspects, there is an increasing integration of the different critical aspects at the higher category levels. An understanding that flow of control is sequential in nature can be held independent of the awareness of objects within the programming environment. With the move to interacting entities, there is awareness not only of the existence of objects but also of the relationship between those objects to establish a flow of control

within the program. The problem solution critical aspect also reflects a move from a perspective where objects are not necessarily relevant for a problem solution to a perspective where the objects are the very building blocks for arriving at a problem solution, and the nature of the way those objects are defined is the foundation for constructing a solution. For the nature of an object-oriented program outcome space, there is an increasing simultaneous awareness of the critical aspects and the way that they fit together to arrive at a program as a solution to the programming problem.

This increasing simultaneous awareness of the critical aspects is also present in the design characteristics outcome space, although in this outcome space there is a transitioning of the focus on different aspects with each category. However, it is not possible to operate at the expression of thought process category without being aware of the technology or the principles that are the core foundations of that thought process. The interview data shows this integration with sections that reflect a cognitive process often also containing a technology or a principle or an aspect of modelling. This integration is not so obvious at lower levels where the focus is more on technology or technology and principles.

This same simultaneous awareness of the design characteristic critical aspects is present in the textbooks where authors, when discussing a paradigm construct, relate these to language constructs. Even in Farrell's (2006b) generic object-oriented programming text, pseudo code that reflects the Java language is used to illustrate concepts and constructs. This also occurs in Brookshear's (2007) text where he is endeavouring to provide a broad foundation of computer science principles. Although Barker (2005) with her focus on modelling through abstraction initially lays the foundation without reference to the technology, it isn't long before the two are linked and talked about simultaneously. Wiener (2007) makes this link between model and technology from the very beginning.

Jeffries (2004) with his emphasis on the thought processes that he uses clearly links those thought processes with the use of the technology both in the form of the language and the paradigm constructs. The cognitive processes are not independent of the technology and principles utilised in the construction of programs. Jeffries' cognitive process also reflects a change in focus with respect to the technology in that instead of the technology being the focus of what is being learnt, it becomes the object of study.

Jeffries explores the technology constructs in a way where he is critically assessing their value and usefulness for the task. This transition reflects the subject of study to object of study that Kegan (2000) refers to as a "constructive-developmental theory". He says that this is "the gradual process by which what was "subject" in our knowing becomes "object". When a way of knowing moved from a place where we are "had by it" (captive of it) to a place where we "have it", and can be in relationship to it, the form of our knowing has become more complex, more expansive" (pp 53-54).

Reviewing the two outcome spaces in the context of the relationship between aspects and in terms of the possible transition in focus, particularly within the design characteristics outcome space, appears to confirm that the outcome spaces reflect a deeper understanding of the phenomenon at the higher category levels.

7.11 Implications for teaching

In proposing the current study, it was argued that uncovering the variations in the ways that practitioners experience object-oriented programming and the critical aspects that characterised those variations would enable teaching to be planned that fostered an appropriate level of understanding through utilising variation in those critical aspects to open up the space of learning. The analysis of the textbooks identified that the critical aspects are currently used by the textbooks, although at different levels within the identified outcome spaces. There is also uncertainty about whether appropriate variations are used to enact the desired learning space.

Wirth (2002) argued for "a succinct introduction into the basic notions of program design" and the use of "a concise, formal notation". The current study did not attempt to identify "a concise formal notation" but it did identify the critical aspects of flow of control and object usage within the nature of an object-oriented program outcome space. At the base of the design characteristics outcome space is the language. A technology foundation would appear to be required in order to learn programming. Wirth argues that this should be kept to a reasonable size and focus on the core concepts. Based on the notation, Wirth contends that the basic concepts should be taught. He identified these basic concepts as iteration, recursion, assertion and invariant. Wirth's view is that object-oriented programming adds abstract data types to procedural programming

(2007) and therefore the same basic concepts should form the foundation for teaching object-oriented programming as are required for procedural programming.

The current study does not directly challenge that thinking, although Wirth makes no reference to object interactions. Iteration and recursion still remain core concepts. Iteration is still required to process collections of objects but with the use of an iterator, iteration is implemented in object interactions rather than a logic construct such as a 'for' loop. Recursion can be used in the form of functional recursion or structural recursion (K. B. Bruce, Danyluk, & Murtagh, 2005; Goldwasser & Letscher, 2007). Structural recursion involves interaction between objects, the calling of the same function but on another object of the same type or related type, while functional recursion (Booth, 1993) emphasises self reference or repeatedly calling the same function. The difference between these two forms of recursion emphasises the variations in the flow of control critical aspect. Within the design characteristics outcome space, these variations can be used to illustrate the different approaches used within the paradigms at the technology level. Assertion and invariant still also apply within the object-oriented context, but the way that they are applied may differ.

Using variations in the application of constructs was used by Thompson (1992) as the foundation for the teaching strategy used for procedural programming. Students learnt the concepts of iteration and selection by applying these concepts to a range of different programming problems. The course utilised structure diagrams, flow charts, and program code to illustrate each construct. Different forms of iteration and selection constructs were used to solve the same problems so that the students were introduced to variations in the solutions and could learn to evaluate the effectiveness and ability of each construct to communicate the design.

Extending this thinking to the outcomes spaces uncovered by the current study, the same form of object interaction should be used for a range of programming problems to foster an understanding of how object interactions can be used to solve programming problems. Variations in object interactions, and the contrasting object-interactions with the use of logic constructs, for the same problem should be used to further reinforce an understanding of the way that objects and the interaction between objects can be used to build program solutions. Bergin et al. (2005) use different applications of polymorphism to emphasise the underlying principles and later discuss the use of an 'if' statement to

implement ad-hoc polymorphism to further deepen that understanding. However, they do not use the two constructs to solve the same problem. Stein (2003) discusses the use of conditionals and polymorphism as a form of conditional dispatch. She illustrates the distinction by working from an implementation based on ‘if’ constructs slowly refactoring the solution until she has a solution utilising polymorphism. The objective of this sequence is to introduce polymorphism, and to show how, through utilising polymorphic constructs, the code structure can be made more succinct thus improving the design. The same examples provide variations that draw out simultaneous awareness of object interactions, object usage, and design objectives. The effectiveness of the variations in problems and construct usage depends on how clearly they open up the desired space of learning and reinforce the desired conceptual outcome (Marton & Tsui, 2003).

Wirth’s (2002) next criterion relates to the structuring of statements and typing of data. The use of data abstractions is part of the object usage aspect and, although Wirth’s own view is toward the use of data abstractions, the emphasis in the nature of an object-oriented program outcome space would appear to relate to the use of objects as the building bocks of a program. The emphasis in Wirth’s first four criteria appear to focus on the nature of an object-oriented program, although it could also be argued as focusing on the technology aspect of the design outcome space.

Wirth’s final criterion has an emphasis on information hiding, modularization, and interface design. This appears to be a shift toward generic concepts and design objectives. Wirth’s set of concepts is quite small compared with the range of concepts mentioned by the practitioners in the current study.

Wirth’s emphasis is on there being a concise core set of notations and terms that lay the foundation for programming. This same emphasis may be seen in van Roy and Haridi (2004) where they focus on core concepts that can be used to build a kernel language that supports multiple programming paradigms. This core set of notations form a learning space focused on the technology with some emphasis on generic principles and design characteristics. Wirth makes no reference to fostering cognitive processes. The current study does not contradict the view that the emphasis should be on a core set of concepts, although it challenges what those core concepts are and the interpretation of

those concepts and it places more emphasis on the way that variations are used to illustrate those core concepts.

This emphasis on a small number of core concepts is also present in the role of variables approach to teaching introductory programming (Byckling & Sajaniemi, 2005; Kuittinen & Sajaniemi, 2004; Sajaniemi, 2002). Sajaniemi and colleagues have identified and used a small range of roles that variables play within programs and used these to teach introductory programming. By focusing on the roles, they have reduced the complexity of programming and been able to examine those roles in different contexts. However, it is difficult to see how the roles of variables would foster either the understanding of a program as interacting entities or the variations in object use. The generic principles and design objectives only become a focus in the sense that the role patterns are implementations of the principles and design objectives. The research literature on expertise supports the idea of using such patterns or plans to introduce core concepts since expert programmers appear to utilise such patterns and plans in their programming (Beck, 1997, 2007; Gamma et al., 1995; Soloway, 1986).

Ramsden's relational learning model (2003) and Biggs' 3P model (J. B. Biggs & Moore, 1993) both emphasise the learner's representation of the task as influencing their approach to the task and the consequential learning outcome. From the perspective of phenomenography, learning is about changing the learner's conception of the phenomenon (Marton & Booth, 1997). The strategy for teaching should therefore aim to challenge the conceptions of the phenomenon held by the learner by providing alternative experiences of the phenomenon that promote the desired conception of the critical aspects for that phenomenon.

The outcome spaces of the current study suggest that the phenomena of interest in programming are the program and the criteria for assessing the design of the program. With respect to the nature of an object-oriented program, the program is composed of interacting entities. These entities are objects that have been identified through data or behavioural requirements of the problem domain or problem solution. When assembled into a program these objects form a model that is a useful abstraction of the problem domain. Ideally, those objects should be able to be freed from the constraints of a program as enforced by operating systems and reused in other contexts.

Helping the learner understand a sequential flow of control and the need for data and algorithm can be achieved through linking with the learner's prior experience of the concept of a program (Thompson, 1992). Some examples that have been used for this are in Appendix 1. These examples draw on phenomenon that surround the learner and with which the learner is familiar. By asking appropriate questions about the common characteristics of these phenomena, it is possible to build a conception of a program as a sequential flow of control that utilises data. In using these examples, the focus is on moving the learner's thinking from focusing on the content or message of the program to exploring the structure and underlying concepts that cause these phenomena to be recognised as programs. The variation in the form of the programs, while focusing on the commonalities, helps open up the space of learning with respect to the nature of an object-oriented program.

A similar approach can be used to foster the idea of interaction of entities and the role that these entities play in encapsulating data and behaviour. Stein (2003) and Guzdial and Ericson (2005) used the example of a restaurant. Andrianoff and Levine (2002) used role plays to portray the behaviour of objects. Their role plays present abstract situations rather than situations familiar to the learner. By extending Andrianoff and Levine's examples, it is possible to generate a range of role plays that portray the interaction of objects and the data and behaviour that these objects provide. Like the procedural program examples, more than a single example needs to be provided. The operation of a restaurant and the limited role plays are not the only things surrounding the learner that rely on interactions and object usage. Others include sports teams, business operations, and shopping. In many cases the learners may not recognise the interactions. The learners need help to see the interactions and entities in the familiar things that surround them on a daily basis. They may also need to see how these interact to form an algorithm to achieve desired results and how there is a combination of behaviour and data in those interactions.

Just as the concept of a sequential program surrounded the learners on a daily basis and they failed to recognise the common features of those programs and needed to learn to look at the common characteristics in those sequential programs, the concept of interacting entities surrounds us on a daily basis. The world is made up of such interactions, but these interacting entities are not seen in the context of being a program.

The learner has to make a shift from being part of the process or interactions, which is being captured by the process, to being able to study the interactions and see the components that make them up. It is like the difference between being a participant in a sports team and being the coach planning the strategies and revising the game plans. The participant is involved in the play, reacting to things as they are happening. The coach looks at the overall play and recognises the patterns of play and the game plans that are effective and ineffective. The coach seeks to foster the analytical thinking of the game into the players so that they can respond to the things that are happening in the field of play. This is a shift very similar to the subject to object shift of transformative learning (Kegan, 2000).

The shift to the modelling aspect involves a recognition that the situation being examined can be represented in a form that captures the interactions and object usage in an abstract form. In order to study strategies for a team sport, the coach can utilise diagrams and objects. The learner of object-oriented programming can learn that a program represents a similar abstraction of the problem domain in a form that the computer can execute to arrive at a solution to the problem. Barker argues that this is the natural way that people think about the world (2005, p xl) but the research would contend that this is not the natural way for people to think about programs (Wiedenbeck et al., 1999). The learner needs to be helped to make this connection and to learn to think in terms of the abstract correspondence between objects in the program code and entities in the real world.

The discussion so far has focused on the nature of an object-oriented program outcome space or laying the conceptual foundation for approaching programming and primarily on the technology critical aspect of the design characteristics outcome space. Learning programming would appear to require more than an understanding of the technologies and the nature of a program. There needs to be a balance between laying the technology foundation in order to actually write programs and developing an understanding of the principles and cognitive processes necessary to develop program designs. With the outcome spaces, the emphasis is on increasing simultaneous awareness and not simply an emphasis on each of the critical aspects. To attain the deeper levels of understanding within the design characteristics outcome space, the learner needs to become aware of the different aspects simultaneously. They need to understand the technology in light of

the principles, cognitive processes, and modelling; it is not an either or but rather a combination. The method of opening up the space of learning has to encourage simultaneous awareness while possibly emphasising specific aspects.

A number of the textbooks emphasised the problem solving process (Jeffries, 2002; Langr, 2005; Lewis & Loftus, 2006; Malik, 2006; Morelli & Walde, 2006; Soroka, 2006). With the exception of Langr and Jeffries, most introduced the process in early chapters and either never revisited it or they provided worked solutions at the end of the chapters. For Langr and Jeffries, the problem solving process or approach to writing the code was integrated into the very fabric of the text. For Jeffries, this included the false starts and experiments taken in trying to understand the possible solutions to programming problems.

The procedural programming course (Thompson, 1992) also introduced a problem solving approach at the beginning of the course but during the course as programming problems were approached in class, the lecturer consistently tried to model the process. This approach to teaching was also described by one of the participants in the current study when he says:

> "Sometimes we write it just in class. Sometimes I show them code that I've prepped in advance, I do a lot of that but a big part of what I do is to modify that. So I may have a program and I make sure that I am prepared to do the editing in class, they might participate as a class but we actually do it live" (I 8)

This approach to teaching goes beyond telling. It encourages the learners to participate in the cognitive processes that surround the use of the constructs and the design of the program. The cognitive processes of software development demand an understanding of the principles. To apply the principles demands an understanding of the technology and how the technology can be used to achieve those design objectives. Variations in the use of the technologies can be used not only to focus the learner on the technology concepts but also to bring awareness of the design objectives. Variations with respect to the application of design objectives can be used to foster awareness of the cognitive processes. Variations in one aspect can be used to help stimulate awareness of other aspects within the design characteristics outcome space.

Lewis and Loftus describe the first step of problem solving as understanding the problem (2006, 43). One approach to showing an understanding of the problem is to write tests. Langr describes a test as “specifying what the code needs to do” (2005, p 31). To write a test before coding means to understand the required behaviour of the code in terms of how it is expected to behave when provided with a given set of inputs. The cognitive process involved in deciding what tests to use becomes a foundational part of understanding the requirements for the program. Teaching a test-driven process becomes integrated into the fabric of learning a problem solving approach.

Bergin et al.’s (2005) contrasting of the ’if’ construct with polymorphism not only helps provide an understanding of the language constructs, it also provides an opportunity to discuss design principles and in particular the different localities of maintenance that result from the use of conditionals in contrast to the use of polymorphism (I14). This could be easily extended to the cognitive processes that allow the programmer to make the decision as to which construct to use. Stein’s (2003) refactoring from a solution implemented using conditional constructs to a polymorphism solution also provides an opportunity to explore design principles and the cognitive processes that help identify when to use polymorphism in contrast to conditional constructs. Barker (2005) uses a sequence for developing a calculator to illustrate presentation layer programming. In the process, she makes some design decisions to remove duplication. These provide opportunities not only to explore the technology constructs and their influence on design, but also to discuss the design principles that make one solution preferable to another. The difficulty is that this step to design principles or the cognitive processes has not been explained in the texts. The authors are focused on the technology and mention the design principle almost as an aside. Other texts present a technology and mention that it provides a design advantage but since they provide no contrasting solution, the learner can not see or be helped to see what that design advantage really is.

Providing variations with respect to a critical aspect provides the opportunity to open up the space of learning. Recognising which variations will achieve the desired impact on the space of learning can be identified from the outcome spaces. The current study has been unable to plan a teaching strategy or test that strategy in the classroom but this discussion illustrates the possibility for this to be achieved.

7.12 Conclusion

Although the current study identified a number of issues that relate to current research in relation to object-oriented programming and to the teaching of object-oriented programming, the hypothesis that drove the original research remains unanswered. It is possible to discuss the possible strategies for teaching based on the outcome spaces but due to factors outside the researcher's control, it was not possible to implement the teaching strategy or gather data that would show that it was causing the desired conceptual change in the learner. This study therefore leaves a number of questions to be addressed in follow up studies.

The current study focussed on object-oriented programming but the design characteristics outcome space suggests that a multi-programming paradigm approach to teaching programming may be desirable. This raises additional questions about the nature of a program in a multi-programming paradigm context. The question of the nature of a program was never addressed in the current study although there may be some clues from the work of Stolin and Hazzan (2007) that parallel the nature of an object-oriented program outcome space that could be used for further investigation.

The current study has classified concepts and principles into broader categories and not sought to analyse the expressions of understanding of these paradigm concepts and design principles. Further exploration of these concepts would enhance the range of variations opened up for planning teaching.

Chapter 8. Charting a path forward

The hypothesis behind the present research is that if the critical aspects for the categories of description for a phenomenon as recognised by practitioners in the field can be identified, then these can be used as the basis for planning teaching. This hypothesis draws on the ideas of Ramsden's learning in context model (2003) where the learner's perception of the task requirements influences their approach to learning and consequentially the learning outcomes. The practitioners, through their experience in the field and the variations in how they perceive the phenomenon with which they are working, potentially lay the foundation for a revised approach to teaching programming.

The original plan for this research involved the identification of practitioner perceptions of an object-oriented program using a phenomenographic approach followed by the assessment of a teaching strategy based on the identified critical aspects. This was revised to the reported research of verifying the applicability of the identified perceptions against textbooks aimed at teaching object-oriented programming.

8.1 Findings

This research has identified two outcome spaces of the ways that practitioners have expressed their understanding of an object-oriented program and the influences on their approach to designing programs. It has also identified that there is wide variation of how these critical aspects are covered in existing programming texts.

In examining the outcome spaces in relation to the literature, a number of specific areas were highlighted. These are:

1) The two extremes of the nature of an object-oriented program outcome space indicate that there is a difference between procedural programming and object-oriented programming.
2) The object usage critical aspect of the nature of an object-oriented program outcome space indicates that there is greater distinction in the variations than reported in the current literature. The literature distinguishes between objects seen as data abstraction and as encapsulating behavioural abstractions, but does not refer to objects that are liberated from the confines of a program boundary.

3) The useful abstraction variation is present in the practitioner texts in relation to the modelling of object-oriented systems and model-driven architecture, but is not identified in the research literature which has focused on learner perceptions of object-oriented concepts.
4) There appears to be difficulty in illustrating the concept of interacting entities within the context of programming textbooks. There seems to be scope for further investigation of the flow of control critical aspect with respect to interacting entities to uncover the variations in presentation that would open up a space of learning around this aspect.
5) There were limited variations of examples and application of concepts in the textbooks analysed. The focus appears to be on the quantity of material (i.e. the range of concepts and ideas presented) rather than on ensuring that the space of learning is opened up for the learner.
6) The model and modelling provide a link between the 'how' and 'what' of object-oriented programming. This needs further investigation to clarify the nature of the link and the influence of the other critical aspects in the outcome spaces.

8.2 Issues:

The current study raises a number of issues that need further investigation. These include:

1) If the design characteristics and industry practice are increasingly emphasising multi-paradigm program thinking, then what is the nature of a program in this multi-paradigm context? Are there critical aspects missing from the current analysis because of the focus on the object-oriented paradigm?
2) Although the issues surrounding the understanding of type, inheritance, and code reuse were noted in the current study, further investigation is required in order to fully understand the differences in the ways that these concepts are experienced by practitioners and communicated to novice programmers through programming texts.

3) The levels of understanding or usage of the terms encapsulation and abstraction need to be explored in the context of object-oriented programming and also in their usage in all forms of programming.
4) The idea of an object being liberated from the confines of a program was poorly supported by the literature. Further investigation of this variation of object usage is needed in order to understand the implications with respect to teaching and software development.
5) For the original hypothesis of this research, the biggest issue that remains outstanding is whether a teaching program focused on opening up a space of learning based on the variations in the critical aspects will achieve the desired level of understanding.

8.3 Recommendations

The current research has identified critical aspects that relate to the way that practitioners express their awareness of object-oriented programming. The original hypothesis for this research aimed to improve learning of object-oriented programming. In this section, recommendations are made based on the findings and their application in teaching and the writing of textbooks.

8.3.1 For academics

The core recommendation to academics involved in the teaching of object-oriented programming is to explore the use of variations focussed on the critical aspects that help expose the desired level of awareness of the critical aspects and to implement assessment strategies that help determine whether this level of awareness is being obtained. A proposed research strategy for achieving the evaluation of the teaching strategy is described later.

Although it is easy to see the levels of awareness as stages in development and to contend, particularly with the design characteristics hierarchy, that a technology foundation needs to be laid before design principles and cognitive processes, the current research is neither supportive nor unsupportive of this approach. However, the basis of the levels of awareness is the increasing of simultaneous awareness of the critical aspects. To obtain simultaneous awareness means that the critical aspects and the

relationships between them need to be introduced in parallel. Endeavouring to separate the critical aspects reduces the possibility of simultaneous awareness.

In order to bring simultaneous awareness of the critical aspects, the researcher would use two different strategies. These would be:

1. to provide alternative solutions to programming problems that contrasted the different variations in awareness of the critical aspects, and
2. to utilise a number of different problems that applied the critical aspect in a similar way.

For any critical aspect within the outcome space, there is the ability to provide variations in terms of the approach taken to solve a particular problem and to provide variations in terms of the types of problems that can be solved with the particular variation of the critical aspect. The first of these two strategies highlights the variations within the critical aspect while the second strategy provides variations of the application of the critical aspect variation. The second of these strategies was more visible in the textbooks examined in the current study.

With respect to the first strategy, to highlight the nature of an object-oriented program, solutions that used sequential flow of control would be contrasted with solutions that used interacting entities (i.e. conditional constructs verses the use of polymorphism, or loop constructs verses functional and structural recursion). The interacting entities would be based on solutions that used abstract data types and solutions that used behaviour-based entities. The entities would reflect both real world entities and abstract entities. The differences and similarities could then be discussed to bring simultaneous awareness of the different critical aspects.

A similar strategy would be used for the design characteristics critical aspects. Solutions would show variations in technology usage, design principles, and cognitive thought processes. The variations in thought processes would be illustrated by working through the process of development highlighting the reasoning for the design decisions. These thought processes would reflect the decisions made as a result of applying the design principles and the utilisation of different technology solutions.

Although language constructs, paradigm concepts, and design principles need to be introduced individually, the focus in their introduction is on providing variations in usage and providing contrasts with already learnt material.

The emphasis in planning the teaching strategy is on how to highlight the variations and to bring simultaneous awareness of the critical aspects from each of the hierarchies. The same solution set would be used to bring simultaneous awareness of the different critical aspects.

8.3.2 For textbook authors

The analysis of the textbooks used in this research was based on the critical aspects uncovered in the analysis of the practitioner interviews. None of the textbooks examined were written based on the results of such an analysis nor did they utilise variation theory in their writing. It should also be recognised that the analysis performed did not aim to analyse the textbooks to discover their particular focus or space of learning although that was visible in some contexts. For example, Barker (2005) clearly opened up issues around object-oriented modelling. Because this study did not analyse these textbooks looking for the space of learning opened up by the text, it is not possible to assess how effectively they achieved their objective. Any recommendation made from the current research has to be seen in the light of the understanding of learning promoted by phenomenography and the principles of variation theory used for the analysis of teaching.

The core recommendation is that textbook authors endeavour to utilise the variations discussed in the teaching strategy above. The focus should be on highlighting the critical aspects of learning to program as highlighted in this research and on variations that open up a space of learning to enable the desired level of understanding. This may mean reducing the amount of material introduced in the text while ensuring that the critical aspects become visible.

8.4 Future research

There are two potential problems with the current outcome spaces. The first is whether they have a more generic application to programming across paradigms. The second is

that there are a number of underlying concepts that require further investigation. This is particularly true of the design characteristics outcome space. Although the data gathered in this research sheds some light on these concepts, a number of these concepts require further research and exploration.

For these outcome spaces to be of any value, they need to be applied to their intended context; the teaching of programming in a tertiary environment. This leads to the identification of further questions that need to be explored. These are:

1. How do the outcome spaces identified in this study apply to programming in other paradigms?
2. How are the underlying concepts related to each of the outcome spaces understood by practitioners?
3. How do these understandings impact the way that these practitioners approach the programming task?
4. How can a course be structured to foster development and change in awareness of the "critical aspects"?
5. How do variations in tertiary student awareness of the critical aspects relate to their effectiveness in writing object-oriented programs?
6. How does tertiary student awareness of the critical aspects change during learning of object-oriented programming especially when the critical aspects are part of the focus of teaching?

The first question requires investigations on the nature of program in other paradigms. Questions two and three can be addressed through similar studies to this one. Rather than starting with the nature of an object-oriented program, they would start with one of the concepts and explore the understanding that practitioners have of that concept. If these further studies are to have applicability to the results of the current study, there may be a requirement to be able to link back to the outcome spaces and critical aspects or to verify the results of the current study against the findings for the concepts.

8.4.1 Teaching Strategy Evaluation

Questions four through six above were originally planned to be addressed in the current study but the changing organisational circumstances and the size of the project meant

that it would have been difficult to complete in the time available. The original intention was to develop a teaching strategy based on the variations in the critical aspects as outlined above and then to apply this strategy in a teaching environment that utilised classroom research techniques (Angelo & Cross, 1993; Cross, 1998; Cross & Steadman, 1996) to determine whether the appropriate level of awareness was being achieved. This strategy was presented as part of the ethics application for the current study.

The emphasis in this teaching and research strategy is on the scholarship of teaching and learning (Boyer, 1990; Cross, 1998; Glassick, 2000; Glassick, Huber, & Maeroff, 1997; Lister, 2008). The focus is on being able to assess what is happening in the classroom and to uncover ways of improving the opportunities for learning. In seeking to apply the results of this study to teaching, there needs to be a program of evaluation that determines the effectiveness of the teaching strategy and seeks for continuous improvement.

Core components of the planned research strategy are the use of naturally occurring course data, classroom learning exercises, and a learning journal.

1) Naturally Occurring Course Data

Naturally occurring course data can provide a rich source of data for uncovering student understanding of the topic other than through the grades assigned. The BRACELet research has shown that in applying alternative analysis strategies to data collected in the normal exam process, it is possible to uncover issues that relate to student difficulties in understanding the subject matter (Clear, Edwards, Lister, Simon, Thompson *et al.*, 2008; Lister, 2008; Lister et al., 2006b). Assessment focuses on assigning a grade to the student while research focuses on learning something about the student that will aid in future teaching cycles. The research is less interested in individual achievement and more interested in uncovering the issues that might hinder the desired level of understanding.

2) Classroom Learning Exercise

Classroom learning exercises involve the whole class completing a short open question related to what is being learned (Angelo & Cross, 1993). The nature of the questions should encourage free thinking about the concepts. Classroom exercises provide

feedback to the teacher with respect to the learning that is occurring and the problems that students are having. The exercises take a short period of time at the end of a lecture to complete. Such data can be gathered either anonymously or with some form of identification that allows the results to be matched with other gathered data. This strategy was used in the teaching strategy for the course that was originally to be used for the current research.

The approach used and planned involved the responses being gathered in by the researcher. The researcher would complete an analysis to produce descriptions that relate to variations in awareness of the concepts and provide feedback to the next class session. The students would discuss the variations in awareness in focus groups. A summary of these discussions would be collected for later analysis. Issues in understanding can also be addressed in the lecture.

3) Learning Journals

Learning journals provide a mechanism for students to reflect on their learning and to identify issues with their understanding. Students can be encouraged to write in their journal at specific times and also encouraged to use it for general reflection on their learning. Specific prompt questions can be used to encourage them to write on specific themes related to what they are learning. These can include writing entries describing their critical moments which caused a change in view or in understanding the value of a critical aspect.

8.4.2 Looking beyond learning to program

This research also raises questions in terms of innovation in computing environments and directions in development. The concept of a program as an artificial construct raises issues about the way that computer systems are structured and used. There are areas for further research in freeing the user and the programmer from the program centric way of thinking about software development and to look for new paradigms that are more flexible and maybe more user friendly. This thinking is reflected in the recent “future of the browser” ideas from Mozilla and Adaptive Path (Beard, 2008; Garrett, 2008). Although their thinking is in a browser context, it reflects a style of thinking that removes the boundaries to enable more flexible interaction.

In the design characteristics outcome space, the critical aspect of the cognitive process needs further examination. Although this critical aspect was identified, the characteristics that foster the desired cognitive processes were not identified. One of the participants in this study placed emphasis on the removal of duplication as being at the core of this cognitive process and of the writing of quality code. Others expressed a concern that the cognitive processes used within development approaches such as extreme programming were poorly identified and difficult to communicate.

8.5 Conclusion

This research sought to uncover the way in which practitioners expressed their understanding of object-oriented programming and the critical aspects that comprise the differences in those variations. These were then compared with textbooks and uncovered the variability in the way that these critical aspects were presented.

The original intent of this research included a desire to apply the critical aspects to the planning of a teaching strategy and to verify the effectiveness of that strategy. This was not achieved and remains an issue for further research. The proposed strategy for achieving this objective has been outlined in this chapter.

Appendix 1 What is a program(me)?

Overview

Introduction

The purpose of this Worksheet is to learn:

the definitions of at least 5 key terms

at least 4 uses of programming techniques as related to computing

at least 4 reasons for having programming principles

what is and is not a programme.

To accomplish this task, we will take a look at a number of programmes that we may use on a regular basis. Some of these are not computer programs but, they do have the many of the same characteristics.

In This Worksheet

This Worksheet contains the following Topics.

Examples Of Programmes

Introduction

The Programmes we will look at are:

a TV programme

part of The Phantom of the Opera programme

part of the instructions for building a model aircraft

a recipe for a cake

payroll calculations instructions

instructions for a walk around the college campus.

Instructions

Review these programmes and list what are the key elements which you believe make these programmes.

TV Programme

The list below details a TV Programme.

6.00 Network News

6.30 Holmes: Current Affairs

7.00 Coronation Street: Drama

8.00 The Bill: Police Drama

8.35 Casualty: Hospital drama.

9.35 Foreign Correspondent

10.40 One Network News

10.50 Mini-Series: The Billionaire Boy's Club

12.45 Closedown

Your List Of Key Elements

Continued on next page

Examples Of Programmes, Continued

The Phantom Of The Opera

Below is detailed Part of the Phantom of the Opera Program.

PROLOGUE

The stage of the Paris Opera, 1905
Auctioneer, Raoul and Company

OVERTURE

ACT ONE - PARIS 1861

SCENE 1 - The dress rehearsal of "Hannibal"
Think of Me..... *Carlotta, Christine and Raoul*

SCENE 2 - After the Gala
Angel of Music ..... *Christine and Meg*

SCENE 3 - Christine's dressing room
Little Lotte.../The Mirror...(Angel of Music)
Raoul, Christine and Phantom

SCENE 4 - The labyrinth underground
The Phantom of the Opera .....*Phantom and Christine*

SCENE 5 - Beyond the lake
The Music of the Night.....*Phantom*

SCENE 6 - Beyond the lake, the next morning
I Remember../Strange Than You Dream It...
Christine and Phantom

SCENE 7 - Backstage
Magical Lasso.....*Buquet, Meg, Madame Giry and Ballet Girls*

SCENE 8 - The manager's office
Notes../Prima Donna..... *Firmin, André, Raoul, Carlotta, Giry, Meg and Phantom*

SCENE 9 - A performance of "Il Muto"
Poor Fool, Makes Me Laugh ..... *Carlotta and Company*

Your List Of Key Elements

Continued on next page

Examples Of Programmes, Continued

Instructions For Building A Model Aeroplane

A list of instructions for building a model aeroplane **FUSELAGE** is detailed below.

1.Build a left hand and right hand side as follows:

(a)Take one fuselage side and cement the sheet doublers in place on a 45E angle as shown.

(b)Build **opposite** side in the same manner.

2.When dry, trim off the excess doubler sheeting and epoxy formers 3 & 4 to one side, gluing the wing supports in place at the same time to give correct location. Ensure the formers are held square whilst drying.

3.When dry, fit the other wing support and the other fuselage side to the formers and epoxy in place, once again maintaining the formers and sides square.

4.Leaving the diecut skid in its parent sheet, cut the tow hook wire supplied in half and bend up two tow hooks (one left bend and one right bend as shown in the detail). The marks on the ply sheet show the shape and location of hooks. Drill the two holes for the tow hook wire where shown, and drill a series of fine holes (or push pin through instead of drilling) for the binding later. When satisfied with the fit, cut the tow hook area as shown in the detail and remove the skid from its sheet. Fit the hooks, bind together tightly with heavy cotton or fuse wire and epoxy both hooks in place - leave to dry.

5.Mark the centre of the formers 3 & 4, and fit the plywood skid in position, allowing 1.5mm at former 4 to match the bottom sheet later. (The back two hooks are used for normal launching, the front ply hook is only used for advanced flying or high wind conditions). Sand skid and put aside.

6.Sit the fuselage on a flat surface, and fit the diecut front bottom half sheets in position, using a piece of scrap plywood to space the centre slot. Pull the front fuselage pieces together and fit and epoxy in place the square ply former 1. (Masking tape around the nose of the fuselage is an ideal clamp while drying). When dry, glue the front bottom half sheets in position, but **not** the scrap ply. Glue scrap corner gussets in position at the front former.

Your List Of Key Elements

Continued on next page

Examples Of Programmes, Continued

Recipe For A Cake

A list of instructions for making a Farm Sultana Cake is detailed below.

Ingredients:

1 cup sultanas

2 cup milk

lb butter

3 cups of flour

2 cups sugar

2 teaspoons baking powder

5 eggs

Instructions:

Half cover sultanas with water and bring to the boil. Turn off heat and allow to absorb water and cool.

Cream butter and sugar, beat well. Add eggs one at a time, add milk, flour, baking powder and lastly sultanas.

Cook in a large meat dish in moderate (350E) oven for one hour.

Can add peel, cherries and nuts.

Your List Of Key Elements

Continued on next page

Examples Of Programmes, Continued

Payroll Calculations

A list of instructions to pay an employee is detailed below.

Pay Calculation

```
    If hours worked is greater than 40.0 then
        wages = (40.0 H pay rate) +
                (hours worked - 40.0) H 1.5 H pay rate
    otherwise
        wages = hours worked H pay rate
```

Main Algorithm

```
    Prepare to write a list of the employees' wages
    Set the total payroll to zero
Prompt the user for the employee number (put a message on the screen)
    read the employee number
As long as the employee number is not 0, repeat the following steps:
    Prompt the user for the employee's hourly pay rate
        Read the pay rate
        Prompt the user for the number of hours worked
        Read the number of hours worked
        Perform the procedure for calculating pay (above)
        Add the employee's wages to the total payroll
    Write the employee number, pay rate, hours worked, and wages on
            the list
        Prompt the user for the employee number
        Read the employee number
When an employee number equal to 0 is read, continue with the
        following steps:
    Write the total company payroll on the screen
    Stop
```

Your List Of Key Elements

Continued on next page

Examples Of Programmes, Continued

Instructions For A Walk Around Campus

These instructions should allow you to explore the campus. This will be of practical benefit to those who have not been around the campus long. The purpose of the exercise however is to consider how well the instructions can be followed and to identify any issues that you need to consider when constructing instructions.

1) Leave the classroom and cross the courtyard to the path between V36 and V37.

2) Follow the path to the first covered way on your right. There is a building at the end of this covered way leading off to your right, what services are in this building? (Do not go down this covered way.)

3) Continue straight ahead until you come to a path on your left. Turn left.

4) At the end of this path turn left along the road and the right so that you continue along the road.

5) At the far end of the last building on your left, turn left. What is this building used for?

6) Turn left again at the far end of this building. Stop at the covered walk way. What building is in front of you and slightly to your left?

7) Turn right. As you walk along the covered walk way, what is on your right?

8) At the tee intersection of the covered ways, turn right and stop just inside the doors of the building. What services are located in this area?

9) Turn around. Retrace your steps to the tee intersection of the covered ways.

Your List Of Key Elements

Examples Of Non-Programmes

Introduction

The following examples are not programmes. Review these examples to check that your above list is complete. The examples are:

a Table of Contents

a Sporting Events for a Carnival

a recipe for Sherbet.

Instructions

Record your reasons why each of these examples is not a programme.

A Table Of Contents

The list below details the **PREFACE** ***xii*** of a Table of Contents.

Unit I Introduction to Data Processing

Your List Of Key Elements

Continued on next page

Examples Of Non-Programmes, Continued

Sporting Events

The list below details Sporting Events For Carnival.

100 Metres Sprint
200 Metres Sprint
400 Metres Running
800 Metres Running
1500 Metres Running
3000 Metres Running
10 Kilometres Running
3000 Metres Steeplechase
110 Metres Hurdles
Marathon
Long Jump
High Jump
Hop, Step and Jump
Discus
Shotput
Javelin
Hammer Throwing
4 x 400 Metres Relay
4 x 100 Metres Relay

Your List Of Key Elements

Ingredients For Sherbet

The list below details the ingredients for Sherbet.

1 cup Icing Sugar
1 Teaspoon Baking Soda
1 Teaspoon Powder Citric Acid
2 Teaspoon Powder Tartaric Acid
2 Teaspoon Lemon or other Essence or 2 Tablespoons Fruit Drink Powder

Your List Of Key Elements

Identifying Programmes

Introduction

Now we will identify some programmes - some of these examples are programmes and some are not. We will look at:

a shopping list

some learning units

instructions for starting a car

a banquet schedule

a Basic Program

instructions for turning the page of a book

a Pascal Program

a dinner menu.

Instructions

Identify which are programmes and which are not programmes and record your reasons why each of these examples is/is not a programme.

A Shopping List

The list below details a shopping list.

Groceries

1 loaf bread
1 litre milk
6 eggs
2 rolls paper towels
3 tomatoes
1 Kg of mince

Chemist

Pandol
Shampoo
Toothpaste

Programme?

Is this a programme?

Reasons?

Continued on next page

Identifying Programmes, Continued

Some Learning Worksheets

The list below details Learning Worksheets.

General and Background

What is Programming?

Procedure Design

Procedure Design
The Design Phase
Module Logic
Programming Shop

Programming Functions

Introduction to Language
The Implementation of Modules
Further Use of Modules
Further Language Constructs

Program Translation, Compiling, Testing and Modifying Programs

Programming Environment
Debugging and Testing

Optional Worksheets

File Handling
Alternative Approaches to Procedure Design
Advanced Data Structures
Standard Algorithms
Code Reuse
Screen Handling
Student Specific

Programme?

Is this a Programme?

Reasons:

Continued on next page

Identifying Programmes, Continued

Instructions For Starting A Car

The list below details instructions for starting a car.

Insert the key.

Make sure the transmission is in Park (or Neutral).

Depress the gas pedal.

Turn the key to the start position.

If the engine starts within six seconds, release the key to the ignition position.

If the engine doesn't start in six seconds, release the key, wait ten seconds, and repeat steps 3 through 6, but not more than five times.

If the car doesn't start, call the garage.

Programme?

Is this a Programme?

Reasons:

A Banquet Schedule

The list below details a schedule for a banquet.

6:00pm Guests Arrive
6:30pm Official Welcome
7:00pm Meal Served
8:00pm After Dinner Speech - Dr Earhart
9:00pm Bar Open

Programme?

Is this a programme?

Reasons:

Continued on next page

Identifying Programmes, Continued

A Basic Program

The list below details a Basic Program.

```
1000 REM FOOTSTEPS PROGRAM
1010 PRINT "ENTER FOOTSTEP DEPTH"
1020 PRINT "BETWEEN STREET AND"
1030 PRINT "GARAGE"
1040 INPUT A
1050 PRINT "ENTER FOOTSTEP DEPTH"
1060 PRINT "BETWEEN GARAGE AND"
1070 PRINT "HOUSE"
1080 INPUT B
1090 LET C=A/B
1100 PRINT "FOOTSTEPS BETWEEN STREET"
1110 PRINT "AND GARAGE ARE ";C;" TIMES"
1120 PRINT "AS DEEP AS THOSE BETWEEN"
1130 PRINT "GARAGE AND HOUSE"
1140 END
```

Programme ?

Is this a programme?

Reasons:

Instructions For Turning A Page

The list below details instructions for turning the page of a book.

Lift hand.
Move hand to right side of book.
Grasp corner of top page.
Move hand from right to left until page is positioned so that what is on other side can be read.
Let go of page.

Programme ?

Is this a programme?

Reasons:

Continued on next page

Identifying Programmes, Continued

A Pascal Program

The list below details a Pascal Program.

```
program Footsteps;
(*
                    Ratio of Footstep Depths
   Author: Errol Thompson                  Date: 1 Apr 92

A program from Byte Brothers for calculating the ratio of
   footstep depths.

*)

var
 Depth1,  (* Depth of footsteps between garage and street
*)
 Depth2,   (* Depth of footsteps between house and garage
*)
 Ratio : real;(* Calculated ratio depths               *)

begin
 Write('Enter the footstep depth between street and',
       ' garage');
  Readln(Depth1);

 Write('Enter the footstep depth between garage and',
       ' house');
 Readln(Depth2);

                   (* Calculate the ratio between the  *)
                   (* two depths.                      *)
 Ratio := Depth1 / Depth2;

 Writeln('The footsteps between the street and the',
        'garage are ', Ratio:4:2 ,' times as');
 Writeln('deep as those between the garage and the',
         ' house');

  Readln;           (* Pause for user to read results *)
end.
```

Programme? Is this a programme?

Reasons:

Continued on next page

Identifying Programmes, Continued

A Dinner Menu

The list below details a dinner menu.

Smoked chicken with rock melon and oriental salad of capsicum, carrot and turnip

- o0o -

Chargrilled fillet steak topped with bearnaise sauce and served with duchesse potatoes and asparagus spears

or

Shallow fried breast of turkey coated with cranberry and port wine sauce, served with potatoes and fresh garden vegetables

or

Poached orange roughy coated with crayfish and vermouth sauces, served with rice and stirfried vegetables

- o0o -

Sweet pastry fan filled with nut and spices in honey, topped with a baked egg custard, dusted with icing sugar

- o0o -

A selection of fine chesses and fresh seasonal fruit

- o0o -

Freshly brewed coffee, decaffeinated coffee or a blended tea

Programme?

Is this a Programme?

Reasons:

Appendix 2 Syntax and Semantics

Pascal Language

Overview

Introduction This Worksheet contains:

the difference between syntax and semantics
the basic program structure for a program in the language
basic types of data
at least three language constructs
write valid expressions
at least six standard procedures and functions.

In This Worksheet This Worksheet contains the following Topics.

Language Constructs And Terminology

Introduction We want to look at how we define a language and the terminology that we use with respect to programming languages.

Is it sufficient to have words and a concept of a sentence?

Exercise 1 Examine the following attempts at English sentences.

Which ones are valid or invalid and why?

- The cat sat on the fence
- Fence cat the on the sat
- Cat the sat the on fence
- Ran the cat dog after the
- The dog ran after the cat
- Dog the cat ran the after

How do we communicate what are valid or invalid sentences?

How can we communicate the rules?

Is there one rule or a set of rules?

Continued on next page

Language Constructs And Terminology, Continued

Simple English Statement Syntax

Can we represent the rules or grammar for constructing sentences for the simple English statements above as:

<Sentence> -> <Noun Phrase> <Verb> <Noun Phrase>

<Noun Phrase> -> [<Preposition>] <Article> <Noun>

<Preposition> -> after | on

<Article> -> the

<Noun> -> dog | cat | fence

<Verb> -> ran | sat.

The symbols used are interrupted as:

<Noun> means that this is a symbol which needs to be defined or is being defined

→ means is (ie. An article is the)

| means or (ie. A Noun is either dog or cat or fence)

[] means that the construct in the square brackets is optional.

The first rule of the grammar defines a **sentence** as a **noun phrase** followed by a **verb** followed by a **noun phrase**. The last rule defines a **verb** as either ran or sat.

Defining The Syntax Of A Language

In programming languages, we would refer to such rules as defining the **syntax** of the language. Using such rules, we should be able to determine whether a particular sentence is valid for a given language. Is the following sentence valid in terms of the rules specified above?

The fence ran on the cat.

With spoken or written languages, it is in theory possible to devise such rules to define all valid sentences in the language. However, the number of alternatives is extremely large. As a result, we seldom formally define a spoken or written language, although, we may use such rules to teach a language supported by some explanation of what we mean by a noun clause.

Continued on next page

Language Constructs And Terminology, Continued

Exercise 2

The previous sentence was valid for the language rules that we specified.

Here are some more English sentences. Which of these would you regard as being acceptable sentences?

Endeavour to explain why you would regard them as acceptable or unacceptable. Is the problem one of syntax or is there some other problem?

- The house jumped over the dog.
- Dog house the over jumped.
- The road ran through the field.
- Jumped rattled from up came.
- The breeze rattled the windows.
- The tree swam across the river.
- The man stood on the burning deck.
- Swam river the across tree the.
- The man came up from the deep to receive the ball.

Semantics

We would probably argue that although these sentences could be determined as correct from the view of syntax, that what they are saying was not possible. That is their meaning would not be acceptable. In programming languages we call the aspect which defines the meaning of a language construct as **semantics**.

The semantics are important in a programming language as they tell us what we should expect a certain construct or statement in the language to do. What did we define as a program? Do we simply group constructs together or do we organise constructs that perform certain functions to achieve a given objective or task?

Pascal Programming Language Syntax

Exercise 3 Examine the six sample programs at the back of this workbook. What we want to develop is the syntax for a Pascal program. This is the top level syntax rule. Consider using the key words: name, declarations and statements. The objective of this exercise to understand the structure and syntax of Pascal and not to be able to quote the Pascal syntax exactly. There are reference books for obtaining syntax information.

Exercise 4 Now examine the variable declaration statements in the six sample programs. These start with the word var. See whether you can identify the structure of this statement. You could also examine the executable statements between the begin and end. In particular see whether you can define the structure of the assignment statement (this uses the symbol :=), the while statement and the if statement.

Continued on next page

Pascal Programming Language Syntax, Continued

Exercise 5 Now look at the syntax rules in Chapter 1 of the Reference Materials. These define a subset of the Pascal programming language. Using these syntax diagrams work through the expansion of a program to verify that the following is a valid Pascal program. Start with the program definition and expand each symbol enclosed in <> until you have created the program text.

```
Program Hello_World;
begin
  Writeln('Hello World');
end.
```

Note: The statement "Writeln('Hello World')" is a standard procedure call.

Repeat the process for the following variation which you saw back in the module "What is Programming?".

```
Program Hello_World;
var
  Hello : string;
begin
  Hello := 'Hello World';
  Writeln(Hello);
end.
```

NOTE: For this exercise treat 'Hello World' as an expression.

Repeat the process and correct where necessary.

```
Program Sample;

var
Answer : str;
begin
Answer = 'No'
Writeln(Answer);
```

Continued on next page

Programming Language Syntax, Continued

Using Syntax To Write Code

When you write a Pascal program, you are not going to work through the syntax diagrams. As you become more familiar, you will find that you reference the diagrams less and less. They will become a reference for verifying that what you have done is correct.

One way of looking at writing program code is that you are inserting instructions or statements within a template. The highest level template is the program structure. There are then lower level templates for each language construct. As a programmer, you will remember many of the templates but, at other times you will want to look at the templates to determine how a particular construct should be used.

Pascal Programming Language Semantics

Understanding Semantics

Just examining the syntax alone does not help us understand how to program using Pascal. We need to understand the semantics or meanings of the constructs. That is why we have included the tables on operator meanings. There are still difficulties. How do variables of type string work and what are the restrictions on the use of operators with some variable types.

A compiler like Turbo Pascal does check that variables of the right type are being used in the correct places during the compile. However this is the compiler checking some of the semantic rules rather that the syntax rules.

Exercise 6

Read the Semantics Material in Chapter 1 of the Reference Material, then do the Exercise Set 3 Practical Pascal Language. (**NOTE**: You should have completed Exercise Sets 1 and 2 before attempting Exercise Set 3.)

Appendix 3 Interview Schedule

The following is the interview schedule as originally designed and submitted with the ethics application. All interviews included the first question from each major sections but deviated in terms of the follow-up questions based on the direction of the interview and the emphasis perceived by the interviewer.

Interviewer Schedule for Practitioner Interviews

Notes

The aim is to keep the person talking and being as discursive as possible. Do not interrupt to ask follow-up questions; let them talk as much as possible and ask questions only if they stop.

When I come to a question they have answered before, it is still worth asking it unless they had nothing to say on it (e.g. it was irrelevant). Introduce the question with "You have already discussed this before, but…"

Encourage the interviewee. I am not there to judge, and positive feedback will help the study get to the root of what the critical aspects are for success in the use of object-oriented development techniques.

In the rest of this script, the text indicates questions that should be asked, preferably using the exact wording provided. Further notes to the interviewer are in italics.

Introduction

The questions I am asking are general questions on your use and understanding of object-oriented software development techniques. Sometimes the questions may be repetitive or seem trivial in the context of your understanding and work, but we want to hear this in your words, not assume it ourselves.

Background

What is your background in the use of object-oriented software development practices?

[The objective here is to get them to talk about the length of time that they have been using these techniques and what those techniques are.]

How did you come to be using object-oriented software development practices?

How long have you been using object-oriented software development practices?

Did you have any formal training in object-oriented development practices?

If so what was the training?

How do you apply object-oriented development practices?

Do you use any particular approach or methodology for applying object-oriented development practices?

If so describe the approach or methodology?

If not is there a reason why you don't use a particular approach or methodology?

[It is possible that these background questions may lead to some of the more specific questions that we want to ask later. This isn't a problem and will be helpful in later analysis.]

Success Factors

How do you assess whether a program has been written using object-oriented techniques and practices?

[The objective here is to uncover how they might assess a program as being object-oriented or not object-oriented. Two or three examples might help draw this out. Examples should all do the same thing but use different coding structures. A simple teaching example would be ideal. Examples should be coded in Java, VB.NET, and C#. (Things example)]

Have a look at these, how do you assess these in terms of writing code using object-oriented practices?

Program written using an object-oriented language such as Java or C#. code structure unimportant

Program written using an object-oriented language and using objects to modularise the code

Program written using domain objects, multiple tier architecture, and design patterns.

What made you say something was object-oriented or not object-oriented?

Learning

How do you think a person learns object-oriented techniques and practices?

If you were asked to help a person learn about object-oriented software, how would you describe it to them?

What things would you want to emphasise in helping a person learn about object-oriented techniques and practices?

*For **X** = each thing elicited in the above question*

Do

How would you describe **X**?

Why would you give emphasis to **X**?

Can you provide an example of **X**?

*End of For **X** = each concept*

Object-oriented concepts

What are the concepts that you see as being important when performing object-oriented development?

[Caution: If they identified a concept in the previous question related to learning object-oriented then going over it again now may irritate the interviewee. The previous question was designed to elicit higher level thoughts or ideas maybe process related rather than necessarily understanding of object-orientation and object-oriented concepts. This question is pushing toward the lower level concepts that are usually listed in the front of texts on object-oriented programming.]

How do you describe an object?

What is the significance of an object in object-oriented software development?

How would you describe a class?

What is the significance of a class in object-oriented development?

*For **X** = each concept elicited in the above question*

Do

How would you describe **X**?

Why is **X** important?

Can you provide an example of **X**?

*End of For **X** = each concept*

Generic Ideas

How would you explain to a person learning about object-oriented practices, what a 'system' is?

How do you relate this description of a 'system' to the way you think about object-oriented systems?

How would you describe what a ‘program’ is?

How do you relate this description of a program to the way it is written?

[Some may respond with very high level definitions or descriptions. The objective is to try and draw these descriptions back to a specifically object-oriented context.]

Conclusion

Is there anything else you’d like to tell me about your use or understanding of object-oriented techniques, anything that you think is an important aspect of it but that these questions haven’t brought out?

[Once they seem to have finished talking, thank them very much for their time and their willingness to help.]

Appendix 4 Ethics Application

This appendix contains the ethics application and supporting documents that were submitted to the Massey University Ethics committee. The documents included are:

Practitioner information sheet

Practitioner consent form

Consent letter

Massey University
Department of Information Systems
College of Business

P.O.Box 756
Wellington
New Zealand
T 64 4 801 5799
F 64 4 801 2696
www.massey.ac.nz

Improving Learning in Software Development Topics

INTERVIEWEE INFORMATION SHEET

Researcher Introduction

This research is being conducted by Errol Thompson, a PhD student and Lecturer in Information Systems on the Wellington Campus of Massey University. It is being completed as part of his PhD studies that are being supervised by Associate Professor Janet Davies, Department of Technology, Science, and Mathematics Education and Associate Professor Kinshuk Department of Information Systems (Palmerston North).

Introduction

The teaching of introductory computer programming has focussed on the use of procedural programming techniques. With the wide spread use of object-oriented languages, interest in shifting this teaching to an objects-first approach has increased. This move is the result of poor results from students in learning object-oriented programming techniques. This research aims to develop an approach to improving learning in the context of object-oriented software development based on relational learning theory. This theory suggests that the learner's perception of the learning task influences the learner's approach to learning and consequently the achievement of the learning outcomes. The focus of the research is on identifying the "educationally critical aspects" by first interviewing experienced practitioners, lecturers, and graduate students. Educationally critical aspects are those aspects of the subject area that have the highest impact on the learner's ability to achieve the required learning.

The identified educational aspects will then be used in the teaching of object-oriented software development. The second phase of the research will endeavour to assess the effectiveness of a teaching strategy with respect to these critical aspects and how the learner's awareness of the identified aspects relates to their ability to develop object-oriented software.

Participant Recruitment

I am approaching people who have some experience in using object-oriented software development techniques to invite them to participate in the first part of this study. Those

approached include people who are using object-oriented techniques in the workplace, graduate students who have completed papers that introduced object-oriented concepts, and lecturers who teach the techniques. A total of twenty to thirty people across the three categories will be interviewed.

Participant involvement

Your involvement in the project will be to participate in a recorded conversational interview that will last approximately an hour. You will also be given an opportunity to review, make comment on, and request changes to the transcript of the interview.

Interviews will be conducted at a time and place convenient to both you and the researcher. Whenever possible, the interviews will be conducted face-to-face to eliminate as many technical difficulties as possible. Where a face-to-face meeting is not practical, you will be offered the option of a telephone / Internet audio conference or an Internet video conference using either MSN or Skype. As a last resort conducting the interview via an e-mail exchange will be considered.

Project Procedures

Recordings of practitioner interviews will be transcribed verbatim and destroyed. A copy of the recording and transcript of your interview will be offered to you. The transcripts will be analysed by the researcher using the techniques of phenomenography to identify the “critical aspects” and the variations in the ways of experiencing the identified “critical aspects”.

No participants will be identified in any papers written or in the thesis. Confidentiality of participants will be retained. Transcripts will be identified by a unique code and kept separate from consent forms. All transcripts will be held in a secure location and be accessible only to the researcher and supervisors.

The findings of this research will be published in the PhD thesis and in papers presented at conferences and printed in journals. A summary of the findings will be sent to all participants.

Participant's Rights

You are under no obligation to accept this invitation. If you decide to participate, you have the right to:

- decline to answer any particular question;
- withdraw from the study at anytime prior to the finalisation of the results;
- ask any questions about the study at any time during participation;
- provide information on the understanding that your name will not be used unless you give permission to the researcher;
- be given access to a summary of the project findings when it is concluded.

- ask for the recording to be turned off at any time during the interview.

Project Contacts

Should you have any questions related to this research then you may approach the researcher or his supervisors as listed below.

Errol Thompson
Information Systems
Massey University
Private Bag 756
Wellington

Phone +64 4 8015799 x6531
e-mail:
E.L.Thompson@massey.ac.nz

Associate Professor Janet Davies
Technology, Science, and Mathematics Education
Massey University
Private Bag 756
Wellington
Phone: +64 6 8015799 x6321
e-mail:
j.r.davies@massey.ac.nz

Associate Professor Kinshuk
Information Systems
Massey University
Private bag 11-222
Palmerston North

Phone: +64 4 8015799 x2090
e-mail:
kinshuk@massey.ac.nz

Committee Approval Statement

This project has been reviewed and approved by the Massey University Human Ethics Committee, Wellington Application 05/69. If you have any concerns about the conduct of this research, please contact Professor Sylvia Rumball, Chair, Massey University Campus Human Ethics Committee: Wellington, telephone +64 6 350 5249, email humanethicswn@massey.ac.nz

Department of Information Systems
College of Business

Improving learning in software development topics

INTERVIEW PARTICIPANT CONSENT FORM

This consent form will be held for a period of five (5) years

I have read the Information Sheet and have had the details of the study explained to me. My questions have been answered to my satisfaction, and I understand that I may ask further questions at any time.

I agree/do not agree to the interview being audio taped.

I wish/do not wish to have my tapes and transcript returned to me.

I agree to participate in this study under the conditions set out in the Information Sheet.

Signature: ..... **Date:** .....

Full Name - printed .....

To xxxx, Head of Department Information Systems (via e-mail),

My PhD proposal, “Improving Learning in Software Development Topics”, aims to identify “educationally critical aspects” with respect to the learning of object-oriented software development. I have proposed a two stage data gathering process. The first involves interviews of practitioners, lecturers, and graduate students to identify the “educationally critical aspects” that they use, and the variations in awareness of the identified aspects. Educationally critical aspects are those aspects of the subject area that have the highest impact on the learner’s ability to achieve the required learning.

The second involves gather data from the teaching strategy that I have been using with 157.374 Object-Oriented Design and Development and now will apply to the teaching of 157.232 Business Applications Programming and 157.231 Implementation of Information Systems. The data being gathered relates to the perceptions of these concepts that the students are developing through the teaching of the course.

I am seeking approval to

1) approach lecturers and graduate students within the department to be involved in the interviews,
2) ask the Wellington Departmental Secretary to hold student consent forms until all marking of 157.231 is completed,
3) trial the data gathering techniques with volunteer students from 157.232 Business Applications Programming, and
4) conduct the main study with volunteer students from 157.231 Implementation of Information Systems.

I have attached copies of the information sheets that will be given to the two participating groups.

Approval is being sought from the Human Ethics Committee for this project. My primary supervisor for this research is Janet Davies of the Department of Technology, Science, and Mathematics Education. Janet is a member of the HEC and has helped me prepare the ethics application.

I look forward to hearing a positive response from you.

Department of Information Systems
College of Business
Massey University

Professor xxxx
Head of xxxxxxxxxxxxxxxx
xxxxxxxxxxxx University

Dear Professor xxxx,

My name is Errol Thompson. I am a lecturer in Information Systems at Massey University and a PhD student.

As part of the research for a PhD, I am seeking to interview lecturers and graduate students who are teaching or applying object-oriented principles. The purpose of this research is to identify “educationally critical aspects” for the learning of object-oriented programming and the variations in awareness of those aspects held by the participants.

I am seeking approval to approach lecturers and graduate students within your department to be involved in these interviews.

I have attached a copy of the information sheet that will be given to the potential interviewees.

Approval is being sought from the Human Ethics Committee of Massey University for this project. My primary supervisor for this research is Janet Davies of the Department of Technology, Science, and Mathematics Education at Massey University.

I look forward to hearing a positive response from you.

Yours Faithfully,

Errol Thompson

Appendix 5 Selected Texbooks

Anderson, J., & Franceschi, H. (2005). *Java 5 illuminated: An active learning approach*. Sudbury, MA: Jones and Bartlett Publishers.

Barker, J. (2005). *Beginning Java objects: From concepts to code* (2nd ed.). Berkeley, CA: Apress.

Barnes, D. J., & Kölling, M. (2006). *Objects first with Java: A practical introduction using BlueJ* (3rd ed.). Harlow, England: Prentice Hall.

Becker, B. W. (2007). *Java: Learning to program with robots*. Boston, MA: Thomson: Course Technology.

Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (2005). *Karel J Robot: A gentle introduction to the art of object-oriented programming in Java* (Preliminary ed.): Dream Songs Press.

Bronson, G. J. (2006). *Object-oriented program development using Java: A class-centered approach* (Enhanced ed.). Boston, MA: Thomson: Course Technology.

Brookshear, J. G. (2007). *Computer science: An overview* (9th ed.). Boston: Pearson Education Limited.

Farrell, J. (2006a). *Java programming* (3rd ed.): Thomson: Course Technology.

Farrell, J. (2006b). *An object-oriented approach to programming logic and design*. Boston , MA: Thomson: Course Technology.

Felleisen, M. (2005). How to design class hierarchies, *Proceedings of the 2005 workshop on Functional and declarative programming in education*. Tallinn, Estonia: ACM.

Guzdial, M., & Ericson, B. (2005). *Introduction to computing & programming with Java: A multimedia approach*. Upper Saddle River, NJ: Prentice Hall.

Jeffries, R. E. (2004). *Extreme programming adventures in C#*. Redmond, WA: Microsoft Press.

Langr, J. (2005). *Agile Java: Crafting code with test-driven development*. Upper Saddle River, NJ: Prentice Hall PTR.

Lewis, J., & Loftus, W. (2006). *Java software solutions: Foundations of program design* (5th ed.). Boston: Addison Wesley.

Malik, D. S. (2006). *Java™ programming: From problem analysis to program design* (2nd ed.): Thomson: Course Technology.

Morelli, R., & Walde, R. (2006). *Java, Java, Java: Object-oriented problem solving* (3rd ed.). Upper Saddle River: Prentice Hall.

Sierra, K., & Bates, B. (2005). *Head first Java* (2nd ed.). Sebastopol, CA: O'Reilly Media Inc.

Soroka, B. I. (2006). *Java 5: Objects first*: Jones and Bartlett Publishers.

Stein, L. A. (2003, 24 March 2004). Interactive programming in Java. Retrieved 28 September, 2004, from http://www.cs101.org/ipij/ObjectOriented/InteractiveProgramminginJava

Wiener, R. (2007). *Modern software development using C#.NET*. Boston, MA: Thomson: Course Technology.

References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition, 9*, 422-433.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 10*, 483-495.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language*. New York: Oxford University Press.

Ambler, A. L., Burnett, M. M., & Zimmerman, B. A. (1992). Operational versus definitional: a perspective on programming paradigms. *IEEE Computer, 25*(9), 28-43.

Anderson, J., & Franceschi, H. (2005). *Java 5 illuminated: An active learning approach*. Sudbury, MA: Jones and Bartlett Publishers.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Raths, J., & Wittrock, M. C. (Eds.). (2001). *A taxonomy for learning and teaching and assessing: A revision of Bloom's taxonomy of educational objectives*: Addison Wesley Longman.

Andrianoff, S., & Levine, D. B. (2002). Role playing in an object-oriented world. *Inroads - The SIGCSE Bulletin, 34*(1), 121-125.

Angelo, T. A., & Cross, K. P. (1993). *Classroom assessment techniques: A handbook for college teachers*. San Francisco: Jossey-Bass Publishers.

Armour, P. G. (2000a). The case for a new business model: Is software a product or a medium? *Communications of the ACM, 43*(8), 19-22.

Armour, P. G. (2000b). The five orders of ignorance: Viewing software development as knowledge acquisition and ignorance reduction. *Communications of the ACM, 43*(10), 17-20.

Armour, P. G. (2003a). Closing the learning application gap. *Communications of the ACM, 46*(9), 27-31.

Armour, P. G. (2003b). *The Laws of Software Process: A new model for the production and management of software*. Boca Raton: Auerbach Publications.

Ashworth, P., & Lucas, U. (1998). What is the 'world' of phenomenography? *Scandinavian Journal of Educational research, 42*(4), 415-431.

Auer, K., & Miller, R. W. (2002). *Extreme programming applied: Playing to win*. Boston: Addison Wesley.

Austing, R. H., Barnes, B. H., Bonnette, D. T., Engel, G. L., & Stokes, G. (1979). Curriculum '78: recommendations for the undergraduate program in computer science - a report of the ACM curriculum committee on computer science. *Communication of the ACM, 22*(3), 147-166.

Barclay, W. (1975). *The gospel of Matthew* (Vol. Two - Chapters 11 to 28). Edinburgh: The Saint Andrews Press.

Barker, J. (2005). *Beginning Java objects: From concepts to code* (2nd ed.). Berkeley, CA: Apress.

Barnes, D. J., & Kölling, M. (2006). *Objects first with Java: A practical introduction using BlueJ* (3rd ed.). Harlow, England: Prentice Hall.

Beard, C. (2008). Introducing the concept series; Call for participation. Retrieved 9 September, 2008, from http://labs.mozilla.com/2008/08/introducing-the-concept-series-call-for-participation/

Beck, K. (1997). *Smalltalk best practice patterns* (1st ed.). Upper Saddle River, NJ: Prentice Hall.

Beck, K. (2000). *Extreme programming explained: Embrace change*. Boston: Addison Wesley Longman.

Beck, K. (2003). *Test-driven development by example*. Boston, MA: Addison Wesley.

Beck, K. (2007). *Implementation patterns*. Upper Saddle River, NJ: Addison Wesley.

Beck, K. (2008). Re: [XP] Eerie similarities... is UCD dead? In extremeprogramming@yahoogroups.com (Ed.).

Becker, B. W. (2007). *Java: Learning to program with robots*. Boston, MA: Thomson: Course Technology.

Ben-Ari, M., & Sajaniemi, J. (2004). Roles of variables as seen by CS educators. *Inroads - The SIGCSE Bulletin, 36*(3), 52-56.

Bennedsen, J., & Caspersen, M. E. (2005, 1-2 October). *An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course.* Paper presented at the Proceedings of the 2005 international workshop on Computing education research (ICER 2005), University of Washington, Seattle.

Bennedsen, J., & Caspersen, M. E. (2006). Abstraction ability as an indicator of success for learning object-oriented programming? *Inroads - The SIGCSE Bulletin, 38*(2), 39-43.

Bennedsen, J., & Caspersen, M. E. (2008, 6-7 September). *Abstraction Ability as an Indicator of Success for Learning Computing Science?* Paper presented at the Fourth international workshop on computing education research (ICER 2008), Sydney, Australia.

Bergin, J. (2000, July). Fourteen pedagogical patterns. From http://csis.pace.edu/~bergin/PedPat1.3.html

Bergin, J. (2002, 11 May). Some pedagogical patterns. From http://csis.pace.edu/~bergin/patterns/fewpedpats.html

Bergin, J. (2003). *Teaching polymorphism with elementary design patterns.* Paper presented at the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA.

Bergin, J., Eckstein, J., Manns, M. L., & Sharp, H. (2001a). Feedback patterns. In Addison-Wesley (Ed.), *Pattern Languages of Program Design*.

Bergin, J., Eckstein, J., Manns, M. L., & Sharp, H. (2001b). Patterns for active learning. In Addison-Wesley (Ed.), *Pattern Languages of Program Design*.

Bergin, J., Eckstein, J., Manns, M. L., Sharp, H., & Sipos, M. (2001). Teaching from Different Perspectives. In Addison-Wesley (Ed.), *Pattern Languages of Program Design*.

Bergin, J., Eckstein, J., Manns, M. L., & Wallingford, E. (2001). Patterns for gaining different perspectives. In Addison-Wesley (Ed.), *Pattern Languages of Program Design*.

Bergin, J., Marquardt, K., Manns, M. L., Eckstein, J., Sharp, H., & Wallingford, E. (2001). Patterns for experiential learning. In Addison-Wesley (Ed.), *Pattern Languages of Program Design*.

Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (2005). *Karel J Robot: A gentle introduction to the art of object-oriented programming in Java* (Preliminary ed.): Dream Songs Press.

Berglund, A., & Lister, R. (2007). Debating the OO debate: Where is the problem? In R. Lister & Simon (Eds.), *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)* (Vol. 88, pp. 171-174). Koli National Park, Finland: Australian Computer Society, Inc.

Biggs, J. (1987). *Student approaches to learning and studying*. Hawthorn, Vic: Australian Council of Educational Research.

Biggs, J. B. (1993). From theory to practice: a cognitive systems approach. *Higher education research and development, 12*(1), 73-85.

Biggs, J. B. (1999). *Teaching for quality learning at University*. Buckingham: Open University Press.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press.

Biggs, J. B., & Moore, P. J. (1993). *The process of learning* (3rd ed.). New York: Prentice Hall.

Black, A. P., Hutchinson, N. C., Jul, E., & Levy, H. M. (2007, 9-10 June). *The development of the Emerald programming language.* Paper presented at the History of programming languages, San Diego, California.

Booth, S. A. (1992). *Learning to program: A phenomenographic perspective* (Vol. 89). Göteborg: Acta Universtatis Gothoburgensis.

Booth, S. A. (1993). *The experience of learning to program. Example: recursion.* Paper presented at the 5 éme Workshop sur la Psychologie de la Programmation (the fifth workshop of the Psychology of programming Interest Group), Paris.

Booth, S. A. (1997). On phenomenography, learning and teaching. *Higher Education Research & Development, 16*(2), 135-158.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007). Threshold concepts in computer science: Do they exist and are they useful? *ACM SIGCSE Bulletin, 39*(1), 504-508.

Bowden, J. A. (2000). The nature of phenomenographic research. In J. A. Bowden & E. Walsh (Eds.), *Phenomenography* (pp. 1-18). Melbourne, Australia: RMIT University Press.

Bowden, J. A. (2005). Reflections on the phenomenographic team research process. In J. A. Bowden & P. Green (Eds.), *Doing developmental phenomenography* (pp. 11-31). Melbourne, Australia: RMIT University Press.

Bowden, J. A., & Green, P. (Eds.). (2005). *Doing developmental phenomenography*. Melbourne, Australia: RMIT University Press.

Bowden, J. A., & Marton, F. (2003). *University of learning*. London: Routledge Falmer.

Bowden, J. A., & Walsh, E. (Eds.). (2000). *Phenomenography*. Melbourne, Australia: RMIT University Press.

Boyer, E. L. (1990). *Scholarship reconsidered: priorities of the professoriate*. San Francisco: Jossey-Bass Inc.

Bradley, J. C., & Millspaugh, A. C. (2000). *Advanced Programming in Visual Basic, version 6.0* (International ed.). Boston, MA: Irwin, McGraw-Hill.

Bradley, J. C., & Millspaugh, A. C. (2003). *Advanced Programming in Visual Basic .NET*. Boston, MA: Irwin, McGraw-Hill.

Bransford, J. D., Brown, A. L., & Cocking, R. R. (Eds.). (2000). *How people learn: brain, mind, experience, and school* (Expanded Edition ed.). Washington: National Academy Press.

Brew, A. (2001). *The nature of research : inquiry in academic contexts*. London ; New York: Routledge Falmer.

Bronson, G. J. (2006). *Object-oriented program development using Java: A class-centered approach* (Enhanced ed.). Boston, MA: Thomson: Course Technology.

Brookshear, J. G. (2007). *Computer science: An overview* (9th ed.). Boston: Pearson Education Limited.

Brown, A. L. (1978). Knowing when, where, and how to remember: A problem of metacognition. In R. Glaser (Ed.), *Advances in instructional psychology* (Vol. 1, pp. 77-165). Hillsdale, NJ: Lawrence Erlbaum Associates.

Brown, A. L. (1987). Metacognition, executive control, self-regulation, and other more mysterious mechanisms. In F. E. Weinert & R. H. Kluwe (Eds.), *Metacognition, motivation, and understanding* (pp. 65-116). Hillsdale, NJ: Lawrence Erlbaum Associates.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodley, I. (2004). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education, 3*, 143-160.

Bruce, C., McMahon, C., Buckingham, L., Hynd, J., & Roggenkamp, M. (2003). *Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university*.

Bruce, K. B., Danyluk, A., & Murtagh, T. (2005). Why structural recursion should be taught before arrays in CS 1. *Inroads - The SIGCSE Bulletin, 37*(1), 246-250.

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (1997). *Mental representations constructed by experts and novices in object-oriented program comprehension.* Paper presented at the Human-computer interaction: INTERACT'97.

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering, 7*(2), 115-156.

Burns, J. (1994). *Extending critique within phenomenography.* Paper presented at the Proceedings of Phenomenography: Philosophy and Practice, Brisbane, Queensland.

Burton, P. J., & Bruhn, R. E. (2003). Teaching programming in the OOP era. *Inroads - The SIGCSE Bulletin, 35*(2), 111-114.

Byckling, P., & Sajaniemi, J. (2005, June). *Using roles of variables in teaching: Effects on program construction.* Paper presented at the 17th Workshop of the Psychology of programming Interest Group, Sussex University.

Byrne, P., & Lyons, G. (2001). The effect of student attributes on success in programming. *Inroads - The SIGCSE Bulletin, 33*(3), 49-52.

Chi, M. T. H., Glaser, R., & Farr, M. J. (Eds.). (1988). *The nature of expertise*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Chik, P. P. M., & Lo, M. L. (2003). Simultaneity and the enacted object of learning. In F. Marton & A. B. M. Tsui (Eds.), *Classroom discourse and the space of learning* (pp. 89-110). Mahwah, NJ: London: Lawrence Erlbaum Associates, Publishers.

Clancy, M. J., & Linn, M. C. (1999). *Patterns and pedagogy.* Paper presented at the Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education.

Clarke, B. J. (2000). *An introduction to object-oriented systems development with Jade* (4th ed.). Auckland: Addison Wesley.

Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E., & Whalley, J. (2008). The teaching of novice computer programmers: bringing the scholarly-research approach to Australia. In Simon & M. Hamilton (Eds.), *Tenth Australasian Computing Education Conference (ACE 2008)* (Vol. 78, pp. 63-68). Wollongong, NSW, Australia: ACS. From http://crpit.com/confpapers/CRPITV78Clear.pdf

Cockburn, A. (2002). *Agile software development*. Boston: Addison Wesley.

Cockburn, A. (2005). *Crystal Clear: A human-powered methodology for small teams*. Boston: Addison Wesley.

Cohen, L., & Manion, L. (1994). *Research methods in education* (4th ed.). London: Routledge.

Cope, C. (2000). *Educationally critical aspects of the experience of learning about the concept of an information system.* Unpublished Dissertation, La Trobe University, Bundoora, Victoria.

Cope, C. (2002a). Educationally critical aspects of the concept of an Information System. *Informing Science, 5*(2), 67-79.

Cope, C. (2002b, June). *Seeking meaning: The educationally critical aspect of learning about Information Systems.* Paper presented at the InSITE - "Where parallels intersect".

Cope, C. (2002c). Using the analytical framework of a structure of awareness to establish validity and reliability in phenomenographic research, *Proceedings of the international symposium on current issues in phenomenography*. Canberra, Australia: Australian National University, Center for Educational Development and Academic Methods. From http://www.anu.edu.au/CEDAM/ilearn/symposium/abstracts.html

Coplien, J. O. (1992). *Advanced C++: programming styles and idioms*: Addison Wesley Longman.

Coplien, J. O. (2000). *Multi-paradigm design.* Vrije Universiteit Brussel, Brussel.

Corritore, C. L., & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal Human-Computer Studies, 50*(1), 61-83.

Corritore, C. L., & Wiedenbeck, S. (2001). An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal Human-Computer Studies, 54*(1), 1-23.

Covey, S. R. (1990). *The seven habits of highly effective people*. New York: Simon and Schuster.

Cross, K. P. (1998). Classroom research: Implementing the scholarship of teaching. *New Directions for Teaching and Learning, 75*, 5-12.

Cross, K. P., & Steadman, M. H. (1996). *Classroom research: Implementing the scholarship of teaching*. San Francisco: Jossey-Bass Publishers.

Czerwinski, R. (1985). Programming concepts and principles in the introductory computer science textbook. *SIGCSE Bull., 17*(4), 65-68.

Dale, N., & Weems, C. (1992). *Introduction to Pascal and structured design* (3rd Turbo version ed.). Lexington, Mass: Heath.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies, 39*, 237-267.

de Raadt, M., Hamilton, M., Lister, R., Tutty, J., Baker, B., Box, I., Cutts, Q. I., Fincher, S. A., Hamer, J., Haden, P., Petre, M., Robins, A., Simon, Sutton, K., & Tolhurst, D. (2005, 3-6 July). *Approaches to learning in computer programming students and their effect on success.* Paper presented at the Higher Education in a changing world: Research and Development in Higher Education, The University of Sydney.

Détienne, F. (1997). Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers, 9*(1), 47-72.

diSessa, A. A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: London, England: The MIT Press.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. C. Sphorer (Eds.), *Studying the novice programmer* (pp. 283-299): Lawrence Erlbaum.

Eckerdal, A., & Berglund, A. (2005, 1-2 October). *What does it take to learn 'Program thinking'?* Paper presented at the Proceedings of the 2005 international workshop on Computing education research (ICER 2005), University of Washington, Seattle.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2006). Putting threshold concepts into context in computer science education, *Proceedings of the 11th Conference on innovation and technology in computer science education* (pp. 103-107). Bologna, Italy: ACM.

Eckerdal, A., & Thuné, M. (2005, 27-29 June). *Novice Java programmers' conceptions of "object" and "class", and variation theory.* Paper presented at the ITiCSE'05, Monte de Caparica, Portugal.

Eckstein, J., Manns, M. L., & Voelter, M. (2001). Pedagogical patterns: capturing best practices in teaching object technology. *Software Focus, 2*(1), 9-12.

Entwistle, N. J. (1997). Contrasting perspectives on learning. In F. Marton, D. Hounsell & N. J. Entwistle (Eds.), *The experience of learning : implications for teaching and studying in higher education* (2nd ed., pp. 3-22). Edinburgh: Scottish Academic Press.

Evans, E. (2003). *Domain Driven Design: Tackling complexity in the heart of software*: Addison Wesley.

Farrell, J. (2006a). *Java programming* (3rd ed.): Thomson: Course Technology.

Farrell, J. (2006b). *An object-oriented approach to programming logic and design*. Boston , MA: Thomson: Course Technology.

Felleisen, M. (2005). How to design class hierarchies, *Proceedings of the 2005 workshop on Functional and declarative programming in education*. Tallinn, Estonia: ACM.

Felleisen, M., Flatt, M., Findler, R. B., Gray, K. E., Krishnamurthi, S., & Proulx, V. K. (2001). *How to design class hierarchies: Object-oriented programming and computing*.

Ferguson Smart, J. (2008). *Java Power Tools*. Beijing: O'Reilly.

Fincher, S. A. (1999). Analysis of design: An exploration of patterns and pattern languages for pedagogy. *Journal of Computers in Mathematics and Science Teaching: Special Issue CS-ED Research, 18*(3), 331-348.

Fincher, S. A., & Utting, I. (2002). Pedagogical patterns: Their place in the genre. *Inroads - The SIGCSE Bulletin, 34*(3), 199-200.

Fleury, A. E. (2000). Programming in Java: Student-constructed rules. *Inroads - The SIGCSE Bulletin, 32*(1), 197-201.

Fleury, A. E. (2001). Encapsulation and reuse as viewed by Java students. *Inroads - The SIGCSE Bulletin, 33*(1), 189-193.

Floyd, R. W. (1979). The paradigms of programming. *Communication of the ACM, 22*(8), 455-460.

Fowler, M. (1999). *Refactoring: Improving the design of existing code*. Reading, MA: Addison Wesley.

Freeman, E., Freeman, E., Sierra, K., & Bates, B. (2005). *Head first design patterns*. Sebastopol, CA: O'Reilly Media Inc.

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns : Elements of reusable object-oriented software*. Reading, Massachusetts: Addison Wesley Longman.

Garner, S., Haden, P., & Robins, A. (2005, 31 January - 4 February). *My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems.* Paper presented at the Seventh Australasian Computing Education Conference (ACE2005), The University of Newcastle, Australia.

Garrett, J. J. (2008). Aurora: Concept video part 1. Retrieved 9 September 2008, 2008, from http://www.adaptivepath.com/blog/2008/08/04/aurora-concept-video-part-1/

Gestwicki, P. V. (2007). Computer games as motivation for design patterns. *ACM SIGCSE Bulletin, 39*(1), 233-237.

Glaser, B. G., & Strauss, A. L. (1967). *The discovery of grounded theory: strategies for qualitative research*. New York: Aldine de Gruyter.

Glassick, C. E. (2000). Reconsidering scholarship. *Journal of Public Health Management and Practice, 6*(1), 4-9.

Glassick, C. E., Huber, M. T., & Maeroff, G. I. (1997). *Scholarship assessed: evaluation of the professoriate*. San Francisco: Jossey-Bass Inc.

Goldwasser, M., & Letscher, D. (2007). Teaching strategies for reinforcing structural recursion with lists, *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*. Montreal, Quebec, Canada: ACM.

Guzdial, M., & Ericson, B. (2005). *Introduction to computing & programming with Java: A multimedia approach*. Upper Saddle River, NJ: Prentice Hall.

Haberman, B. (2006). Pedagogical patterns: A means for communication within the CS teaching community of practice. *Computer Science Education, 16*(2), 87-103.

Hanks, B. (2007). Problems encountered by novice pair programmers. In R. Anderson, S. A. Fincher & M. Guzdial (Eds.), *The Third International Computing*

Education Research Workshop (ICER'07) (pp. 159-164). Georgia Institute of Technology, Atlanta, GA, USA: ACM Press.

Hazzan, O. (2002). Reducing abstraction level when learning computability theory concepts, *Proceedings of the 7th annual conference on Innovation and technology in computer science education* (pp. 156-160). Aarhus, Denmark: ACM Press.

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.

Hazzan, O. (2008). Reflections on teaching abstraction and other soft ideas. *SIGCSE Bull., 40*(2), 40-43.

Highsmith, J. (2002). *Agile software development ecosystems*. Boston: Addison Wesley.

Hoadley, C. M., Linn, M. C., Mann, L. M., & Clancy, M. J. (1996). When, why, and how do novice programmers reuse code? In W. D. Gray & D. A. Boehm-Davis (Eds.), *Empirical studies of programmers: Sixth workshop* (pp. 109-129). Norwood, NJ: Ablex Publishing Corporation.

Hoc, J.-M., Green, T. R. G., Samurçay, R., & Gilmore, D. J. (1990). *Psychology of programming*. London: Academic Press.

Hu, C. (2005). Dataless object considered harmful. *Communications of the ACM, 48*(2), 99-101.

Hu, C. (2008). Just say 'A class defines a data type'. *Communications of the ACM, 51*(3), 19-21.

Jeffries, R. E. (2002, 20 November). Adventures in C#: A better code manager. *XP Magazine,* from http://www.xprogramming.com/xpmag/acsCodeManager.htm

Jeffries, R. E. (2004). *Extreme programming adventures in C#*. Redmond, WA: Microsoft Press.

Jeffries, R. E. (2008). Re: [XP] Eerie similarities... is UCD dead? In extremeprogramming@yahoogroups.com (Ed.).

Jeffries, R. E., Anderson, A., & Hendrickson, C. (2000). *Extreme programming installed*: Addison Wesley.

Kegan, R. (2000). What "form" transforms?: A constructive-developmental approach to transformative learning. In J. Mezirow & Associates (Eds.), *Learning as transformation: critical perspectives on a theory in progress* (1st ed., pp. 35-69). San Francisco: Jossey-Bass.

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The model driven architecture^{TM} : Practice and promise*. Boston: Addison Wesley.

Ko, P. Y., & Marton, F. (2003). Variation and the secret of the virtuoso. In F. Marton & A. B. M. Tsui (Eds.), *Classroom discourse and the space of learning* (pp. 43-62). Mahwah, NJ; London: Lawrence Erlbaum Associates, Publishers.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM, 50*(4), 37-42.

Kuhn, T. S. (1996). *The structure of scientific revolutions* (3rd ed.). Chicago: University of Chicago.

Kuittinen, M., & Sajaniemi, J. (2003). First results of an experiment on using roles of variables in teaching. In M. Petre & D. Budgen (Eds.), *Joint 7th International Conference on Empirical Assessment of Software Engineering (EASE 2003) and 15th Psychology of Programming Interest Group Workshop (PPIG 2003)* (pp. 347-357). Keele University, Keele, UK. From http://www.ppig.org/papers/15th-kuittinen.pdf

Kuittinen, M., & Sajaniemi, J. (2004). Teaching roles of variables in elementary programming courses. *Inroads - The SIGCSE Bulletin, 36*(3), 57-61.

Langr, J. (2005). *Agile Java: Crafting code with test-driven development*. Upper Saddle River, NJ: Prentice Hall PTR.

Leach, L. (2003). Beyond independence. In N. Zepke, D. Nugent & L. Leach (Eds.), *Reflection to transformation: A self-help book for teachers* (pp. 105-119). Palmerston North: Dunmore Press Ltd.

Lewis, J., & Loftus, W. (2006). *Java software solutions: Foundations of program design* (5th ed.). Boston: Addison Wesley.

Lin, J. M.-C., & Wu, C.-C. (2007). Suggestions for content selection and presentation in high school computer textbooks. *Computers & Education, 48*(3), 508-521.

Liskov, B. (1988). Keynote address - data abstraction and hierarchy. *SIGPLAN Not., 23*(5), 17-34.

Liskov, B., & Guttag, J. (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*: Addison-Wesley Longman Publishing Co., Inc.

Liskov, B., & Zilles, S. (1974). Programming with abstract data types. *SIGPLAN Not., 9*(4), 50-59.

Lister, R. (2003). The five orders of teaching ignorance. *Inroads - The SIGCSE Bulletin, 35*(4), 16-17.

Lister, R. (2007). The neglected middle novice programmer: Reading and writing without abstracting. In S. Mann & N. Bridgeman (Eds.), *20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2007)* (pp. 133-140). Nelson: NACCQ.

Lister, R. (2008). After the gold rush: toward sustainable scholarship in computing. In Simon & M. Hamilton (Eds.), *Tenth Australasian Computing Education Conference (ACE 2008)* (Vol. 78, pp. 3-18). Wollongong, NSW, Australia: ACS.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004a). A multi-national study of reading and tracing skills in novice programmers. *Inroads - The SIGCSE Bulletin, 36*(4), 119-150.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C., & Whalley, J. (2006a). Research perspectives on the object-first debate. *Inroads - The SIGCSE Bulletin, 38*(4), 146-165.

Lister, R., Box, I., Morrison, B., Teneberg, J., & Westbrook, S. (2004b). The dimensions of variation in the teaching of data structures. *Inroads - The SIGCSE Bulletin, 36*(3), 92-96.

Lister, R., Simon, B., Thompson, E., Whalley, J., & Prasad, C. (2006b). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. In M. Goldweber & P. Salomoni (Eds.), *Innovation and Technology in Computer Science Education (ITiCSE 2006)* (pp. 118-122). Bolonga, Italy: Association for Computing Machinery.

Lo, M. L., Marton, F., Pang, M. F., & Pong, W. Y. (2003). Toward a pedagogy of learning. In F. Marton & A. B. M. Tsui (Eds.), *Classroom discourse and the space of learning* (pp. 189-225). Mahwah, NJ; London: Lawrence Erlbaum Associates, Publishers.

Loui, R. P. (2008). In praise of scripting: Real programming pragmatism. *IEEE Computer, 41*(7), 22-26.

Malik, D. S. (2006). *Java™ programming: From problem analysis to program design* (2nd ed.): Thomson: Course Technology.

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*(3), 28-49.

Marton, F. (2000). The structure of awareness. In J. A. Bowden & E. Walsh (Eds.), *Phenomenography* (pp. 102-116). Melbourne, Australia: RMIT University Press.

Marton, F., & Booth, S. A. (1997). *Learning and awareness*. Mahwah, NJ: Lawrence Erlbaum Associates.

Marton, F., Dall'Alba, G., & Beaty, E. (1993). Conceptions of learning. *International Journal of Educational Research, 19*(3), 277-300.

Marton, F., & Pang, M. F. (1999). Two faces of variation, *8th European Conference for Learning and Instruction*. Göteborg University, Göteborg, Sweden.

Marton, F., & Pang, M. F. (2006). On some necessary conditions of learning. *Journal of the Learning Sciences, 15*(2), 193-220.

Marton, F., & Pong, W. Y. (2005). On the unit of description in phenomenography. *Higher Education Research and Development, 24*(4), 335-348.

Marton, F., Runesson, U., & Tsui, A. B. M. (2003). The space of learning. In F. Marton & A. B. M. Tsui (Eds.), *Classroom discourse and the space of learning* (pp. 3-40). Mahwah, NJ; London: Lawrence Erlbaum Associates, Publishers.

Marton, F., & Säljö, R. (1976a). Qualitative differences in learning - 1: Outcome and process. *British Journal of Educational Psychology, 46*, 4-11.

Marton, F., & Säljö, R. (1976b). Qualitative differences in learning - 2: Outcome as a function of the learner's conception of the task. *British Journal of Educational Psychology, 46*, 115-127.

Marton, F., & Säljö, R. (1997). Approaches to learning. In F. Marton, D. Hounsell & N. J. Entwistle (Eds.), *The experience of learning : implications for teaching and studying in higher education* (2nd ed., pp. 39-58). Edinburgh: Scottish Academic Press.

Marton, F., & Tsui, A. B. M. (Eds.). (2003). *Classroom discourse and the space of learning*. Mahwah, NJ; London: Lawrence Erlbaum Associates, Publishers.

McCracken, M., Kolikant, Y. B.-D., Almstrum, V., Laxer, C., Diaz, D., Thomas, L., Guzdial, M., Utting, I., Hagan, D., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills for first-year CS students. *Inroads - The SIGCSE Bulletin, 33*(4), 125-140.

McKenzie, J. A. (2003). *Variation and change in university teachers' ways of experiencing teaching.* University of Technology, Sydney, Sydney.

Means, H. W. (1988). A content analysis of ten introduction to programming textbooks. *SIGCSE Bull., 20*(1), 283-287.

Mellor, S. J., Clark, A. N., & Futagami, T. (2003). Model-driven development. *IEEE Software*, 14-18.

Mellor, S. J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA distilled: Principles of model-driven architecture*. Boston: Addison Wesley.

Meyer, B. (1997). *Object-oriented software construction* (Second ed.). Upper Saddle River: Prentice Hall PTR.

Meyer, J. H. F., & Land, R. (2003). *Threshold concepts and troublesome knowledge: linkages to ways of thinking practising within the disciplines.*

Meyer, J. H. F., & Land, R. (2005). Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education, 49*(3), 373-388.

Mezirow, J. (2000). Learning to think like an adult: Core concepts of transformation theory. In J. Mezirow & Associates (Eds.), *Learning as transformation: critical perspectives on a theory in progress* (1st ed., pp. 3-33). San Francisco: Jossey-Bass.

Mezirow, J., & Associates. (2000). *Learning as transformation: critical perspectives on a theory in progress* (1st ed.). San Francisco: Jossey-Bass.

Miller, J., & Mukerji, J. (2001). *Model Driven Architecture (MDA)* (No. ormsc/2001-07-01): Architecture Board ORMSC.

Miller, J., & Mukerji, J. (2003). *MDA guide version 1.0.1* (No. omg/2003-06-01): Object Management Group.

Morelli, R., & Walde, R. (2006). *Java, Java, Java: Object-oriented problem solving* (3rd ed.). Upper Saddle River: Prentice Hall.

Morrison, M. (2007). *Head first JavaScript*: O'Reilly.

Mugridge, R., & Cunningham, W. (2005). *Fit for developing software: Framework for integrating tests*. Upper Saddle River, NJ: Prentice Hall PTR.

Muller, O., Haberman, B., & Averbuch, H. (2004). (An almost) pedagogical pattern for pattern-based problem-solving instruction. *Inroads - The SIGCSE Bulletin, 36*(3), 102-106.

Murray-Harvey, R., & Keeves, J. P. (1994). *Student's learning processes and progress in higher education* (No. ED 374 703). New Orleans, LA: American Educational Research Association.

Neubauer, B. J., & Strong, D. D. (2002). The object-oriented paradigm: more natural or less familiar? *Journal of Computing in Small Colleges, 18*(1), 280-289.

Neuman, D. (1997). Chapter 5: Phenomenography: Exploring the Roots of Numeracy. *Journal for Research in Mathematics Education. Monograph, 9*, 63-177.

Newkirk, J. W., & Vorontsov, A. A. (2004). *Test driven development in Microsoft NET*. Redmond: Microsoft Press.

Nguyen, D. Z. (1998). Design patterns for data structures. *Inroads - The SIGCSE Bulletin, 30*(1), 336-340.

Nguyen, D. Z., & Ricken, M. (2006). Temperature calculator: Programming for change. *OOPSLA 2006 Educator's Symposium: Nifty Assignment* Retrieved 25 April, 2008, from http://www.cs.rice.edu/~mgricken/research/tempcalc/website/assignment.html

Nguyen, D. Z., Ricken, M., & Wong, S. (2005). Design patterns for parsing. *Inroads - The SIGCSE Bulletin, 37*(1), 477-481.

Nguyen, D. Z., & Wong, S. (2002a). *Design patterns for self-balancing trees.* Paper presented at the OOPSLA 2002, Seatle, WA.

Nguyen, D. Z., & Wong, S. B. (1999). Patterns for decoupling data structures and algorithms. *Inroads - The SIGCSE Bulletin, 31*(1), 87-91.

Nguyen, D. Z., & Wong, S. B. (2002b). Design patterns for games. *Inroads - The SIGCSE Bulletin, 34*(1), 126-130.

Nygaard, K. (1986). Basic concepts in object oriented programming. *ACM SIGPLAN Notices, 21*(10), 128-132.

Or-Bach, R., & Lavy, I. (2004). Cognitive activities of abstraction in object orientation: An emperical study. *Inroads - The SIGCSE Bulletin, 36*(2), 82-86.

Pang, M. F. (2003). Two faces of variation: on continuity in the phenomenographic movement. *Scandinavian Journal of Educational Research, 47*(2), 145-156.

Pang, M. F., & Marton, F. (2005). Learning theory as teaching resource: Enhancing students' understanding of economic concepts. *Instructional Science, 33*, 159-191.

Patrick, K. (2000). Exploring conceptions: phenomenography and the object of study. In J. A. Bowden & E. Walsh (Eds.), *Phenomenography* (pp. 117-136). Melbourne, Australia: RMIT University Press.

Patrick, K. A. (1998). *Teaching and learning: the construction of an object of study.* Unpublished Dissertation, University of Melbourne, Melbourne.

Pennington, N. (1987a). Comprehension strategies in programming. In G. M. Olson, S. Sheppard & E. Soloway (Eds.), *Empirical studies of programmers: Second Workshop* (pp. 100-113). Norwood, NJ: Ablex Publishing.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*, 295-341.

Pennington, N., Lee, A. Y., & Rehder, B. (1995). Cognitive Activities And Levels Of Abstraction In Procedural And Object-Oriented Design. *Human-Computer Interaction, 10*(2-3), 171.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1989). Conditions of learning in novice programmers. In E. Soloway & J. C. Sphorer (Eds.), *Studying the novice programmer* (pp. 261-279): Lawrence Erlbaum.

Perry, W. G., Jr. (1968). *Forms of intellectual and ethical development in the college years: A scheme*. Fort Worth: Harcourt Brace Jovanovich College Publishers.

Perry, W. G., Jr. (1988). Different worlds in the same classroom. In P. Ramsden (Ed.), *Improving learning : new perspectives* (pp. 145-161). London : New York, NY: Kogan Page ; Nichols Pub. Co.

Polanyi, M. (1958). *Personal knowledge: towards a post-critical philosophy*. Chicago: Routledge and Kegan Paul Ltd.

Polanyi, M. (1966). *The tacit dimension*. Gloucester, MA: Double Day and Company.

Post, E. (2000). *JADE for developers* (2nd ed.). Auckland: Addison Wesley.

Quatrani, T. (2000). *Visual modeling with Rational Rose 2000 and UML*. Reading, MA: Addison Wesley Longman.

Ramsden, P. (1988a). Studying learning: Improving teaching. In P. Ramsden (Ed.), *Improving learning : new perspectives* (pp. 13-31). London : New York, NY: Kogan Page ; Nichols Pub. Co.

Ramsden, P. (2003). *Learning to teach in higher education* (2nd ed.). London: Routledge Falmer.

Ramsden, P. (Ed.). (1988b). *Improving learning : new perspectives*. London : New York, NY: Kogan Page ; Nichols Pub. Co.

Richardson, J. T. E. (1999). The conceptions and methods of phenomenographic research. *Review of Educational Research, 69*(1), 53-82.

Robillard, P. N. (1999). The role of knowledge in software development. *Communications of the ACM, 42*(1), 87-92.

Robins, A., Haden, P., & Garner, S. (2006). *Problem distributions in a CS1 course.* Paper presented at the Proceedings of the 8th Australian conference on Computing Education (ACE2006).

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137-172.

Robins, A., Rountree, N., & Rountree, J. (2001). *My program is correct but it doesn't run: A review of novice programming and a study of an introductory programming paper* (Technical Report No. OUCS-2001-06). Dunedin: Department of Computer Science, University of Otago.

Ross, J. M. (2005). Polymorphism in decline? *Journal of Computing in Small Colleges, 21*(2), 328-334.

Rosson, M. B., & Carroll, J. M. (1996). Scaffolded examples for learning object-oriented design. *Communications of the ACM, 39*(4), 46-47.

Rountree, N., Rountree, J., & Robins, A. (2002). Predictors of success and failure in a CS1 course. *Inroads - The SIGCSE Bulletin, 34*(4), 121-124.

Rountree, N., Rountree, J., Robins, A., & Hannah, R. (2004). Interacting factors that predict success and failure in a CS1 course. *Inroads - The SIGCSE Bulletin, 36*(4), 101-104.

Runesson, U. (1999a). *The pedagogy of variation: Different ways of handling a mathematical topic.* Unpublished summary, Acta Univertsitatis Gothoburgensis, Göteborg.

Runesson, U. (1999b). Teaching as constituting a space of variation, *8th European Conference for Learning and Instruction*. Göteborg University, Göteborg, Sweden.

Runesson, U., & Mok, I. A. C. (2003). Discernment and question, "What can be learned?" In F. Marton & A. B. M. Tsui (Eds.), *Classroom discourse and the space of learning* (pp. 63-87). Mahwah, NJ; London: Lawrence Erlbaum Associates, Publishers.

Sajaniemi, J. (2002). Visualising roles of variables to novice programmers. In J. Kuljis, L. Baldwin & R. Scoble (Eds.), *14th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2002)* (pp. 111-127). Brunel University, London. From http://www.ppig.org/papers/14th-sajaniemi.pdf

Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., & Kulikova, Y. (2006). Roles of variables in three programming paradigms. *Computer Science Education, 16*(4), 261-279.

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education, 15*(1), 59-82.

Sajaniemi, J., & Kuittinen, M. (2007, 2-6 July). *From procedures to objects: What have we (not) done?* Paper presented at the 19th Workshop of the Psychology of programming Interest Group, University of Joensuu, Finland.

Sajaniemi, J., & Prieto, R. N. (2005, June). *Roles of variables in experts' programming knowledge.* Paper presented at the 17th Workshop of the Psychology of programming Interest Group, Sussex University.

Säljö, R. (1979a). *Learning in the learner's perspective: I: some commonplace misconceptions* (No. 76). Mölndal, Sweden: Institute of Education, University of Gothenburg.

Säljö, R. (1979b). *Learning in the learner's perspective: II: Differences in awareness* (No. 77). Mölndal, Sweden: Institute of Education, University of Gothenburg.

Säljö, R. (1988). Learning in educational settings: Methods of inquiry. In P. Ramsden (Ed.), *Improving learning: new perspectives* (pp. 32-48). London : New York, NY: Kogan Page ; Nichols Pub. Co.

Säljö, R. (1997). Talk as data and practice - a critical look at phenomenographic inquiry and the appeal of experience. *Higher Education Research & Development, 16*(2), 173-190.

Sandberg, J. (1997). Are phenomenographic results reliable? *Higher Education Research & Development, 16*(2), 203-212.

Scott, D., & Usher, R. (1999). *Researching education: Data, methods and theory in educational enquiry*. London and New York: Cassell.

Seale, C. (2003). *Social research methods: A reader*: Routledge.

Shalloway, A., & Trott, J. R. (2005). *Design patterns explained: A new perspective on object-oriented design* (2nd ed.). Boston: Addison Wesley.

Sheil, B. A. (1981). The psychological study of programming. *Computing Surveys, 13*(1), 101-120.

Shuell, T. J. (1986). Cognitive conceptions of learning. *Review of Educational Research, 56*(4), 411-436.

Sierra, K., & Bates, B. (2005). *Head first Java* (2nd ed.). Sebastopol, CA: O'Reilly Media Inc.

Sims-Knight, J. E., & Upchurch, R. L. (1998, 1998). *The acquisition of expertise in software engineering education.* Paper presented at the Frontiers in Education Conference.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850-858.

Soloway, E., Adelson, B., & Ehrlich, K. (1988). Knowledge and processes in the comprehension of computer programs. In M. T. H. Chi, R. Glaser & M. J. Farr (Eds.), *The nature of expertise* (pp. 129-152). Hillsdale, NJ: Lawrence Erlbaum Associates.

Soloway, E., & Sphorer, J. C. (Eds.). (1989). *Studying the novice programmer*: Lawrence Erlbaum.

Soroka, B. I. (2006). *Java 5: Objects first*: Jones and Bartlett Publishers.

Sprague, P., & Schahczenski, C. (2002). Abstraction the key to CS1. *Journal of Computing in Small Colleges, 17*(3), 211-218.

Stein, L. A. (1998). What we swept under the rug: Radically rethinking CS1. *Computer Science Education, 8*(2), 118-129.

Stein, L. A. (1999). Challenging the Computational Metaphor: Implications for How We Think. *Cybernetics and Systems, 30*(6).

Stein, L. A. (2003, 24 March 2004). Interactive programming in Java. Retrieved 28 September, 2004, from http://www.cs101.org/ipij/ObjectOriented/InteractiveProgramminginJava

Stolin, Y., & Hazzan, O. (2007). Students' understanding of computer science soft ideas: the case of programming paradigm. *SIGCSE Bull., 39*(2), 65-69.

Svensson, L. (1997). Theoretical foundations of phenomenography. *Higher Education Research & Development, 16*(2), 159-171.

Thomas, D. (2005). *Programming Ruby: The pragmatic programmer's guide* (Second ed.). Raleigh, NC, Dallas, TX: The Pragmatic Bookshelf.

Thomas, D., & Hansson, D. H. (2005). *Agile web development with Rails*. Raleigh, NC, Dallas, TX: The Pragmatic Bookshelf.

Thomas, L., Ratcliffe, M., & Thomasson, B. (2004). Scaffolding with object diagrams in first year programming classes: Some unexpected results. *Inroads - The SIGCSE Bulletin, 36*(1), 250-254.

Thompson, E. (1992). *CBC-PP100 programming principles workbook*. Auckland: Carrington Polytechnic.

Thompson, E. (2006, 27 December). What is a program(me)? Retrieved 27 December 2006, from http://www.teach.thompsonz.net/What_is_a_program/programmes.html

Thompson, E., Hunt, L. M., & Kinshuk. (2006a). Exploring learner conceptions of programming. In D. Tolhurst & S. Mann (Eds.), *Eighth Australasian Computing Education Conference (ACE2006)* (Vol. 52, pp. 205-211). Hobart, Tasmania, Australia: Australian Computer Society Inc. From http://crpit.com/confpapers/CRPITV52Thompson1.pdf.

Thompson, E., Whalley, J., Lister, R., & Simon, B. (2006b). Code Classification as a Learning and Assessment Exercise for Novice Programmers. In S. Mann & N. Bridgeman (Eds.), *The 19th Annual Conference of the National Advisory Committee on Computing Qualifications: Preparing for the Future — Capitalising on IT* (pp. 291-298). Wellington: National Advisory Committee on Computing Qualifications. From http://bitweb.tekotago.ac.nz/staticdata/allpapers/2006/papers/291.pdf.

Trigwell, K., & Prosser, M. (1996a). Changing approaches to teaching: A relational perspective. *Studies in Higher Education, 21*(3), 275-284.

Trigwell, K., & Prosser, M. (1996b). Congruence between intention and strategy in university science teachers' approaches to teaching. *Higher Education, 32*(1), 77-87.

Trigwell, K., & Prosser, M. (1997). Towards an understanding of individual acts of teaching and learning. *Higher Education Research & Development, 16*(2), 241-252.

Trigwell, K., Prosser, M., & Waterhouse, F. (1999). Relations between teachers' approaches to teaching and students' approaches to learning. *Higher Education, 37*, 57-70.

Tucker, A. B., & Noonan, R. E. (2007). *Programming languages: principles and paradigms* (2nd International ed.). Boston: McGraw Hill.

Uljens, M. (1996). On the philosophical foundation of phenomenography. In G. Dall'Alba & B. Hasselgren (Eds.), *Reflections on phenomenography - Toward a methodology?* (pp. 105-130). Göteborg: Acta Universitatis Gothoburgensis.

Ungar, D., & Smith, R. B. (1987). Self: The power of simplicity. In N. Meyrowitz (Ed.), *Conference on Object Oriented Programming Systems Languages and Applications* (pp. 227-242). Orlando, FL: ACM.

Ungar, D., & Smith, R. B. (2007, 9-10 June). *Self.* Paper presented at the History of programming languages, San Diego, California.

Upchurch, R. L., & Sims-Knight, J. E. (1998). *In support of student process improvement.* Paper presented at the Proceedings of Conference Software Eengineering Education & Training '98, Atlanta, Georgia.

van Roy, P., & Haridi, S. (2004). *Concepts, techniques, and models of computer programming*. Cambridge, Massachusetts, USA: The MIT Press.

VanDrunen, T. (2006). Java interfaces in CS 1 textbooks. In R. Mercer (Ed.), *OOPSLA 2006 Educators' Symposium*. Portland, Oregon, USA: ACM Press.

Ventura, P., & Ramamurthy, B. (2004). Wanted: CS1 students. No experience required. *Inroads - The SIGCSE Bulletin, 36*(1), 240-244.

Vilner, T., Zur, E., & Gal-Ezer, J. (2007). Fundamental concepts of CS1: procedural vs. object oriented paradigm - a case study. *Inroads - The SIGCSE Bulletin, 39*(3), 171-175.

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *ACM SIGPLAN OOPS Messenger, 1*(1), 7-87.

Weinberg, G. M. (1992). *Quality Software Management: Systems Thinking* (Vol. 1): Dorset House Publishing.

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, A., & Prasard, C. (2006, 16 - 19 January). *An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies.* Paper presented at the Eighth Australasian Computing Education Conference (ACE2006), Hobart, Tasmania, Australia.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies, 51*(1), 71-87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with computers, 11*(3), 255-282.

Wiener, R. (2007). *Modern software development using C#.NET*. Boston, MA: Thomson: Course Technology.

Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors, *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education* (pp. 184-188): ACM Press.

Wirth, N. (1976). *Algorithms + data structures = programs*. Englewood Cliffs NJ: Prentice Hall.

Wirth, N. (2002). Computing science education: The road not taken. *Inroads - The SIGCSE Bulletin, 34*(3), 1-3.

Wirth, N. (2007, 9-10 June). *Modula-2 and Oberon.* Paper presented at the History of programming languages, San Diego, California.