The Influence of Class Structure on Program Comprehension: An Empirical Study

Ahmed Alardawi

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

April 2013

Abstract

This thesis describes and reports on two sets of empirical studies investigating the ease of comprehension of Object Oriented (OO) programs, including the underlying various types of knowledge that can be present in the program text during the process of comprehension.

The two empirical studies are referred to as the Car and the Line-Edit. These are two well established programming problems in the early literature from the Psychology of programming research. Both novice and experienced OO programmers were asked to undertake comprehension tasks based on a paper and pen exercise and a set of comprehension questions associated with either an OO or a non OO programming version of the Car or the Line-Edit. The studies focus on the elements of class concept, problem characteristics, and solution decompositions and their effect on the comprehension of different types of knowledge which are present in the program text. It is found that OO programs are better understood than of the non OO programs. It is also found that the class concept, problem characteristics, and solution decompositions are empirically to be the influential elements in the comprehension of OO programs, especially for Control Flow, State, and Problem Classes types of knowledge.

An empirical grounded based model of OO program comprehension is proposed; the model forms a framework to the future empirical studies that focus on the critical aspects of the OO program comprehension. The thesis suggests a knowledge-based categorisation of the example programs. This categorisation should be embodied for better OO program comprehension amongst novices. The methodological issues for future investigations are also discussed. In particular it is suggested that different OO versions of the same program should be used as the experimental material as the next step.

Acknowledgements

I would like to thank Dr. Babak Khazaei and Prof. Jawed Siddiqi for their support, supervision, encouragement, and stimulating discussions. I believe it is fair to say that without their help and guidance it is likely that this thesis would never have been completed. It has been a pleasure working with you, gentlemen.

There are also several other individuals who have contributed, in one way or another, towards the body of research contained within this thesis. They are: Alla Elakari, Mohammed Jomma.

I also wish to thank researchers, who have been, or still are, PhD candidates within the C^{3}RI as well as many of the departmental staff, our friendly discussions helped me gain confidence in my research and sustain my attempt to complete it.

Dedication

To the soul of my father, Salem, and my mother, Badrea, for their never-ending love and prayers for my success and happiness.

To my wife, Nessrin, and my beloved children, Logien, Lama, Sanad and Adam for love, patience, sacrifices, and continuous support.

To all my brothers and sisters for their eternal faith in my ability to complete this thesis and for helping whenever they could

It is to them I dedicate this modest piece of work.

List of Publications

1. Empirical Study of Novices Comprehension of Object-Oriented Programs, Ahmed Alardawi, Babak Khazaei, Jawed Siddiqi (2010) In Proceedings of 6th Work-in-Progress Workshop of the Psychology of Programming Interest Group (PPIG), Dundee, UK, 7-8 January.
2. Influence on Novices of Class Structure on Program Comprehension, Ahmed Alardawi, Babak Khazaei, Jawed Siddiqi (2011a) In Proceedings of 7th Work-in-Progress Workshop of the Psychology of Programming Interest Group (PPIG), Sheffield, UK, 18-19 April.
3. Influence of Class Structure on Program Comprehension, Ahmed Alardawi, Babak Khazaei, Jawed Siddiqi (2011b) In Proceedings of the 23th Annual Workshop of the Psychology of Programming Interest Group (PPIG), York, UK, 6-8 September.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

In a society where considerable reliance is placed on computer software, it is vital constantly to improve software construction methods and practitioner skills, so that ultimately we are able to justify, and have confidence in, this reliance. The motivation and perhaps the ultimate goal of this thesis is to attempt to have a direct bearing on this continuing need for improved software. However, the immediate aim to which this thesis addresses itself is to contribute to the field of software engineering by improving our understanding of the program comprehension process. This investigation, in common with many that involve the study of human behaviour, is empirical in nature. It employs the established principle of such research known as the "scientific method", which consists of conducting empirical experiments to gather, evaluate, and interpret empirical evidence.

This chapter is structured as following: first, the cognitive benefits of Object Oriented (OO) programs and the suggestions from research on empirical evaluation of OO are highlighted. Secondly, the approach followed in this thesis to investigating OO program comprehension and the idea underpinnings this approach is introduced. The main thesis questions and aims are given then. Finally, the structure of the thesis is outlined.

There is a reasonable argument about the cognitive benefits of the Object Oriented (OO) approach; Briand et al. 1999 and Détienne 2006a are considered to be good sources for some of the claims. Briand et al. 1999 claimed that although some concepts of an OO approach (class, encapsulation, inheritance, client-server relationships, polymorphism, and decentralised architecture) have changed the nature of software development, these concepts have not brought the unconditional enhancements that were promised, for they have also set a number of new challenges from human factors as well as software engineering perspectives. It is likely that various OO concepts will show different advantages and drawbacks and there is a necessity to understand them better. Advocators of the OO approach (for example, Détienne, 2006a) are claiming that there is a direct correspondence between an OO approach and the nature of how people think about a computation problem. Therefore, the breakdown of a problem's entities into classes may be easier in the OO approach than by any other approach, and the mapping from problem domain to the program domain can be relatively easier and straightforward (Rosson, 1990; Borgida, 1985; Détienne 2006a). Taking a starting position that the OO approach is connected more closely to the problem domain, it has been thought that it might be of benefit not only in program design but also in program maintenance, comprehension, and reuse (Daly, 1996; Burkhardt et.al. 2006a, b).

Research on empirical evaluation of OO began to appear as early as 1995, for example, in the Special Issue of Human Computer Interaction on Object-Oriented programming, in which a number of empirical studies on OO technologies were undertaken (for example: Daly 1996; Corritore and Wiedenbeck, 1999; Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al, 1999; Harrison, 2000; Khazaei, 2002; Burkhardt et al. 2006a, b; Détienne, 2006a). There is a need to refocus some of the research on replicating prior studies across different programming approaches and environments, taking into account the methodological issues, limitations, and threats of these prior studies. These studies had also suggested some further research directions. For example: deploying different OO programming languages may differ in significant ways from the programming language used, determine the role-play in the comprehension of specific OO concept, determine the stages by which programmers develop more balanced what so called "mental representations" in OO approach, and determine the factors' effect on the comprehension. All these above have started to help in establishing a solid body of empirical knowledge from which general conclusions can be drawn. Most of these studies concentrate on program design and reuse, for example Détienne, (2006b) and Pennington, Lee and Rehder, (1995). However, the number of empirical studies of an OO approach is still comparatively limited compared to the rapid growth of this technology in both industrial and educational fields, especially empirical work on OO program comprehension.

Program comprehension is important, and yet difficult. It is an integral part of the programming process, playing a role in activities such as coding, debugging, and maintenance. Unfortunately, computer science students, especially those who take introductory programming courses, (hereafter referred to as novices) often find it extremely problematic to understand a program; the types of difficulty have been well documented (see Mayer, 1988 for a summary of some of these). Novices' problems may be compounded by the fact that comprehension *per se* is often not an explicit part of the curriculum. This may be because attempting to isolate the skill of comprehension and teach it directly can prove to be difficult, as there seems to be no universally agreed definition of what it is, and how it proceeds. This is unfortunate, as comprehension is an implicit first step in coding: learning a new programming approach almost inevitably starts with exposing novices to a short program (e.g. the ubiquitous 'hello world') and then writing similar programs. Thus, even before writing programs, novices must be able to understand them. Good (1999) claimed that, if teaching comprehension is difficult, a number of other techniques might be used to approach program comprehension in a more indirect way, for example, by choosing a programming approach which claims to make comprehension less painful, or by building learning environments to tackle program comprehension difficulties in novel ways. These are, however, not without their own problems; Ben-Ari (2001), for example, argued that many introductory programming courses teach programming starting with an OO approach. However, several programming teachers and educators argue it is impossible for novices to properly understand and use OO concepts without a viable understanding of fundamental programming concepts such as variables and assignment.

This thesis takes the view that novice program comprehension should be supported as a recognised activity rather than as a by-product of learning to program. It envisages an approach based on the combination of a number of external factors, many of which have been presented in some form in previous solutions to novice programmer difficulties. It is felt that a more detailed examination of the effects on comprehension on a programming approach, combined with a change in the conceptualisation of program comprehension itself, have implications for the ways in which novice problems can be addressed: by moving from a traditional "process" view to one based on information "knowledge" entities, novel types of program comprehension support can be envisaged. By combining this conceptualisation with an approach which takes into account potential novice difficulties with particular types of knowledge display, the characteristics of problems novices make use of as programming examples, and the type of solution to this particular problem to learning programming, this thesis lays the groundwork for an empirically grounded based model for OO program comprehension.

This thesis suggests a new approach to investigating OO program comprehension which is based on lessons learned from previous approaches. The proposed approach represents in many ways an ideal one. It relies on a number of as yet unsubstantiated suppositions about ways in which OO program comprehension might usefully be fostered. This thesis undertakes a detailed exploration of some of these hypotheses, looking at the elements influencing OO program comprehension and the ways in which they interact. The outcome of the thesis should inform the design of empirically grounded proposed model of OO program comprehension.

This thesis puts forward the idea of replacing process with types of knowledge. Many differences in existent program comprehension models are not so much related to the knowledge necessary for OO program comprehension, yet they do not incorporate knowledge that is considered important to OO program comprehension. They mostly relate to the process used in searching for this knowledge. Therefore, rather than focusing on the temporal aspects of program comprehension, one can focus on the entities thought to be involved in comprehension. It is postulated that the comprehension process can be conceived of as combinations of steps, where each step involves the search for a particular type of knowledge. Different comprehension directions/processes (e.g. top-down, bottom-up, and mixed) would therefore involve different combinations of steps, rather than trying to determine a fixed order on the sets of steps themselves. In other words, the focus should be on the product of each particular step, rather than the process which combines them. This implies a less prescriptive approach, which leads to the following research question: Can we teach novices about the different types of knowledge present in a program text, and how to allocate that knowledge, providing support for them as they do so, rather than limiting teaching to a single, invariant process?

Knowledge types are a way of describing different types of knowledge or information, as Pennington (1987a, b) called them, which are present in the program text, whose detection is necessary for program comprehension (Pennington, 1987a). In the non OO approach, these include such entities as function, data flow, control flow, etc. However, in the OO approach, Burkhardt et al., (2006a, b) have expanded the knowledge proposed by Pennington to include other knowledge that they consider more related to OO concepts, such as problem classes and the client-server relationship. Novices' support could be based on these types of knowledge, first by informing novices what they are, and secondly, by helping them to learn how to recognise them in program text. Despite their potential usefulness, unanswered questions remain, both on a theoretical and empirical level. In theoretical terms, both Pennington and Burkhardt et al. sought to embed knowledge types within a theory of program comprehension, based on Kintsch and van Dijk's theory of text comprehension (Kintsch and van Dijk, 1978; van Dijk and Kintsch, 1983). This thesis examines whether knowledge types, especially in the case of OO programs, can have a useful role out with this theory of comprehension. For an empirical perspective, work on knowledge types has focused on finding empirical evidence for the comprehension theory described above, and on uncovering the nature of programmers' understanding, which represents their comprehension of different sets of knowledge, (Pennington, 1987a, b; Corritore and Wiedenbeck, 1991; Ramalingam and Wiedenbeck, 1997; Corritore and Wiedenbeck, 1999;

Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al, 1999; Khazaei and Jackson, 2002; and Affandy et al 2011). Issues such as the influence of particular elements (in this thesis, the class concept, the problem characteristics, and solution decompositions) on the ease of comprehension of these different types of knowledge have not been investigated in detail.

1.1 Main Thesis Questions and Aims

The proposed model of OO program comprehension centres on activities such as searching for and identifying particular types of knowledge in a program. As such, the features of the base programming approach (in this case we consider the concept of class) may play a role in determining the ease (or difficulty) with which different knowledge can be comprehended compared to a program without this particular concept. The characteristics of the problem used and thus the possible solution decompositions derived and used in designing the program may also play a key role in comprehending these different types of knowledge. The thesis attempts to assess the ease of comprehension of OO programs for different problem characteristics that can possess different possible solution decompositions. This suggests that preliminary work should focus on issues such as the relationship between these elements and the comprehension of different types of knowledge. In light of this aim, the following lists the research questions explored in this thesis:

1. Are OO programs easier to comprehend than non OO programs? How to go about empirically investigate OO program comprehension?
2. What are the existing models of program comprehension? What are the main types of knowledge appropriate to investigate OO program comprehension?

3. How do different elements, such as, class concept, problem characteristics, and solution decomposition influence the comprehension of OO programs?

To effectively answer these research questions, the aims of this thesis are to:

- provide a rich view of the difference in the ease of comprehension between OO programs and non OO programs;
- evaluate current models of OO program comprehension;
- propose a new empirically grounded based model of OO program comprehension.

1.2 Outline of the Thesis

This thesis examines the role which different types of knowledge, hypothesised to be present in a program, play in program comprehension. It does that by examining the influence of class concept, the problem characteristics, and solution decomposition in the comprehension of object based programs versus non object based programs. In this thesis, Object based programs are defined as software programs written in OO way. This was done by mainly using of class concept. However, the non object based programs are defined as software programs written in non OO way. More precisely, the non object based programs are written without using of class concept. As such it does not make the claim that any of the hypotheses or findings described are in any way applicable to expert programmers. However, it considers both novice and experienced programming students. Furthermore, it considers primarily the notion of program comprehension rather than program design.

Chapter 2 provides an overview of the OO programming approach. It then reviews generic models of program comprehension, looking at how the best-known models of comprehension, based on their primary emphasis, fit into and might be relevant to OO program comprehension. It also reviews empirical literature in related empirical research with respect to OO program comprehension, and looks at the notion of program comprehension as derived from different types of knowledge, before going on to describe experiments which have looked specifically at these types of knowledge. Finally, it identifies various implications of using an OO programming approach for studying program comprehension.

Chapter 3 describes the research methodology, research design, and research procedures considered appropriate to the empirical work reported in this thesis. It examines methodologies used in empirical software engineering research and establishes a framework for conducting this investigation.

Chapter 4 discusses the rationale behind using specific settings of the investigation. It identifies important empirical issues that should be given more consideration by researchers in the design of comparable experiments.

Chapter 5 reports the design and conduct of a set of two studies (Car and Line-Edit) carried out using the tailored experimental methodology discussed in chapter 4. It also presents a statistical analysis of the studies' findings.

Chapter 6 includes an interpretation of the findings obtained in the studies described in Chapter 5 and evaluation of the program comprehension model used in this investigation. An empirically grounded based of OO program comprehension model is proposed. It then discusses methodological issues and the way in which they may affect the investigation's findings. Suggestions for several pedagogical issues to consider when teaching OO programming come at the end of this chapter.

The final chapter summarises the findings of this thesis. It states clearly the main contributions. It also summarises the main findings of the thesis and ends with suggestions for further work.

Chapter 2 Literature Review

2.1 Introduction

The purpose of this chapter is to provide information related to the main issues of this thesis. These issues are summarised in four major sections. The first section provides an overview of the OO programming approach and its associated concepts. It also reviews the cognitive benefits of OO approach and its related claims about the ease of comprehension of OO programs. The second section presents a brief discussion about the existent theories, strategies, and models of program comprehension in general and how they can be related specifically to OO program comprehension. The third section describes empirical work on OO program comprehension. It also discusses how elements of problem characteristics and solution decompositions can influence the comprehension of OO programs. Finally, the fourth section gives details about implications of the OO approach on different types of knowledge that are considered important to OO program comprehension. It then gives a research plan that to be followed in this investigation

2.2 Object-Oriented Approach and concepts

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' of 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 programmers to understand the intellectual issues that are involved in programming. Weinberg's statement provides some 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.

OO programming is described as a programming approach. A programming approach is defined in a number of different ways, each emphasising different aspects of the concept.

Ambler et al. (1992) describes a programming approach as:

> *"A collection of conceptual patterns that together model the design process and ultimately determine a program's structure." (p 28).*

In contrast, Stolin and Hazzan (2007) initially talk about approach in a generic sense that reflects the concept of approach. For their investigation into how the concept of a programming approach is understood, they 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).*

Programming approach would appear to be a way of thinking and constructing software solutions. Each approach will bring its own tools and techniques and its own way of thinking through how to construct software. Pfleeger (2010) defines an OO approach as:

"An approach to software development that organises both problem and its solution as a collection of discrete objects; both data structure and behaviour are included in the representation". (p 286)

She also identifies the OO approach by seven concepts: identity, abstraction, classification, encapsulation, inheritance, polymorphism, and persistence (Pfleeger 2010). These concepts have changed the nature of software development; however, Briand et al. (1999) argue that they have set a considerable debate about their appropriateness from both human factors and a software engineering perspective.

The concept of identity in an OO approach refers to the fact that data are organised into discrete, distinguishable entities called "class". A single class has states and behaviours associated with it. Class structure represents one of the essential concepts of the OO approach. Classes are program entities which integrate a structure defined by a type and functionalities. Class is a construct that is defined as a template used to instantiate objects of the class; these objects are instances of classes. Objects and classes will be used interchangeably throughout this thesis. Attributes and methods are defined for the entire class. A class is defined as a structure (a type) and a set of methods. A method is a function attached to a class that describes a part of the behaviour of the objects which are instances of this class (Détienne, 2006a).

Abstraction is essential for building any software system, whether it is OO or non OO. Pfleeger (2010) contends that abstractions in OO help to represent the different viewpoints incorporated in the system being developed. Together, the abstractions form a hierarchy that shows how different system entities relate to one another. Détienne (2006a) argues that abstraction is obtained by the means of encapsulation, polymorphism and late binding. Encapsulation means an object owns its data and methods. The data and methods are private and may be accessed and used by other objects only if the other objects send an appropriate message to the owner. The initiating object may send the same message to multiple objects which will act on it differently according to their own interpretations. This is the property of polymorphism, which aids abstraction by allowing messages to remain abstract. It is only during execution that the system decides which method will be executed according to the object with which the method is called. This property is referred to as late binding Pfleeger (2010).

Classes can accelerate software development by reducing redundant program code, testing and bug fixing. If a class has been thoroughly tested and is known to be a 'solid work', it is usually true that using or extending the well-tested class will reduce the number of bugs - as compared to the use of freshly-developed or ad hoc code - in the final output. In addition, efficient class reuse means that many bugs need to be fixed in only one place when problems are discovered. It has been asserted that the OO approach promotes reuse of software because the code is encapsulated into objects and the internal details of each object are hidden. The claim about reuse rests on an argument that hierarchies, which form the model of classes, are well-suited for reuse (Johnson and Foote, 1988). A programmer needs only to adopt a hierarchy appropriate for the domain of the problem, and then provide the specialisation needed for a particular problem by adding new low-level classes. Thus, much of the needed structure and functionality already present in the higher levels of the class hierarchy is automatically reused by inheritance Pfleeger (2010).

2.2.1 The Cognitive Benefits of Object-Oriented Approach

Advocates of an OO approach have made strong claims about what they called the 'naturalness', 'ease of use', and 'power' of this programming approach compared to the procedural programming approach (see for example: Meyer, 1988;Rosson and Alpert, 1990; Zhu and Zhou 2003; Détienne, 2006a, b).

Rosson and Alpert (1990) suggest that OO may be especially valuable in new domains or when practised by relatively experienced designers. However, in the procedural approach, problem decomposition is driven by generic programming constructs and specialised design knowledge. In terms of the concepts of problem and solution spaces introduced by Kant and Newell (2002), this implies that reasoning in the problem space is not separate from reasoning in the software solution space; thus, the objects are considered, but remain implicit.

Meyer (1988) does emphasise that objects can be picked directly from physical reality and modelled in software. However, the emphasis is on the way in which software addresses the needs of the problem domain through the development of a model that describes that domain. As such, the model becomes

> *"If you have a good model for describing the problem domain, you will find it desirable to keep 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 programming approach becomes a way of thinking about the problem domain. This idea is picked up by Quatrani (2003) who contends that

> *"Visual modelling is a way of thinking about problems using models organised around real-world ideas." (p 13)*

Concerning problem understanding, Meyer (1988) assumes that the identification of classes should be easy as they form natural representations of problem entities. Thus it seems particularly relevant to organise a model of design around a software representation of these classes. Therefore, we can say that in an OO approach, decomposing the problem into a solution consists of identifying the relations between objects and the associations between their structures and the functionalities.

Détienne (2006b) argues that mapping between the problem domain and the program domain should be more straightforward in an OO approach than in a procedural approach. She states that there is a direct correspondence between the OO approach and the way people naturally think about problem, therefore, decomposition of a problem into classes may be easier in the OO approach than by any other approach, and programmers can easily switch from the problem domain to the program domain. The theoretical argument in support of ease of mapping between domains in OO is that program classes are clear and visible entities in the problem domain. They are represented as explicit entities in the solution domain, and thus the mapping between the problem and solution domains is simple and clear. The domain objects are identified and used to structure the software system. Détienne (2006b) says that

> *"It has been suggested that OO design, in its initial phase, is based on understanding of the problem itself rather than on specialised knowledge of design, in contrast, procedural design solutions are structured by generic knowledge of programming rather than problem domain entities" (p 60).*

The activity of problem decomposition with OO languages is more likely to be derived from a designer's knowledge about the structure of the world than by knowledge about the design process or particular software design. However, in the case of the procedural approach, this activity is more likely to be derived from generic programming constructs and specialised design knowledge.

In their endeavour to propose an OO methodology as first taught, Zhu and Zhou (2003) argue that OO is not only a programming approach, but also a methodology that deduces from general concepts to the special and induces from the special to the general. These methods are similar to a human's natural thinking style. Therefore, its basic concepts (identity, abstraction, information hiding, encapsulation, and modularity) can be introduced as a very powerful methodology for both thinking and programming. The authors suggest that the general concept of OO can be effectively shown from a methodology viewpoint as following: everything in the world is an object. For example, in the real world, flowers, trees, and animals are objects; students and professors are objects; desks, chairs, classrooms, and buildings are objects; universities, cities, and countries are objects; even the world and the universe are objects. The second view is that: every system is composed of objects. A subject, such as electrical engineering, computer engineering, and history, is also an object. A cultural system includes history, language, food, costumes, relationships, and people etc. that are all objects; an educational system includes schools, students, professors, administrators, etc. that are also objects; an economic system includes objects like economic regulations, services, customers, and currency, etc.; a control system includes a plant to be controlled, a controller, sensors, actuators, and so on; a computer system includes monitor, keyboard, motherboard, CPU, memory, I/O devices, operating systems, and application software that are all objects. The third methodological perspective is that the development of a system X is caused by the interactions among the objects not only inside but also outside X. For example, a specific X Institute is a system, its development is caused by the interactions among students, tutors, staffs, and even government officers of the country who are outside X (Zhu and Zhou (2003).

Détienne (2006a) claimed that, since an OO approach is more intimately connected to the problem domain, it might be of benefit not only in program design but also in program comprehension. Ramalingam and Wiedenbeck (1999) stated that:

> *"In program design the problem is establishing mappings between real world entities and their representation in a program. In program comprehension the problem is making reverse mappings from the given program to comprehending of the real world entities and actions involved"(p 134).*

Moreover, Détienne (2006a) argues that classes allow a clear correspondence (mapping) between the model and the domain, making it easier to design, build, modify and even comprehend these models. Classes also provide some control over the often challenging complexity of such models. Computer programs usually model aspects of some real or abstract world (the Domain). Because each class models a concept, OO advocators argue that classes provide a more natural way to create such models. Each class in the model represents a noun in the domain, and the methods of the class represent verbs that may apply to that noun. For example, in a typical business system various aspects of the business are modelled, using such classes as *Customer*, *Product*, *Worker*, *Invoice*, *Job*, etc. An *Invoice* may have methods like *Create*, *Print* or *Send*; a *Job* may be *Performed* or *Cancelled*, etc. Once the system can model aspects of the business accurately, it can provide users of the system with useful information about those aspects. However; a number of different difficulties and negative effects of other approaches are associated with OO design activity, such as difficulties associated with process of class creation, difficulties in articulating declarative and procedural aspects of the solution (e.g. the hierarchy of classes and the main procedure), misconceptions about some fundamental OO concepts, and the transfer effect associated with shifting from traditional design to OO design. Détienne (2006a) provides a comprehensive survey of these difficulties).

Rist (1996a) claimed that the structure of an OO system is built around its control flow, data flow, and class encapsulation. This system is created by the interaction between the plans and objects in the system. He has defined a plan as a set of actions that, when placed in a correct order, achieves some desired goal. Applying this to OO systems, the actions in a plan are encapsulated in a set of routines, and the routines are divided among a set of classes and connected by control flow. He claims that there is an orthogonal link between plans and objects in OO; he contends that:

> *"Plans and objects are orthogonal, because one plan can use many objects and one object can take part in many plans. The code that executes a plan will thus be spread over several classes." (p 555).*

This orthogonality allows the structure of a system to be shown as a lattice of nodes, where a node marks the intersection between a plan and an object. Rist argues that this can also be considered as a reflection of the real world, where a plan can use many objects (i.e., a plan for a cake uses flour, eggs, water, and so on), and an object can be used in many plans (i.e., an egg can be used to make a cake, omelette, soufflé and so on).

There is a wealth of literature that takes for granted that the OO approach entails a greater focus on the problem, thus it should facilitate not only problem decomposition but also program design and thus program comprehension. However, other studies argue that the OO approach does not support the claims mentioned and argue in favour of the procedural approach being natural. Neubauer and Strong (2002) stated that

> *"It is true that the world around us consists of objects which possess attributes and have behaviours. But good object-oriented programming does not depend so much upon the identification of objects (as challenging as that can be) as upon the ability to grasp complex patterns of interactions among many objects. While it is possible to anticipate and run many scenarios through an object-oriented application, a complex application has a very large number of possible states. At some point these complex systems begin to exhibit unanticipated behaviour." (pp 284-285).*

It seems clear that the ease of comprehension of OO programs is mainly based on the nature of the problem domain "problem characteristics", and solution decompositions. However, little empirical evidence exists in supporting this claim. There is a need to assess the claim about the ease of comprehension of OO programs on the basis of these elements. In terms of the problem domain, the claim cannot be valid for all types of problems, where the classes are not clear and visible in the problem domain. The overall aim of this thesis is to focus on assessing the ease of comprehension of OO programs for different problem types that can possess different possible solution decompositions.

2.3 Program Comprehension

This section provides a description of program comprehension and considers the importance of program comprehension in the field of software engineering.

2.3.1 What is the Definition of Program Comprehension?

Paradoxically, literature on comprehension tends not to define comprehension explicitly, perhaps because it seems so naturally obvious, in the same way that research papers on reading do not begin by asking what reading is. However, it may be because comprehension, like reading, covers a wide range of activities, with subtle differences between them.

From the perspective of Biggerstaff et al, (1993), a programmer starts to construct an understanding of an unknown software system. He/she is creating an informal, human oriented expression of computational intent. The creation of this expression happens through a process of analysis, experimentation, guessing and puzzle-like assembly. When it comes to a definition of what program comprehension means, we follow the explanation by Biggerstaff et al. (1993):

> *"A person understands a program when able to explain the program, its structure, its behaviour, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program." (p 482)*

Pennington and Grabowski (1990) have also offered a more precise description of the program comprehension task:

"Understanding a program involves assigning meaning to a program text, more meaning than is literally `there`. A programmer must understand not only what each program statement does, but also the execution sequence (control flow), the transformational effects on data objects (data flow), and the purposes of groups of statements (function). In order to do this, the programmer will employ a comprehension strategy that co-ordinates information `in the program text` with the programmer's knowledge about programs and the application area. This results in a mental representation of the program meaning." (p 54)

From these descriptions, one possible reason why a general definition of comprehension is not forthcoming is that program comprehension is a highly individual activity. Therefore, the scope of this activity varies from person to person. It can involve an attempt to comprehend an entire program in detail, looking for a specific piece of information in the program, or acquiring a general overview of the program. Since program comprehension is an individual activity, persons who are carrying out the activity could have different levels of previous experience from different domains. Moreover, a person attempts to comprehend program code for a particular programming task (i.e., maintenance, debugging, transferring some aspect of the program to another person, etc.). Since the nature of the task varies, they will necessarily have different requirements in terms of the type and the amount of information extracted from the program and the way it is combined. Comprehension activity also can involve experience about programming domains with very different programming approaches. Each programming approach should have its own characteristics which could largely affect the comprehension process.

Numerous attempts have been made to derive a theory of program comprehension from various points of views. Good (1999) contends that these views vary, focusing on the processes which arise when people try to comprehend programs, the kind of prior experience they have about programming, the mental representations they construct during the comprehension process, and the role of information the programmer extracts from the program. The author adds that program comprehension can be seen as multi-faceted, encompassing different aspects. There is a need for a generic definition of program comprehension. This definition can serve as a framework to position theories and research on program comprehension, and therefore will lead to better understanding of the relationships between these various aspects.

Good (1999) provides a notable generic model with a definition of program comprehension:

> *"Given a program in a particular language, program comprehension is a process in which the programmer uses prior knowledge about programming and information present in the program to form a dynamic, evolving model of the program which can then be applied to a task." (p 14)*

She highlights a number of interrelated entities and processes, which are central to theories of program comprehension. These entities represent: comprehension processes and strategies, programming knowledge, mental models of the program, information contained in the program, and purposes of the comprehension.

Despite all attempts to provide a useful description of program comprehension and highlight entities and processes that form theories and models of program comprehension, they lack generalisation in terms of different programming approaches. These attempts were carried out largely in the context of procedural and imperative programming approaches and do not highlight the major aspects and concepts of other programming approaches, especially the OO approach. Since the descriptions provided are approach-dependent and our research interests lie in the OO approach, there is a need to either generalise these descriptions, or to provide an OO-related description. This OO description could outline entities and processes that take into account OO concepts to form a theoretical framework of OO program comprehension.

2.3.2 Program Comprehension from Different Perspectives

Program comprehension could be considered as a necessary prerequisite and plays a key role in several programming tasks. Works by Pennington, 1987a; Corbi, 1989; von Mayrhauser and Vans, 1995; Spinellis, 2003, and Zaidman, 2006 suggest that more than half of software engineers' task time is spent in program comprehension activity; more precisely, in maintenance tasks which require a certain level of insight into the application to be maintained. Von Mayrhauser and Vans (1995) have made a compilation of software maintenance-specific scenarios in which program comprehension is a necessary prerequisite activity in all maintenance tasks. Table 2.1 provides an overview of these maintenance tasks.

Table 2.1: Tasks and activities requiring code understanding (source, Von Mayrhauser and Vans, 1995)

Maintenance tasks	Activities
Adoptive	Understand system. Define adaptation requirements. Develop preliminary and detailed adaption design. Code changes. Debug. Regression tests.
Perfective	Understand system. Diagnosis and requirements definition for improvements. Develop preliminary and detailed perfective design. Code changes/additions. Debug. Regression tests.
Corrective	Understand problem. Understand system. Generate/evaluate hypotheses concerning problem. Repair code. Regression tests.
Reuse	Understand problem, find solution based on close fit with reusable components. Locate components. Integrate components
Code leverage	Understand problem, find solution based on predefined components. Reconfigure solution to increase likelihood of using predefined components. Obtain and modify predefined components Integrate modified components

Being aware of the fact that program comprehension is a prerequisite in all software evolution and maintenance tasks, improving the efficiency of program comprehension activity will lead to a significant overall efficiency gain. Table 2.1 also shows clearly the link between various programming tasks and program comprehension.

Shneiderman and Marey (1979) categorise the tasks of programming as composition, comprehension, debugging, maintenance, and learning. The learning task could reasonably cover the first four tasks as each includes skills to be learnt. Learning how to comprehend a program is an open question in the software engineering community.

Good (1999) has also highlighted a number of factors which may have caused the lack of instruction in program comprehension. Firstly, many theories and models of program comprehension do not take into account the fact that program comprehension is multi-faceted. They try to propose an invariant model for comprehending an entire program while they are taking a single activity. Additionally, the issue of novice/expert variation is ignored in comprehension models; this difference plays a role in the suitability of the comprehension process the model provides. Von Mayrhauser and Vans (1995) point out that prior knowledge is an important factor in applying some comprehension processes, top-down process in particular, and so would not even be available to novices, at least at the very beginning stage of comprehension. However, in the OO case the situation may be different: the representation of the class, for instance, could facilitate comprehension for novices (Détienne, 2006a).

Despite the above points, considering program comprehension as a process, and describing this process in terms of a set of steps would be more valuable. However, taking into account issues of variability and appropriateness of the process for an explicit programming approach, it may seem sensible to consider comprehension at a lower level of granularity rather than trying to explain the whole process of comprehension from start to end. This makes a comprehension model more widely applicable, possibly introducing a more flexible approach to program comprehension that serves an explicit programming approach.

2.3.3 Cognitive Theories of Program comprehension

Francois Détienne represents a good example of a comprehensive survey of the history of cognitive models of program comprehension and sets of empirical experiments over the past forty years (Détienne, 2002). She delves back to a time in the early 1970s when the research lacked a theoretical framework to underpin the evaluation of the software tools. Story (2005) added:

> *"it was neither possible to understand nor to explain to others why one tool might be superior to other tools" (p 188).*

All these lacks, however, have made it difficult to explain the mechanism of an effect. In other words, cognitive models, which provide a richer explanation of the processes of how an effect works, were not applicable. The research of program comprehension started to borrow related theoretical frameworks from other areas of research, such as text comprehension, problem solving, and education (Storey, 2005). Using these theoretical underpinnings, this period is characterised by the development of cognitive theories that give a rich explanation of the way in which programmers comprehend software programs. The perceived benefits of these would lead to more efficient and enriched theoretical frameworks that provide rich and comprehensive descriptions of program comprehension activity (Détienne, 2002; Storey, 2005).

As preliminary, this thesis will recount a brief survey of the different theoretical frameworks of comprehension of natural language text, as these frameworks represent the underpinning theoretical frameworks of program comprehension. This will enrich the readers' view about program comprehension models and where they were derived from.

Following this section, a description of the ways in which program comprehension can be categorised will be discussed. The aspects forming this discussion are:

- program comprehension as a temporal activity
- program comprehension as strategies for deployment
- program comprehension as a construction of different knowledge categories

2.3.4 Program Comprehension as a Temporal Activity

This section describes theories of program comprehension, many of which are based on exploratory experiments; some of these have been empirically validated - von Mayrhauser and Vans (1995) and Storey (2005) provide a comprehensive survey of these theories. Good (1999) describes these theories as "temporal theories"; she assumes that all these theories primarily focus on the idea of a sequence which the programmer takes to comprehend a program, and the progression direction in which they carry out this sequence. She also reports that this sequence is primarily related to levels of abstraction and varies based on the direction of the comprehension process (i.e., top-down, bottom-up, and mixed). These models will also be discussed in terms of their appropriateness to OO programs.

2.3.4.1 Top-Down Models of Program Comprehension

There are numerous theories that consider program comprehension to be top-down. The most notable one is the behavioural theory of program comprehension by Brooks (1983). He postulates that comprehension is the reverse process to coding: while coding involves mapping from the problem domain, possibly through several intermediate domains, into the programming domain, comprehension is process of reconstructing different knowledge about the programming domain and mapping it to the problem domain. This mapping occurs through mainly hypothesis-driven processes and involves different but closely related intermediate knowledge domains; the type of knowledge is primarily based on the nature of the program under consideration. Brooks also assumes that the mental model is built by successively generating and refining hypotheses and by forming subsidiary hypotheses in a hierarchical top-down, depth first manner. Several knowledge domains will act as cues and contribute to refining and evaluating these subsidiary hypotheses, by searching for *beacons*, until they can be matched to specific code in the program text. Brooks has referred to beacons as:

> *"Sets of features that typically indicate the occurrence of certain structures or operations within the code." (p 548)*

Verification or rejection of these subsidiary hypotheses depends heavily on the absence or presence of beacons. The process will be repeated until the code is understood entirely.

Soloway and Ehrlich (1989) report that a top-down comprehension process typically applies when the code is familiar and therefore the constructed mental model will contain a hierarchy of goals and plans. A top-down comprehension process presumes progression from a high level of abstraction to a lower level of generation, refinement, and finally verification, of the hypotheses. Storey (2005) also argued that this nature of direction requires a good level of experience with the program code and therefore, this model will be inappropriate for novices. However, there is no empirical evidence about the inappropriateness of this model for novices in the case of OO. Theoretically, in modern OO programming languages - with different hierarchical structure of the classes, the levels of abstractions, and comparatively readable programming languages' syntax - a top-down model could be assumed to be appropriate for novice programmers and possibly invoked early in the comprehension process. Therefore, there is a need to investigate this assumption.

2.3.4.2 Bottom-Up Models of Program Comprehension

The idea of bottom-up theories of program comprehension assumes that comprehension occurs initially from the lower-level abstractions of the program to the higher-level abstraction based on the program text. Shneiderman and Mayer (1979) is the most cited article in the literature as the first of the bottom-up theories. They postulate that programmers start from a low level by reading code statements and then mentally group or chunk these statements into a high level abstraction. A high-level comprehension of the program is achieved by further combining these chunks. This theory mainly focused on the structure of the programmers' knowledge, by differentiating between syntactic and semantic knowledge of programs, than on the order in which the comprehension process occurs. The theory describes syntactic knowledge as language dependent, concerning the statements and basic units in a program, while semantic knowledge is language-independent and is built in progressive layers until a mental model is formed which describes the problem domain (Shneiderman and Mayer, 1979). In the case of OO programs, the claim about closer mapping between program domain and problem domain in OO could also be assumed to facilitate the mental model construction.

Pennington's adoption theory of program comprehension focuses on describing the general characteristics of the bottom-up model (Pennington, 1987a, b). She derived an adopted comprehension model of program comprehension from a prior theory of text comprehension put forward by Kintsch and van Dijk (1978) and van Dijk and Kintsch (1983). Pennington hypothesises that the comprehension process results in the production of two distinct but interrelated representations of the text, the Program Model and the Situation Model. These two sub-models represent different abstract views of the program text. She empirically observed that the program model is first built based on a procedural reading of the program text. The programmer first develops a control flow abstraction of the program which captures the sequence of operations in the program. This program model is developed through the chunking of microstructure levels in the program text, which represent the program's statements, control construct, and relationships, into macrostructure levels, which represent text structure abstractions, by making inferences about these microstructures. Once a program model has been fully assimilated, a situation model is developed. The situation model encompasses knowledge about data flow abstraction and abstraction of functional relationships between domain objects. The programmers' comprehension is further enhanced through the cross referencing of the artefacts in the program model and situation model. The model's knowledge categories are fully described in Section 2.3.6.1

2.3.4.3 Integrated Model of Program Comprehension

This theory postulates that comprehension is built at several levels of abstractions simultaneously by switching between different comprehensions processes (Storey, 2005). The theory centres on the idea of combining four major sub-models during the comprehension process:

- The **top-down model** is usually invoked and developed when the programming language or program code is familiar to the programmer. It incorporates domain knowledge as a starting point for formulating hypotheses. However, in the case of OO programs, this model may be invoked in the early stages.

- The **program model** is a control flow abstraction, invoked when the code is completely unfamiliar.

- The **situation model** describes data flow and functional abstractions in the program. It is usually developed after a partial program model is formed.

- The **knowledge base** represents the programmer's current state of knowledge, usually consisting of the information needed to build the above three models, and is used to store new and inferred knowledge.

Von Mayrhauser and Vans (1995) and Storey (2006) argued that, while the first three sub-models are involved in constructing an internal representation of the program and the strategy deployed in this construction, the fourth supplies the related preferred knowledge to the corresponding process involved. As a result of this combination, a mixed model of program comprehension becomes necessary, especially for large size systems. Von Mayrhauser and Vans (1995) also assume that, during the comprehension process, any of the first three sub-models might be activated by the programmer. For example, while constructing a program model, a programmer might identify a beacon representing a common task such as sorting. This will cause a jump to the top-down model. The programmer then generates sub-goals and will search for clues to support these sub-goals. If the programmer finds a section of unrecognised code during this search, the programmer returns to constructing a program model. Structures built by any one sub-model component are accessible by the other two, but each sub-model component has its own preferred knowledge types.

2.3.5 Program Comprehension as Strategies for Deployment

Littman et al. (1987) link the strategy deployed in program comprehension to the breadth of familiarity with program text gained by the programmer during comprehension activities. The authors observed that there are two types of strategy that are deployed by programmers in the context of a maintenance task: a "*systematic*" strategy, and an *"as-needed"* strategy. In the *systematic* strategy, the programmer attempts to read the code in detail, gaining a broad understanding of the entire program by tracing it through the control flow and data flow abstractions in the program before carrying out modifications. On the other hand, in the *as-needed* strategy, the programmer minimises the amount of code to be comprehended and focuses only on the part of the code where a modification should be made. The authors argue that the *systematic* strategy is more successful in modification as it increases the ability to detect interactions between the code central to modification and code elsewhere in the program. However, the problem with *as-needed* strategy is that the modification may have side effects on other parts of the program which the programmer might not anticipate. Thus the programmer tends to apply *as-needed* strategy in the case of maintenance rather than modification tasks.

Littman et al. (1987) also found that there is a direct influence between the strategy used by a programmer and the knowledge gleaned and stored in the programmer's mental model. For example, programmers using *systematic* strategy acquired both static knowledge, which concerns objects, actions and functional components of the program, and dynamic knowledge, which describes the interactions between functional components in the program when it is executed. This strategy enables programmers to form a strong mental model. However, programmers deploying the *as-needed* strategy only acquired static knowledge, resulting in a weaker mental model of how the program worked.

Although the strategy deployed during the comprehension process is task- and knowledge-dependent, it seems that there is an influence between the strategy deployed and the programming approach of the program under comprehension. Burkhardt et al. (2006a, b) argued that, in the case of an OO approach and despite the task being performed - whether modification or maintenance - programmers may deploy both *systematic* and *as-needed* strategies interchangeably to comprehend the whole program. The emphasis on classes representing program entities may encourage use of an *as-needed* strategy to understand the static aspects of the program classes and what classes do through their structure and behaviour/functions, and a *systematic* strategy to know about the dynamic aspects of these classes and how they do it. Whether novice OO programmers are able to deploy these strategies interchangeably during the comprehension process is, however, still open to question.

2.3.6 Program Comprehension as a Construction of Different types of Knowledge

Mental model approaches of program comprehension represent interesting theoretical frameworks to study program comprehension. This section describes works which have focused on the way in which a particular program is mentally represented by the programmer. Many of the works on mental representations are empirically based: a number of empirical studies have aimed to elicit programmers' mental representations of a program, sometimes at distinct intervals (e.g. before and after modification/debugging), and to characterise its structure and contents. Two of the most empirically validated models, Pennington (1987a) and Burkhardt et al. (2006a, b) are described below.

2.3.6.1 Pennington's Model of Program Comprehension

Pennington's model of program comprehension (Pennington, 1987a) is considered to be the most useful existing framework that explains the mental model and has been extensively empirically investigated in various programming approaches (For example, Ramalingam and Wiedenbeck 1997; Wiedenbeck et al. 1999; Wiedenbeck and Ramalingam 1999; Harrison et al. 2000; Khazaei and Jackson 2002, Affandy et al. 2011). The idea of the Pennington model centres on distinguishing between two different mental representations which may be built while comprehending a program: (1) the situation model, which is equivalent to van Dijk and Kintsch's (1983) situation model and reflects entities of the problem domain and their relationships, and (2) the program model, which is equivalent to van Dijk and Kintsch's (1983) propositional textbase and reflects the textbased representation of the program. Pennington argues that the model is built on five different types of knowledge, divided between program and situation models. Table 2.2 represents the correspondence between text relations, knowledge structures, mental representation, and model related in Pennington's program comprehension mental model.

Table 2.2 : Correspondence between text relations, knowledge structures, mental representation, and model in Pennington model of program comprehension (source Pennington, 1987a)

text relations	**knowledge structures**	**mental representation**	**model**
elementary operations	text structure knowledge	dynamic and functional views	program model
control flow	text structure knowledge	dynamic view	program model
function	plan knowledge	functional view	situation model
data flow	plan knowledge	dynamic and functional view	situation model
state	plan Knowledge	dynamic and functional view	program/situation model

In this table, text relations refer to abstractions of the program text. Knowledge structures refer to relatively generic knowledge stored in long-term memory that must be activated to be used. Mental representation refers to the content of working memory at a particular point in the comprehension activity, constructed from activated knowledge in long-term memory, the results of prior comprehension episodes, and external information gathered from the environment. The following is an explanation of text relations, which are defined later as knowledge categories:

- **elementary operations** form part of the text microstructure, and constitute basic text units usually consisting of one or few lines of code. The feature of this category is that it is directly available in the source code;
- **control flow** forms part of the text microstructure, constitutes the links between text units, which in the simplest case are sequential or in complex situations involves looping or calls to subprograms; thus this category is procedural in nature;
- **functions** explain the goal of the whole program, what the program accomplishes in terms of the problem situation it addresses. Function information expresses what the program does in terms of entities, relationships, and actions in the world; this information is not usually directly available in a program text, but must be inferred from the program text in combination with knowledge of the real-world problem domain of the program;
- **data flow** relates to Communication between variables, corresponding to data flow relationships connecting units of local plans within a routine and also changes that occurs to data variables while they pass through the program. The transformations of the data are, thus, at the heart of whatever useful action a program achieves. For this reason, data flow

information is considered to be very closely related to a program's functions and goals and forms a part of the situation model;

- **state** comprises the state of all aspects of the program at the time a given action occurs in a program.

In testing her model, Pennington's methodological approach was to give participants a program to read for a limited time and then ask them comprehension questions reflecting different information/knowledge categories presumed to make up the program and situation models. The correctness of the responses served as an indicator of the nature of their representations.

2.3.6.2 Limitations of Pennington's Model for OO programs

Thompson (2008) argued that, at simple OO program level, the Pennington model's knowledge categories may have some meaning, but the more the classes, and their related interactions, are used in the program, the more difficult it would be to determine whether these flows actually occur. This may be reflected and supported in the many of empirical results reported from related empirical studies that attempted to empirically validate Pennington's model in the case of OO programs (for example, Ramalingam and Wiedenbeck 1997; Wiedenbeck et al. 1999; Wiedenbeck and Ramalingam 1999; Khazaei and Jackson 2002).

Some previous empirical studies have applied Pennington model to investigate the effect of programming approaches on the construction of mental representations e.g. imperative, procedural, event-driven, and OO. Some studies argue that the model cannot be applied to OO approach. Sajaniemi and

Kuittinen (2007, 2008) argue that Pennington's model is more related to the procedural nature of the languages used by her and to the small size of the program she used. Burkhardt et al. (2006a, b) claims that Pennington's model has several limitations with relation to an OO approach. The authors explained this limitation in three main points.

Firstly, the model lacks key characteristics of OO concepts; in more detail, it does not examine representations about classes and objects, or even data structures. Since objects are central entities in OO programs, construction of representations of objects should be taken into account in a model of OO program comprehension, Burkhardt et al. assume that the representation of objects is part of the situation model inasmuch as it reflects the objects of the problem situation.

Secondly, Pennington's model accounts for comprehension of relatively small programs but does not scale up easily to larger programs. She does not account her model for the representation of delocalised plans. This is considered as an important aspect of OO approach. Pennington assumes that the reader uses plan knowledge to construct the situation model, thus that plan representations of a program are primarily based on data flow. Soloway et al. (1982) assume that programmers have knowledge about patterns of program instructions which typically go together to accomplish certain functions or goals. However, in the case of large programs, particularly in OO programs, it happens that many plans are delocalised. Thus the actions in a plan are encapsulated in a set of routines, and the routines are divided among a set of classes and connected by control flow. Détienne (2006a) claims that this can reflect the real world, where a plan can use many objects, and an object can be used in many plans. In the introduced OO model of program comprehension by Burkhardt et al (2006a, b), they take the view that the construction of these complex delocalised plan representations is primarily based on client-server relationships knowledge, in which one object processes and supplies data needed by another object.

Finally, Pennington also accounts for the representation of elementary operations as part of the text microstructure and the control flow between these operations at the level of program model. However, the macrostructure of large programs, consisting of the representation of larger text units such as routines, is not accounted for in her model. Burkhardt et al. (2006a, b) argued that the representation of the macrostructure is based on the elementary functions of the program model.

In accordance with all three points mentioned above, Burkhardt et al. (2006a, b) have proposed a new OO mental model that takes into account all the limitations highlighted in Pennington's model.

2.3.6.3 Burkhardt et al.'s Model of OO Program Comprehension

In pursuit of identifying the mental representations constructed by OO programmers, Burkhardt et al. (2006a, b) developed a model of OO program comprehension based on the mental model approach (see table 2.3). The OO model adopts and expands Pennington's model to take into account OO concepts such as classes, message passing, and the structure of larger programs. To achieve this goal, knowledge related to objects as well as client-server relationships between objects was added to expand the situation model.

Knowledge about objects and goals represents the static aspects of the problem solution, whereas knowledge about data flow and client-server relationships represents more dynamic aspects of the solution to the problem. Knowledge about macrostructure, which represents larger text units such as routines attached to objects, is incorporated into the program model.

Table 2.3 : Correspondence between text relations, knowledge structure, mental representation, and model in Burkhardt et al.'s mental model of OO program comprehension. (Source: Burkhardt et al. 2006b).

text relations	knowledge structures	mental representation	model
Program model			
control flow	text structure knowledge	dynamic view	program model
elementary operations	text structure knowledge	dynamic and functional views	program model
elementary functions	text structure knowledge	dynamic and functional views	program model
situation model			
Static Aspects of Situation Model			
problem classes	problem knowledge and plan knowledge	object view	situation model
relations between problem classes	problem knowledge and plan knowledge	object view	situation model
computing or reified classes	generic programming knowledge and plan knowledge	object view	situation model
main goals	problem knowledge and plan knowledge	functional view	situation model
dynamic aspects of situation model			
client–server	plan knowledge (complex delocalised plans)	dynamic and functional views	situation model
data flow	plan knowledge	dynamic and functional views	situation model

The following is an explanation of each knowledge category used in this OO model as stated by Burkhardt et al (2006b):

- **elementary operations** form part of the text microstructure, constituting basic text units, usually consisting of one or a few lines of code;
- **control flow** also forms part of the text microstructure; control flow constitutes the links between text units. Control flow, at this fine level of granularity, represents the control structure (sequence, loop or test) linking individual operations within a routine;
- **elementary functions** consist of larger units of text, and thus form part of the text macrostructure. These functions correspond to units in the program structure, i.e., routines attached to objects;
- **problem objects** directly model objects of the problem domain;
- **relationships between problem objects** consist of the inheritance and composition relationships between objects;
- **computing or reified objects**: An example of a computing, or reified, object is a string class, which is not a problem domain object per se. Reified objects are represented at the situation model level inasmuch as they are necessary to complete the representation of the relationships between problem objects, i.e., they bundle together program-level elements needed by the domain objects;
- **main goals** of the problem correspond to functions accomplished by the program viewed at a high level of granularity. They do not correspond to

single program units. Rather, the complex plan which realises a single goal is usually a delocalized plan in an OO program;

- **client–server relationships:** Communication between objects corresponds to client–server relationships in which one object processes and supplies data needed by another object. These connections between objects are the links connecting units of complex delocalized plans. In an OO program, the actions in a complex plan which perform a main goal are encapsulated in a set of routines, and the routines are divided among a set of classes and connected by control flow. Client–server relationships represent those delocalized connections;

- **data flow relationships**: Communication between variables correspond to data flow relationships connecting units of local plans within a routine.

Having comprehensively surveyed the OO approach, its claims, and the various theories of program comprehension, in the next section we turn to question what evidence there is to support either the claims or the validity of the proposed models.

2.4 Empirical Works on OO

The major aims of empirical studies of OO software development and evolution are to investigate the effectiveness of an OO approach and to evaluate the quality of OO software products. Briand et al. 1999 stated that:

> *"The overall objective of empirical studies of object-oriented technologies and products is to gather tangible evidence about its properties and gain deeper insights into the nature of the object-oriented paradigm and its relationship to other approaches." (p 394)*

This will in turn help to provide a scientific foundation to the engineering of OO software. A considerable number of empirical studies of the OO software development and maintenance have been involved with developing and assessing quality models of OO software. The goal is to relate structural attribute measures intended to quantify OO concepts to external quality indicators, such as development time, reusability, maintainability, and comprehension. Thus, this would significantly help in technology assessment and comparison. However, there is a gap in various aspects of empirical studies of OO software development to effectively answer some research questions. A very comprehensive survey of this can be found in Briand et al. 1999.

2.4.1 Empirical Works in OO Program Comprehension

Research on an OO programming approach began to appear as early as 1995, in which numbers of empirical studies on the OO approach had been undertaken. This section describes a series of empirical studies based on the framework for determining mental representation developed by Pennington (1987a, b) and adopted by Burkhardt et al. (2006a, b); both models were detailed in the previous section. The idea of these models centres on classification of different knowledge categories, and the distinction between two sub-models, mainly, the program model, and the situation model. Most of the studies applied these sub-models primarily in areas of comparing comprehension of program text written in different programming approaches, and investigating the influence of certain factors such as expertise, tasks (i.e. modification, reuse, maintenance, documentation), and development time on the constructed mental model and thus on comprehension. It has been found that different programming approaches have different effects on the mental representation constructed during comprehension. Although the overall aim of most of these studies was to investigate the nature of the mental representations held by programmers during comprehension. The work reported in this thesis looks specifically at assessing the ease of comprehension of OO programs. More precisely, the investigation uses the concept of mental representations in the form of sets of different types of knowledge categories. Also, the investigation does not use what we called "two-stage" model to distinguish between the two models (program and situation) models mentioned above. Rather it considers the comprehension of each type of knowledge individually. This approach will help in highlighting the most important knowledge, thus incorporating these types of knowledge in a new proposed model of OO program comprehension. In order to do so, the empirical works in this thesis have borrowed, thus tailored, from the methodology used in previous empirical studies. Therefore, it is useful to illustrate these studies.

Pennington (1987a) carried out two experiments to validate her model. She gave subjects relatively short programs to read in a limited, premeasured, time and then asked questions reflecting different categories of knowledge considered. In a first experiment involving 80 professional programmers, she found that after reading a program, whether written in COBOL or FORTRAN, subjects respond better and faster to questions about elementary operations and control flow knowledge than questions about program goals and data flow knowledge. This analysis thus tends to show that, during the process of comprehending the program, representations of the program model are constructed first and the representations of the situation model emerge later. Although the results provide some experimental support for the approach, its generality is in question because the programs used were so short (15 lines) (see Pennington, 1987a).

In the second experiment, 40 professional programmers were involved, and the programs used were longer (200 lines) than in the first experiment. The experimental procedure comprised two phases: the first phase was the same as in the first experiment, i.e. reading the program and answering questions. In the second phase, subjects were asked to modify the program and were then again asked questions. The results of the first phase were the same as in the first experiment. However, different response patterns were observed in the results of the second phase. Answering questions related to program goals and data flow were improved and even exceeded previous results in questions related to control flow. One interpretation is that a situation model can be developed over time, or the nature of the given task (in this case, modification) can also effect the dominant representation constructed or an earlier construction of the situation model.

Pennington concluded that the knowledge related to a program model tend to be initially more available, emerges first to the subjects, but knowledge related to a situation model grow as subjects work longer with a program, emerges later, and is based on parts in the program. However, certain factors, such as the task and given time, can facilitate earlier construction of the situation model.

In endeavouring to compare novices' comprehension in different programming approaches, Ramalingam and Wiedenbeck (1997) carried out an empirical study focusing on how different programming approaches, in this case imperative versus OO, affects the construction of mental representations of novices. 75 novice OO programmers participated in the experiment. Results show that error rate, represented by the percentage of error responses to the questions, was higher on OO programs than imperative programs. Thus the two approaches differ in the nature of the mental representation formed during the comprehension process. The authors claim that OO subjects formed a strong situation model, while the imperative subjects formed a strong program model. The authors also go on to claim that an OO programs are easier to comprehend. They say:

> *"This research suggests that the OO style facilitates the mapping from the program to the domain for novice programmers working on small and simple programs. This may be because there is more explicit and salient domain-related information in the OO style programs than in the imperative style programs" (p 134).*

While the evidence is certainly intriguing, there are several reasons why the claim seems over-confident and thus the study was criticised. Given the higher overall error rate of OO subjects, it seems unlikely that OO programs are easier to comprehend. Additionally, Good (1999) claimed that the data comes from one source only: binary choice questions, as no program summaries were collected during the experiment.

Wiedenbeck, Ramalingam, Sarasamma, and Corritore (Corritore and Wiedenbeck, 1999; Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al, 1999) have completed a series of studies similar to Pennington's second study. They endeavoured to compare mental representations constructed by OO programmers and procedural programmers. Two of these studies included novice programmers (Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al, 1999). Pennington's model, with its associated knowledge categories (Pennington 1987a, b), was used in these sets of studies. The results showed that the distributed nature of control flow and "hidden" actions (e.g. constructor or destructor calls) made it more difficult for novices to comprehend OO programs than of a corresponding procedural programs. However, the class structure of the OO programs made it a bit easier to comprehend program entities. Also program goals and data flow issues were easier to comprehend from a procedural program.

Considering a simple OO program level, Wiedenbeck and her colleagues argued that Pennington's knowledge categories may have some meaning in OO programs. However, the more that advanced OO concepts, such as composition, inheritance and polymorphism, are 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, round 55% correct" (p 274).*

They contend that the procedural programmers performed better in all knowledge categories when working with larger procedural programs. These results may be more a reflection that these types of questions have more meaning for procedural programs than for OO programs rather than an indication of a greater difficulty of comprehension in OO programs. This is acknowledged to some extend by Wiedenbeck et al.:

> *"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." (pp 259-260)

The difficulty is that this research does not seem to have considered how OO programs differ from procedural programs when it comes to the knowledge categories that involved in forming the mental representations. Rather Wiedenbeck et al. are assessing whether those programmers who are familiar with the OO approach perform with similar characteristics to those doing procedural programming. This makes some of the conclusions questionable when Wiedenbeck 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)

They added:

"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. Certainly there is much to learn in OO programming and it may take longer for a beginner to gain a comparable level of skill." (p 277)

Wiedenbeck et al. (1999) failed to elicit a rich view of the mental representations of the OO subjects. In order to obtain such a view, knowledge categories that do not have a direct counterpart in procedural programming approach should be investigated. For example, the authors failed to ask questions specifically about the static and dynamic aspects of classes. This is considered as one of the essential differences between OO and non OO approaches. Thus, they do not have direct evidence about whether the subjects gained a good mental representation of the attributes of objects.

Khazaei and Jackson (2002) have also conducted an empirical study endeavouring to investigate program comprehension differences between Event-Driven (ED) and OO approaches for novice OO programmers. 40 postgraduate computer science students were participated in the experiment. They tailored the experiment of Wiedenbeck et al (1999) towards an investigation of ED and OO approaches. Interestingly, results show that knowledge related to a situation model was more available than that related to a program model in both approaches. The authors agreed that the comprehension of both approaches had "*a lot in common*". They also reported that the Pennington model of program comprehension still has some limitations in the case of these two programming approaches. Authors have raised call for further work to substantiate their findings and to address advanced OO concepts such as inheritance and polymorphism.

Burkhardt et al. (2006a, b) were the first to introduce the OO model of program comprehension. Their study aimed to evaluate the effect of three factors (programmer expertise, programming task, and the development of comprehension over time) on program comprehension. 51 participants (30 OO experts and 21 OO novices) were recruited for the experiment. The experimental materials consisted of a university database program for the documentation task and a library problem, introduced by Wing, (1988), for the reuse task. The library problem was partially isomorphic to the database problem and allowed for reuse by inheritance or by template copying and modification. The materials were large enough to take full advantage of OO concepts of classes, encapsulation, inheritance, composition of classes, function overloading, operator overloading, and polymorphism. The study found that the expertise factor played a role only in the documentation task but not in the reuse task. It was also found that the expertise of programmers affected only the construction of the situation model in the documentation task. The authors argued that the reuse task appears to entail a decrease of expert/novice differences as concerns the construction of a situation model. Thus novices are capable of building a situation model if they are given a task that requires situation knowledge.

Although Burkhardt et al model of OO program comprehension incorporates most important OO concepts; the authors highlighted some limitations to their model. They raised two theoretical questions in this context. The first concerns the suitability of the model to describe a situation model in the case of complex OO programs and the difficulties associated with extending the situation model to distinguish between multiple levels of abstractions. The authors presume that developing more empirical studies of large OO programs, by moving from the study of what they called "*programming-in-the-small to programming-in-the-large*", and by integrating theoretical evidences made in these two branches of the field, could probably provide an answer to this question. The generality of the proposed comprehension model is set as a second theoretical question. The authors expect the type of knowledge to be represented in the situation model regardless of whether the notation of the specific language is similar, in particular static and dynamic, referring to objects and plan knowledge. However, they expect knowledge related to the program model will be notation-dependent.

Replicating the experiment across different OO programming languages and different problem domains will most likely help in answering this question.

Affandy et al. (2011) conducted an empirical study with 294 novice students in an introductory programming course. The study followed Pennington's methodology and aimed to address novices' problems in dealing with tracking the logic flow and writing simple program code. The study found that overall students' tracing skills were poor. More specifically, students lacked knowledge in comprehending the dynamic behaviour of the program. However, they were able to master the static part of programming knowledge. The authors concluded that students' ability to trace a program becomes one of the factors that are related to the ability to solve problems and the ability of problem-solving contributes to programming skills. The study attempted to propose a model to shift the internal working memory load of students through integrated visualisation tools. These tools can work as a learning aid by revealing the dynamic behaviour of programs and related concepts that appear in each level of program abstractions. The authors reported that developing learning aid tools with such complexity can be used to help students with different learning strategies to comprehend the essentials of programming.

Although empirical studies into the psychology of programming have raised call for further empirical research in OO program comprehension (for example, Sajaniemi and Kuittinen 2007, 2008; Briand et al 1999), the work reported above are relatively old in terms of empirical studies of software engineering. This is an indicator of the progress (or lack if it) that has been made during this time.

Regarding the shift to an OO approach in education, Lister et al. (2006) reported that this shift is not motivated by the psychology of programming or computer science education research. Thus there is practically no empirical evidence that would indicate that such a shift is desirable, or even effective, for learning programming. Therefore, further research needs to be conducted to identify what types of knowledge, which represent the mental representations, are used by OO programmers.

Sajaniemi and Kuittinen (2007, 2008) contend that the cognitive consequences of the shift to OO had not been studied before the shift, and only superficially even after it. The authors argued that the assessment used for the claims about the ease of comprehension of OO programs has remained constant despite a change in programming approach, and further empirically based researches needs to be conducted to identify what mental representations are used by OO programmers. Moreover, the authors argue that most researchers introduce various pedagogic techniques and tips, such as visualization tools or curriculum changes, without consideration for educational or psychological theories (see for example Cooper et al., 2003; Bierre et al., 2006; Kölling & Henriksen, 2005).

Rist (1996b) argues that studies in OO do not support claims about the natural way of conceptualising real-world problems. Moreover, he suggests that OO programming adds the complexity of class structure to a procedural system. Therefore, OO educators should not think that OO is particularly easy for students, or that using OO from the very beginning relieves educators from teaching procedural programming issues.

From all the above, it seems difficult to assess whether the OO approach is easier "natural" way of conceptualising and modelling a real world situation. In their state-of-art review of the psychology of OO programming education, Sajaniemi and Kuittinen (2008) mentioned that studies into OO programming are still few and the results mentioned make it clear that both the OO approach itself and learning an OO programming approach are very different from their imperative and procedural counterparts: mental representations of programs are different; problems used have different roots; conceptual contents of knowledge are different; the level of understanding the underlying notional machine is different; and the overall approaches to program design and program comprehension are completely different. These differences are so fundamentals to learn that it is daring to claim that the classic educational and cognitive results of novice imperative and procedural programming should be used in the OO context. Furthermore, the number of cognitive empirical studies about OO programming is small. In the context of program comprehension, the authors argued that if the measures used for verifying mental representation do not accurately reflect OO mental representation then these types of claims are difficult to support. The authors state 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 add:

> *"There are practically no theories on the development of programming skills or comprehension of programming concepts in the OO case." (p 87)*

The authors have presented a research agenda intended to improve the understanding of OO approach. As a result, there is a need to know more about OO mental representations with respect to OO programming, the cognitive development of the approach and novices' and experts' program comprehension processes.

2.4.2 Problem Characteristics and Solution Decomposition in OO Program Comprehension

Example programs in textbooks are the most important tools that play an important role in learning and teaching programming. They also work as a reference for how to solve specific programming problems. However, it is difficult to find or develop examples that are fully faithful to all principles and guidelines of the OO approach and also follow general pedagogical principles and practices. Börstler et al. (2010, 2011) claimed that there are no existent systematic evaluations of textbook examples. In the literature the terms 'problem type', 'problem characteristics', 'example' and 'example program' are used interchangeably. In the context of this thesis, we define problem characteristics as a specification of the given programming problem and the example program as a complete solution of the given problem. Both terms will be used interchangeably throughout this thesis.

Börstler et al. (2010) claimed that the strength of OO is in managing complexity. The authors stated that:

> *"In fact, the strength of object-orientation is in managing complexity. Kristen Nygaard, one of the originators of object-orientation, often stressed that object-orientation is a better problem solving tool for complex problems than for simple ones." (p 128)*

They have identified a number of basic properties that example programs must include to meet the requirements of an effective educational tool. These are: technically correct, readable, valid role model for an OO program, promote "object-oriented thinking", and emphasise programming as a problem solving process. The authors also warn that the cognitive load of the students should be carefully controlled. This control can be achieved through focusing on simple concepts within the example, upholding and enforcing the principles of OO, and keeping the example small and easy to understand, thus within the cognitive load of the students. All above alarming those high-quality OO program examples are a prerequisite for successfully learning OO programming. In their attempt to evaluate example programs for introductory programming courses, Börstler et al. (2010) proposed an evaluator instrument. The authors argued that their quality instrument is highly reliable and measures aspects of quality that are not captured by common size or complexity measures. The study results show that the quality of many examples is not as high as one would expect to find in an introductory programming text. In particular, many examples received low ratings for "object thinking" and reasonable state and behaviour quality factors. Whether the examples used in related studies that assess novices' OO program comprehension meet the requirements as an effective educational tool is an important question.

Daly (1996) claimed that not all problems types are well suited to an OO approach. There is no clear classification of the scope and the characteristics of the problems used in learning OO programming. The author argued that whether the characteristics of the problems are example programs that can be used to fulfil certain properties to be effective as an educational tool in teaching OO programming is not clear in the literature.

The literature of OO design contributed to propose a typology of problems for OO. Hoc (1981) proposed a framework for classifying problems. Two dimensions are distinguished: procedural versus declarative, and prospective versus retrospective. These distinctions have been made in the context of a procedural approach. However, whether the same dimensions are relevant and influence the choice of design strategy in an OO approach is an important question.

Détienne (2006b) suggested that the distinction between declarative versus procedural problems is more relevant for an OO approach. She hypothesised that declarative problems would be easier to solve than procedural problems in an OO approach. However, the hypothesis was not confirmed. She found that experienced OO designers tended to use a declarative plan whatever the problem type.

Chatel and Détienne (2007) revealed the importance of the typology of design problems. However, the critical question is which dimensions of problems are relevant for determining the correspondence between the problem characteristics and the programming approach. The authors discussed the typology of problems regarding OO. They proposed a new problem dimension for an OO approach. They assumed that the solutions in an OO approach should consider not only the structure of the solution, objects and procedures, but also the way objects communicate within this structure. They classified the problems into two dimensions: problems with a hierarchical structure of classes with vertical communication among objects, and problems with a flat structure of classes with horizontal communication among objects. The authors observed that, for the former type of problem, expert OO designers used a declarative plan, and objects and functions guided solution development. OO experts used a procedural plan for the latter type, and dynamic characteristics of the procedure guided solution development. However, the study does not consider novice OO designers. Certainly, it could be argued that the OO approach encourages the development of hierarchical solution structures rather than flat solution structures; however, it is likely, especially for large scale software development, various kinds of plan may be used for developing parts which have different structures.

In the context of program comprehension, an important question is whether problem characteristics form an important element that influences the comprehension process. One of the critical questions that this thesis approaches is which problem dimension is relevant for determining the correspondence between problem characteristics and possible solution decompositions, and program comprehension. This question has not been addressed especially in the context of OO program comprehension.

There is a lack of literature of program comprehension on how a problem characteristic and its possible solution decompositions, as elements, affect program comprehension. However, the focus on these elements was in the sense of the program being developed rather than the program being comprehended (Siddiqi, 1984). Empirical studies carried out on OO program comprehension have considered the variations in control flow and data flow as major elements in evaluating the ease of comprehension of OO approach (for example, Ramalingam and Wiedenbeck 1997; Wiedenbeck, et.al 1999; Wiedenbeck and Ramalingam 1999; Khazaei and Jackson 2002; Burkhardt et al 2006a, b; Affandy, et al., (2011). There has been no explicit distinction made between the characteristics of the problems used and their solution decompositions in these studies. More precisely, the role of problem characteristics and the role of solution decompositions were not clearly addressed. This can be seen from the data produced that were almost considered as one set. The only notable common criterion in selecting problems used in these studies was their familiarity with the domain knowledge.

Thompson (2008) reported that it is unclear from this field what distinguishes the OO approach from other approaches from the point of view of problem characteristics and solution decompositions. These elements seem to be paid less attention in this field. In this investigation, using different problems with different characteristics was an attempt to produce empirical evidence emphasising how problem characteristics and solution decompositions might affect OO program comprehension. This in turn would lead to proposing a new categorisation of problems that would help to improve OO program comprehension. For example, if any study relies only on problems that are more amenable to OO comprehension, this will most probably affect its validity. Thus, to identify characteristics of problems in which their solution decompositions are better comprehended, one possible research direction is using problems with significantly different characteristics. In this context, this thesis used a variation in problem characteristics and a variation of solution decompositions as a basis in assessing the ease of comprehension of OO programs, where each OO program is an implementation of a different problem which possesses different solution decompositions.

2.5 Implications of the OO Approach on Different Types of Knowledge

The use of an OO programming approach for a program comprehension study has various implications. Firstly, the fact that it differs substantially from a procedural programming approach in terms of the nature of the constructed mental models means different types of knowledge are made more salient (or conversely, are obscured). Secondly, the claims for and against the ease of comprehension of OO programs will lead to different predictions about the pattern of the knowledge one might expect to observe: these are considered in turn below. These predictions will be discussed in this section. It should also be mentioned that this discussion will specifically focus on empirical evidences. Before doing so, however, it should be noted that most specific claims tend to refer to program design and construction rather than program comprehension, even though most authors also contend that OO programs are easy to comprehend. Additionally, many of the claims are quite general and slightly vague; they do not map directly onto different types of knowledge, and therefore some speculation is called for. The rest of this section reviews empirical results found about the comprehension of OO programs with respect to different types of knowledge. These are: elementary operations, control flow, data flow, program goals, state, and problem classes. These types of knowledge were found in Pennington and Burkhardt models of program comprehension. They are considered important to the investigation carried out in this thesis. Thus, they form the model of program comprehension used in this investigation. This review could help in building predictions about the comprehension of these types of related knowledge.

Comprehension of elementary operations knowledge did not differ significantly between OO and non OO programs. According to Pennington (1987a), this is due to programmers focusing only on small segments of code (one line or a few lines of code). Also, this expected similarity in comprehension could refer to the fact that this related knowledge is more language dependent. Pennington also argued that the other reason for using elementary operations questions is that they act as warming-up questions that orient programmers to a specific part of the program text.

In terms of comprehension of control flow knowledge, it is expected to find comprehension of control flow knowledge difficult in the OO programs, given that control flow knowledge is explicitly available in the program text. In most prior related empirical studies, it was found that the comprehension of control flow knowledge was more difficult in OO programs (Ramalingam and Wiedenbeck 1997; Wiedenbeck and Ramalingam 1999; Wiedenbeck et al., 1999). This difficulty could refer to the way programmers tend to read the program. These studies argued that programmers, especially novices, tend to read the program line-by-line rather than in execution order, thus the sense of temporal ordering of actions in OO programs is lossless. This in turn makes the comprehension of control flow knowledge difficult. Wiedenbeck et al., (1999) found that determining where the top of the program is and comprehending the calls through successive layers of decomposition is easier in non OO programs. In OO programs there is no top level of the program. Rather, parts of a task are distributed across objects which pass messages to other objects to act on their behalf. This "non-hierarchical" interaction of objects may make comprehension of control flow knowledge more difficult in OO programs. They added that difficulties due to the disparity of different control structures, the partially implicit nature of control flow in OO programs, and the introduction of messages passing among objects negatively affect comprehension of control knowledge.

Regarding data flow and program goals knowledge, it could be hypothesised that comprehension of data flow and program goals knowledge would be relatively easier in OO programs than non OO programs. This can be attributed to the presenting of OO programs in chunks of code represented by objects. Wiedenbeck and Ramalingam (1999) argued that, in OO programs, programmers learn to give more attention to the class declarations, to get an overview of the objects manipulated in the program, the data elements making up the objects, and the functions carried out. This makes tracking data transformations across an OO program's parts less difficult than in an equivalent non OO program. This appears to aid novice OO programmers to better comprehend these types of knowledge.

With respect to the nation of state knowledge, it could be expected that comprehension of state knowledge would be found difficult in OO programs. Wiedenbeck (1997) and Wiedenbeck argued that state knowledge is not directly highlighted in program text. Due to the independent nature of an OO program's parts and the unordered sequence of the program's actions, it is unclear how comprehension of state knowledge category will differ between OO and non-OO programs. However, Khazaei and Jackson (2002) found comprehension of state knowledge easier in OO programs. They argued that spotting state knowledge from the program text was relatively easier. This contradiction will make it difficult to predict the comprehension of state knowledge.

For the problem classes knowledge, it could be also expected that comprehension of knowledge related to problem classes will be relatively easy. The expectation was based on empirical results found in Burkhardt et al.'s (2006a, b) studies, where problem classes knowledge was the most easily comprehended knowledge. It could be claimed that comprehension of this knowledge is more dependent on the problem characteristics, specifically the tangibility of the problem entities. Obviously, if the problem entities are physically tangible and already exist in the real world then highlighting these entities will be more straightforward. On the other hand, the claim is vague if the problem's entities are intangible and do not directly exist in the real world.. Thus, it is unclear what comprehension of this knowledge category would be in problems where entities are relatively intangible. Prior related empirical studies did not show comprehension of problem classes knowledge (for example, Ramalingam and Wiedenbeck, 1997; Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al., 1999; Good, 1999; Khazaei and Jackson, 2002). There is a lack of empirical evidences in these related studies about comprehension of the objects used and their associated attributes and functions.

Table 2.4 summarises the predictions of comprehension of the above mentioned types of knowledge in OO and non OO programs.

Table 2.4 Predictions about the comprehension of different types of knowledge categories in OO and non OO programs

knowledge categories	**predictions of comprehension**
elementary operations	comprehension is similar between OO and non OO programs
control flow	comprehension is difficult in OO programs than of non OO programs
data flow and program goals	comprehension is easier in OO programs than of non OO programs
state	comprehension is difficult in OO programs than of non OO programs. However, empirical evidences are contradict
problem classes	comprehension is easier in OO programs than of non OO programs. However, there is a lack of empirical evidences

In terms of OO concepts, the investigation was only limited to assess the ease of comprehension of the concept of class structure and concepts such as inheritance, polymorphism etc were not considered due to subjects' availability. To better assess the claim about the ease of comprehension of OO programs, the reset of this section gives the research plan followed in this investigation. Pennington's (1987a) and Burkhardt et al.'s (2006a, b) models of program comprehension were taken and their methodology was tailored to meet the purposes of the investigation. For this, there was a necessity for a corresponding shift in emphasis away from memory-based tasks used in related empirical studies (Ramalingam and Wiedenbeck, 1997; Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al., 1999; Khazaei and Jackson, 2002), to a search-based task which required searching through the program text for the necessary knowledge It is also considered worth assessing the ease of comprehension at the types of knowledge level (elementary operations, control flow, data flow, program goals, state, and problem classes) rather than comprehension at the overall program level. The investigation mainly focused on how elements of "*class concept*", "*problem characteristics*" and "*solution decompositions*" can influence the comprehension of these types of knowledge. Additionally, the investigation did not follow the approach of distinguishing between the development of the two sub-models (program model and situation model) as in prior related empirical studies. Rather, we were interested in assessing the comprehension of each type of knowledge individually. We believed this would be more fruitful in evaluating, thus, enhancing the model of program comprehension used in this investigation. Therefore, we were led to propose empirically grounded based model of OO program comprehension.

2.6 Chapter Summary

This chapter reviewed a generic model of program comprehension and considered a number of theories of program comprehension and empirical work in relation to these models. It reviewed research related to empirical work comparing program comprehension of different programming approaches. It described the reasons why the idea of mental representations, which is represented in different types of knowledge, might be useful in this context, and highlighted unanswered questions. It then considered the implications of an OO approach on each type of knowledge. Finally, it called for assessing the ease of comprehension of each type of knowledge and how elements of *class concept*, *problem characteristics* and *solution decompositions* would influence the comprehension of these types of knowledge.

Chapter 3 Research Methodology

3.1 Introduction

This chapter gives an overview of the methodology used in this investigation. It explains the philosophy of the research, and the research method considered appropriate and adopted in this thesis. It examines methodologies used in empirical software engineering researches and establishes a framework for conducting two sets of empirical studies. This chapter also outlines methodological issues that affect conducting these studies, such as choosing subjects, materials, and measurement, and how these issues are tailored to enhance the reliability and validity of the conducted studies. Hypothesis testing, statistical methods of analysis and ethical issues are then outlined.

3.2 Philosophy of Research

Having specified the research questions in Chapter One, it is worth considering what to accept as valid answers to these questions from data generated from the fieldwork. Different researchers make different assumptions about scientific truth. These different assumptions reflect major differences in philosophical stances and opinions about the nature of truth and how researchers arrive at it through scientific investigation (Easterbrook et al., 2008). Because of the differences in philosophical assumptions, underlying philosophical stances are critical for determining the methodological approach to research design. Each of the philosophical stances provides a framework based on ontological and epistemological assumptions about the nature of reality and the way in which information should be organised to explain reality (Jonker and Pennink, 2010).

The ontological assumptions underlying scientific paradigms are concerned with the nature of reality. They describe the nature of human knowledge and how we obtain it. Prior to considering a methodology, the researcher makes an assumption concerning the nature of reality that influences the way in which reality can be understood. Ontology can be normative, which suggests that reality is objective and the same in all situations, or interpretive, which suggests that reality is subjective and differs depending on situations (Grix, 2010). Epistemology involves the way in which information or knowledge is organised to provide an understanding of reality. It describes the nature of the world irrespective of our attempts to understand it. The ontological assumption determines the epistemology used in research because information or knowledge has to be organised in a manner that is consistent with the assumption concerning the objectivity or subjectivity of reality (Grix, 2010).

Creswell (2011) characterises four dominant philosophical stances (positivism, interpretivism, critical theory, and pragmatism). The stance adopted by a researcher determines which research methods the researcher believes lead to acceptable evidence in response to stated research question(s). Being explicit about the stance also helps when undertaking research. It might not be possible to convince other people to change their philosophical stance, but it will be possible to argue persuasively for why certain methods were chosen.

Positivism states that all knowledge must be based on logical inference from a set of basic observable facts. Positivists are reductionist, in that they study things by breaking them into simpler components. This corresponds to belief that scientific knowledge is built up incrementally from verifiable observations, and inferences based on them. Positivism has been much attacked over the past century due to doubts about the reliability of observations of the world, and the complication that scientific "fact" built up in this manner sometimes turns out to be wrong. While positivism still dominates the natural sciences, most positivists today might more accurately be described as post-positivists, in that they tend to accept the idea (due to Popper) that it is more productive to refute theories than to prove them, and we increase our confidence in a theory each time we fail to refute it, without necessarily ever proving it to be true. Easterbrook et al. (2008) argued that positivists prefer methods that start with precise theories from which verifiable hypotheses can be extracted and tested in isolation. Hence, positivism is most closely associated with controlled experiments. However, survey research and case studies are also frequently conducted with a positivist stance.

Interpretivism, also known as constructivism (Klein and Myers, 1999), rejects the idea that scientific knowledge can be separated from its human context. In particular, the meanings of terms used in scientific theories are socially constructed, so interpretations of what a theory means are just as important in judging its truth as the empirical observations on which it is based. Constructivists concentrate less on verifying theories, and more on understanding how different people make sense of the world and how they assign meaning to actions. Theories may emerge from this process but they are always linked to the context being studied. Constructivists prefer methods that collect rich qualitative data about human activities, from which local theories might emerge. Constructivism is most closely associated with ethnographies, although constructivists often use exploratory case studies and survey research too (Easterbrook et.al, 2008).

Critical Theory judges scientific knowledge by its ability to free people from restrictive systems of thought (Calhoun, 1995). Critical theorists argue that research is a political act, because knowledge empowers different groups within society, or entrenches existing power structures. Critical theorists therefore choose what research to undertake based on whom it helps. They prefer participatory approaches in which the groups they are trying to help are engaged in the research, including helping to set its goals. Critical theorists therefore tend to take emancipatory or advocacy roles. In sociology, critical theory is most closely associated with Marxist and Feminist studies, along with research that seeks to improve the status of various minority groups. In software engineering, it includes research that actively seeks to challenge existing perceptions about software practice, most notably the open source movement and, arguably, the process improvement community and the agile community (Easterbrook et al., 2008). Critical theorists often use case studies to draw attention to things that need changing.

Pragmatism acknowledges that all knowledge is approximate and incomplete, and its value depends on the methods by which it was obtained (Peirce and Menand, 1997). For pragmatists, knowledge is judged by how useful it is for solving practical problems. Put simply, truth is whatever works at the time. This stance therefore entails a degree of relativism: what is useful for one person to believe might not be useful for another; therefore truth is relative to the observer. To overcome obvious criticisms, many pragmatists emphasise the importance of consensus – truth is uncovered in the process of rational discourse, and is judged by the participants as whoever has the better arguments. Pragmatism is less dogmatic than the other three stances described above, as pragmatists tend to think a researcher should be free to use whatever research methods shed light on the research problem. In essence, pragmatism adopts an engineering approach to research – it values practical knowledge over abstract knowledge, and uses whatever methods are appropriate to obtain it. Pragmatists use any available methods, and strongly prefer mixed methods research, where several methods are used to shed light on the issue under study (Capps, 2012).

Easterbrook et.al, (2008) argued that although there are examples of research from each of these stances in software engineering literature, the underlying philosophies are never mentioned. There is a belief that this has contributed to confusion around the selection of empirical methods and appropriate evaluation of empirical research in software engineering. In particular, it is impossible to avoid some commitment to a particular stance, as you cannot conduct research, and certainly cannot judge its results, without some criteria for judging what constitutes valid knowledge (Easterbrook et.al, 2008).

The investigation carried out in this thesis used a positivist stance: it is empirically-based research method. Empirical or experimental research can be defined as an investigation based on the observation of actual practice on which to found a theory or answer a question and derive a conclusion in science (Fenton and Bieman, 2013). William (2009) argues that empirical research methods are part of the scientific method that requires all evidences to be empirically based, as opposed to theoretical-based methods that are based on existing theories and explanations. Empirical methods are used extensively in many disciplines, including software engineering. The empirical work of the present thesis focuses on assessing the ease of comprehension of OO programs in comparing to non OO programs in different problems characteristics and different solution decompositions. It seeks to assess ease of comprehension of these problems' solution decomposition, in actual implementation, by programmers who are considered novices to experiences in OO programming. A positivist stance was used in this research; to assess the appropriateness of this selection it is necessary to examine the differences between the deductive and inductive methodological approaches.

3.3 Research Method

This section presents an overview of the choices involved in selecting appropriate empirical research method for software engineering research. The aim was to cover the issues that we faced when deciding how to address the research problem under investigation. The main interest of this research is the investigation of the influence of class structure, problem characteristics, and solution decompositions on the ease of comprehension of OO programs. It is essential to discuss the research strategy used to implement the adopted research method, so as to provide a reliable and valid research result.

The choice of the research method is influenced by the researcher's theoretical perspective and also his attitude towards the ways in which the data will be used (Gray, 2009). It should also explain the rationale behind the selection of the methods adopted. This research has undertaken case study as a research method to reach the overall aim of the research. Two case studies were formed to investigate ease of comprehension of OO programs. The justifications for the selection of case study research method are explained in detail in the following sub-sections.

The choice of a case study as research method had been attributed to a number of reasons. Case study has a distinctive advantage over other research methods when "how" or "why" questions are being posed to discover a current phenomenon and when the researcher has little or no control over the events (Yin, 2009). It offers the opportunity to explain why certain outcomes may happen more than just find out what those outcomes are. This is actually very important for the present research to identify how elements of class structure, problem characteristics, and solution decomposition influence the ease of comprehension of OO programs. Gray (2009) confirmed that a case study approach is particularly useful in revealing the casual relationships between the phenomenon and the context in which it takes place. However, the case study approach has not been widely accepted as a reliable, objective and legitimate research strategy. One of the most critical criticisms directed to this approach related to the difficulty in generalizing the findings to a larger population (Yin, 2009; Thomas, 2003).

It is essential to define a boundary around the phenomenon – what to include and what to exclude (Stark and Torrance, 2005). Yin (2009) proposed four different types for case study designs. These types include: single-case (holistic) designs; single-case (embedded) designs; multiple-case (holistic) designs; multiple-case (embedded) designs. It is important to note that holistic designs are based on single unit of analysis whereas embedded cases include multiple unit of analysis. The undertaking of multiple-case study designs is expensive and time consuming (Yin, 2009). However, this research adopted multiple case (holistic) designs to investigate the ease of comprehension of OO programs in two different problems that have different settings. This can be justified using two main reasons. First, the evidence and conclusions coming out from multiple-case designs are more reliable and convincing than those based on single-case designs and thus the findings are more likely to be generalised (Yin, 2009). Second, the assumptions that there are different types of conditions surrounding comprehension of OO programs and there is a need to have sub-units of cases to cover all different conditions and practices.

According to the purpose of the research, Gray (2009) explained three different forms of study: exploratory, explanatory and descriptive. Robson (2002) indicated that the purpose of the enquiry may change over time. This reflects that the research project may have more than one purpose at the same time. An exploratory study intends to explore what is happening; to seek new insights; to ask questions and to assess the phenomena in a new light. It is valuable particularly when there is very little information known about the phenomenon. On the other hand, explanatory study aims to find out the causal relationships between variables (Saunders et al., 2011). Finally, descriptive study seeks to provide a clear picture about the phenomenon as it already occurs (Hedrick et al., 1993).

The current research is based on multiple-case studies each of which is explanatory in nature. For each case study, well known programming problem for empirical experimentation purposes was chosen. Each empirical experiment intends to investigate factors affect the ease of comprehension of OO programs.

The approach of multiple-case studies is illustrated in Figure 3.1. The figure indicates that the initial step in designing the study must consist of theory development, and then shows that case selection and the definition of specific measures are important steps in the design and data collection process. Each individual case study consists of a "whole" study, in which convergent evidence is sought regarding the facts and conclusions for the case; each case's conclusions are then considered to be the information needing replication by other individual cases. Both the individual cases and the multiple-case results can and should be the focus of a summary report. For each individual case, the report should indicate how and why a particular proposition was demonstrated (or not demonstrated). A cross cases, the report should indicate the extent of the replication logic and why certain cases were predicted to have certain results, whereas other cases, if any, were predicted to have contrasting results.

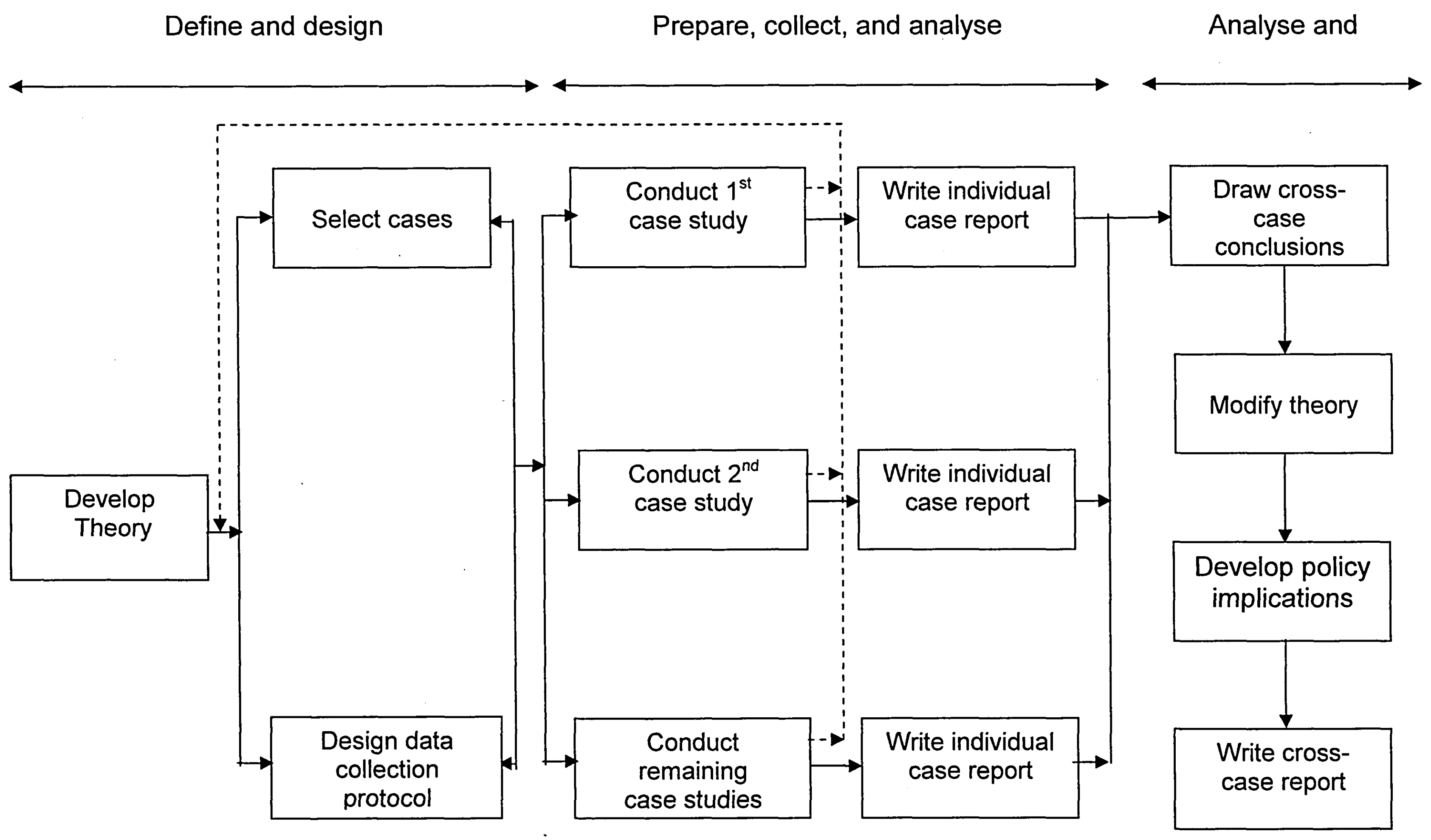
Define and design
Prepare, collect, and analyse
Analyse and
Develop Theory
Select cases
Design data collection protocol
Conduct 1st case study
Conduct 2nd case study
Conduct remaining case studies
Write individual case report
Write individual case report
Write individual case report
Draw cross-case conclusions
Modify theory
Develop policy implications
Write cross-case report

Figure 3.1: Case Study Method (Source, Yin, 2009)

An important part of Figure 3.1 is the dashed-line feedback loop. The loop represents the situation where important discovery occurs during the conduct of one of the individual case studies (e.g., one of the cases did not in fact suit the original design). Such a discovery even may require reconsidering one or more of the study's original theoretical propositions. At this point, "redesign" should take place before proceeding further. Such redesign might involve the selection of alternative cases or changes in the case study.

When using a multiple-case design, a further question you will encounter has to do with the number of cases deemed necessary or sufficient for your study. Multiple-cases, in this sense, resemble multiple experiments, in which a previously developed theory is used as a template with which to compare the empirical results of the case study. If two or more cases are shown to support the same theory, replication may be claimed. The empirical results may be considered yet more potent if two or more cases support the same theory but do not support an equally plausible (Yin, 2009).

3.4 Experimental Framework

Many researchers in the field of software engineering have been motivated by the belief that their recommendations will aid the programmer's task and therefore improve the quality of the software produced. Whilst the contributions made by expert programmers' recommendations have been, in the majority of cases, couched in human factors terms, these recommendations have taken the form that a particular aspect of programming practice will make the programming task either easier, or faster, or less error-prone etc. (Sheil, 1981). Despite the authority and vigour with which these expert recommendations have been made and their common-sense appeal to our intuitive notions of programming, they do not constitute a scientific basis for acceptance but need to be empirically tested. Indeed, experimental evaluation can not only be a useful and powerful tool for assessing such proposals but can also provide empirical evidences augmenting the contributions of practitioners and experts in the field. Therefore, the temptation to accept experts' proposals without evaluation must be resisted.

A researcher can choose from a number of research methods suitable for a particular study and in the area of software engineering. Creswell (2011) highlights five classes of research method that are most likely to be applied in this field of research: controlled experiments, case studies, survey research, ethnographies, and action research.

Many software engineering researchers consider that Weinberg's classic work "The Psychology of Computer Programming" (Weinberg, 1971) was the catalyst for arousing a much-needed interest in human factor investigation generally. In particular, it was directly responsible for most of the investigations on the psychology of programmers' team organisation (Basili and Reiter, 1981; Baker, 2003). The thrust of initial experiments in programming, and to a lesser extent of current works, was in the vein of establishing whether a particular product or practice was in some sense better than others. For example, a number of the earliest contributions comparing different programming approaches from psychological aspects were studies by, for example, Shneiderman (1975, 1977, 1980), Siddiqi, (1984), Pennington (1987a,b), Ramalingam and Wiedenbeck (1997), Wiedenbeck et al. (1999), Wiedenbeck and Ramalingam (1999), Khazaei and Jackson(2002), and Affandy et al. (2011). The primary force responsible for the increased volume of work within the last decade has arisen from the debate caused by the OO programming movement with its radical ideas on programming practices and language constructs. This debate has provided researchers with the opportunity of empirically evaluating various claims made by proponents of the philosophy. However, there has been no parallel increase between OO programming ideas and empirical experimental work in OO programming (Sajaniemi and Kuittinen, 2007, 2008).

In order to understand experimental research, some key terms used should be described. Robson (2002, p.100) uses the term variable to represent a "*defined property or characteristic of a person, thing, group or situation*". *Treatment or Condition* refers to the key factor that is compared or evaluated: a product, a technique, or a method. A precise description of the independent variable is crucial to every experimental design. The independent variable is the variable that is manipulated by the researcher, and takes the form of an experimental treatment, which is either present or not present. This setting, whether the treatment is present or not, describes a conventional model in which the researcher compares two conditions. However, instead of one treatment, two treatments can be used. In this case, treatments are compared with each other. *Subjects* are the people, the participants (not the researchers) involved in the experiment, while *Objects* refer to the entities under investigations and to which the treatment is applied (a project, a program, a product, etc.), the subjects are assigned to *groups.* One group is the "*Experimental group*" (given the experimental treatment) and another group is the "*Control group*" (given no experimental treatment). In a two-treatment comparison, the two groups are designated *Experimental group* 1 and *Experimental group* 2. The *Dependent* variables in experimental research represent the factors that are expected to change in response to the application of the *treatment*. *Extraneous* variables are factors that can affect the dependent variable but are of no interest. These variables should be controlled by the experimental design to eliminate their effect on the outcome. However, in the opposite case, if an extraneous variable influences the dependent variable, due to a weak research design, the variable is considered to be a *confounding* variable.

The two possible empirical evaluation paradigms available to researchers are observational and comparative experiments. Both types involve testing a relationship known as the "*null hypothesis*". This hypothesis asserts that there is no relationship between the independent variable, which is the variable under investigation and therefore the one the experimenter manipulates, and the dependent variable, which is the variable that is affected and therefore the one on which measurements are performed (Siddiqi, 1984). A crucial aspect of designing an experiment is to ensure that the effect on the dependent variable is attributable to the independent variables that may affect the outcome. It is precisely because these controls are absent that there are a number of reservations about results obtained from them.

The simplest form of comparative experiment is introspection and is probably the basis of many recommendations from many prior related empirical studies (for example Pennington 1987a, b; Wiedenbeck et al., 1999). A variant of this rather subjective method is verbal protocol analysis (Simon and Newwell, 1971). Traditionally, this technique involves recording individual subjects "*talking aloud*" about the task they are performing. The recorded speech transcription is divided into lines known as protocols. This technique has been relatively little used in programming experiments, notable exceptions being Brooks, (1977), Atwood and Jeffries (1980), and Burkhardt et.al (2006a, b). However, as

Shneiderman (1980) points out, whilst this technique can be "*worthwhile when the subject is a capable sensitive programmer, since important insights may be obtained*", there is no guarantee about similar behaviour of other programmers.

3.5 Specific of the Methodology

In this section an overview of the approach taken in the experiments conducted in this investigation is presented. A controlled experiment research approach was used in this investigation employing a quantitative technique. The ease of comprehension of OO program was assessed. This was done by asking programmers to carry out a comprehension task.

The experimental part of the investigation consists of two studies, each of which took place in different higher educational institutes, shown later in Chapter 5. Each study employed two groups of programmers to receive different treatments. By choosing university students with relatively convergent programming experience level in each study, an effort was made to eliminate any effect of ability variation and previous programming experience along with knowledge a programmer might have acquired from previous practice in programming. This issue will be discussed later in this section. Aspects related to chosen experimental materials will be also highlighted in this section. To assess subjects' comprehension a combination of metrics were used as measures in both studies and will also be discussed in this section. Other methodological issues related to conducting an experiment, such as data collection and analysis techniques, and ethical issues are also detailed in this section

Having accepted at the outset that programming is a complex form of human problem-solving behaviour, it may seem tempting to consider what psychological theories of problem-solving behaviour have to offer. Unfortunately, as Green (1980) points out, psychology does not have a general theory of thinking and is not likely to have one in any reasonable time to come. Sheil (1981) observes that although some psychological theory is very suggestive, it usually lacks the robustness and precision required to yield exact predictions for behaviour as complex as programming. The need to establish a suitable experimental methodology was recognised earlier by Weissman (1974) and Shneiderman (1975). Since then, there has been little progress and there are few references on investigations into the methodology itself (Easterbrook et al., 2008).

Given that empirical software engineering research is not an exact discipline, several factors, such as design, preparation, and analysis of empirical experiment, should be considered carefully. These factors are not new concepts and, indeed, are very similar to those used within the behavioural sciences (Miller, 2006). There has been considerable good progress in this area (for example, Brooks, 1980; Pennington, 1987a, b; Wiedenbeck et al., 1999; Burkhardt et al. 2006a, b).

Conducting empirical software engineering research has become an important part of evaluating new software technology. However, much existing software technology has still been adopted on the basis of expert opinion and anecdotal evidence, not on the basis of empirical or strong theoretical evidence. Because of this absence, researchers investigating intuitively based on claims of expert programmers have, in many cases, made methodological decision that are, ironically, based on intuitive grounds, (See for example, Basili, 1992; Porter et al, 1995; Sjøberg et al., 2002, 2003, 2005; Fenton and Bieman 2013). While this can be partially blamed on the fact that software engineering is a relatively new field which has grown quickly over a short period of time, empirical evaluation of such technology should be attempted. Evaluation is usually not performed because the need for scientific confirmation is outweighed by the software engineering community's reliance on intuition. However, many software engineering experiments show that many instances of such intuition about software are mistaken (Briand et al. 1999). For example, Daly (1996) reported that in Basili and Selby's (1987) experiments evaluating the efficiency of code reading, functional testing and structural testing, they claimed to have discovered that professional programmers using code reading detected more software faults and had a higher fault detection rate than other methods. This was a surprising result to many of the programmers that participated in the experiments, who felt they had performed better with the testing techniques. While findings about intuition being misleading strengthen the need for more empirical research, it should be noted that performing empirical software engineering research is not an exact discipline and there is a need for researchers to consider various concerns (Briand et al. 1999).

The review of experimental work that follows is not intended to be a comprehensive survey of the literature (for such a treatment, see Sjøberg et al., 2002, 2003, 2005; Easterbrook et al., 2008), but concerns itself specifically with the methodological issues central to programming experiments and the controls necessary for such experiments to be effective.

Daly (1996) argued that the aim of comparative behavioural experiments in programming is to create an environment in which subject behaviour can be observed and analysed effectively. Devising such environments obviously necessitates the selection of suitable subjects, suitable materials that will yield the desired effect of the application of appropriate measures to analyse the effect produced. Therefore, the methodological issues at the heart of this type of experiment relate to a judicious choice of subjects, materials, measures, and data collection (see Brooks, 1980; Siddiqi, 1984; Daly 1996). Additionally, these issues will guide our process in producing an effective experimentation framework.

3.5.1 Subjects

Cook and Campbell (1979) suggest that a weakness in all experiments is that the assumption of initial equivalence between the groups is possibly violated. Even a small difference in the two groups will make the groups not comparable and any observed differences in outcomes could be due to *extraneous* factors or pre-existing differences (Cohen et al., 2013).

There are two primary concerns in the selection of subject, according to Brooks (1980). First, the sample chosen should be representative, that is, the observed behaviour of the sample should be characteristic of the population under consideration. Second, the individuals in it should be relatively homogeneous as regards characteristics other than those under investigation, so as not to influence the results obtained. The reason for insisting that these requirements be satisfied is that, when an experimental sample is sub-divided into groups for differing treatments, it is essential that any significant results obtained for any group are attributable to the treatments and not the characteristics of the subjects in the group. It is not always possible to know all the subject characteristics that will influence experimental results for any programming-related task in advance, although, in practice, for a given task, it may be possible to determine which subject characteristics will introduce an experimental bias. For instance, in an experiment investigating the effect of particular programming practices, differences in such factors as intelligence, discipline studied, and level of education, could introduce an unwanted bias and therefore measures would need to be taken to control their effects.

One aspect that should be considered in this investigation is to minimise the effects of those subject characteristics that are responsible for experimental bias. Anderson (2001) has introduced various well-established techniques which reduce the effect of between-subject variations, these are included:

- random assignment of treatments;
- a "within-subject-design" where all the subjects undergo all experimental treatments;
- the use of "matched pairs", in which subjects of an experiment are matched on some important characteristics; the consequence of this is that no group has a disproportionate number of biased subjects.

There are good reasons for conducting experiments with students as experimental subjects, for example, for testing experimental design and initial hypotheses, or for educational purposes. Depending on the actual experiment, students may also be representative of junior/inexperienced professionals. However, the low proportion of professionals used in software engineering experiments reduces experimental realism, which in turn may inhibit the understanding of industrial software processes and, consequently, technology transfer from the research community to industry (Briand et al. 1999). Hence, to break the trend of few professionals as subjects, Sjøberg et al. (2002, 2003, and 2005) introduced new strategies to overcome these challenges.

A good experimental design should take into consideration any *extraneous* variables that may influence the independent variable by controlling for them. Controlling these *extraneous* variables, usually through randomisation, eliminates systematic bias due to them and increases the internal validity of the experiment (William, 2009).

The implication of experimental investigations with students suggests little justification for assuming that their findings are applicable to experienced programmers. However, comparative experiments involving both types of subjects (novice and expert) need to be performed before such an implication is verified. Some researchers have attempted to design experiments so that the effect of variation in subjects' characteristics is brought under experimental control. They have tried to conduct experimental investigations in such a way as to reveal the class of subjects to which their findings apply. An obvious attempt to control the effect of variability in subject characteristics is to use subjects that are undergoing similar training. However, Cohen et al. (2013) argue that even small differences in experience between subjects will make the subjects not comparable and any observed differences in outcomes could be due to *extraneous* factors or pre-existing differences.

One possible way of ensuring that results obtained are representative of the parent population under consideration is to replicate experiments (Daly 1996). This approach has been successfully adopted by researchers (see for example Pennington 1987a; Wiedenbeck et al 1999; Burkhardt et al., 2006a, b). This can be achieved by performing experiments by novice and expert programmers in such a way that findings can be compared for both groups of subjects.

The technique that is most effective in systematically controlling individual differences in performance between experimental treatments is the within-subjects-design, which has been used in a number of related studies (Ramalingam and Wiedenbeck 1997; Wiedenbeck et al. 1999; Wiedenbeck and Ramalingam 1999; Khazaei and Jackson 2002; Burkhardt et al., 2006a, b). The overall aim of these studies was to investigate the effect of different programming approaches on program comprehension. The experimental procedure consisted of randomly assigning each subject to one of the experiment groups, each of which received exactly the same set of programs to study and recall. Hence, each subject received the same set of experimental treatments. The advantage of this design was that it enabled the investigation to measure the effect of another independent variable that could influence the results, namely, the sequence of programs. However, its major disadvantage generally is that it involves the preparation of large amounts of material. Also it could lead to subjects getting bored because of the number of experimental tasks. More fundamentally, it could lead to what can be referred to as "*carryover effect*". In general, this means that participation in one condition may affect performance in other conditions, thus creating a *confounding* variable.

Robson (2002) suggests the selection of "*equivalent groups through matching*" method by using one or more matching variables. He further draws attention to another approach, a matched-pairs design, used in many experimental designs (Cook and Campbell, 1979), where individuals are matched to form a pair and then each member of the pair is randomly assigned to one group. Creswell

(2011) indicates that a matched-pairs design is expensive, takes time and could possibly result in incomparable groups if some of the participants choose to leave the experiment. Instead of matched-pairs, a *stratified sampling technique* can be used to strengthen the assumption that the two groups are initially equivalent. Another problem with matching is the accuracy with which matching between two individuals will take place. To address this issue, *stratified random sampling* approach has been used in this investigation.

In summary, it must be acknowledged that many researchers were, and still are, forced to use students as subjects. In many cases, because of cost constraints, the use of professionals is impossible. However, the burden of proof still lies on the experimenter to show that the results obtained are representative of the population under consideration.

3.5.2 Materials

The second of the methodological concerns - the choice of experimental materials - is only one factor relating to a broader category, namely, that of "experimental environment" (i.e., that which encompasses all the available stimuli). As Moher and Schneider (1982) point out, behavioural researchers have long realised that differences in results can often be attributed to a variety of factors in the experimental environment. Amongst the environmental factors that investigators need to consider, in their opinion, are:

- the choice of experimental materials;
- the physical setting in which programmers work, so that this can be reflected in the experimental settings;

- different types of incentive (whether money, or the satisfaction of knowing the aims and subsequent achievements of the research, or being reassured that experimental results will not reflect course grades), so that these incentives can be used in a manner that ensure consistent performance of subjects;
- various ways of presenting experimental instructions (i.e., whether in oral or written form, or whether presented informally or formally), as small differences in statement of objectives can be responsible for large differences in results.

The main concern in controlling unwanted bias in the experimental stimuli lies with the choice of material used. There are two issues relating to this choice. Firstly, the material should allow the experimenter to elicit any existing differences in treatments; secondly, the effect of these differences should be attributable to these treatments. When considering the effects of subject variation, it was seen that these could be controlled by the use of a number of standard techniques. However, when choosing experimental material, the controls required for counteracting possible bias will vary from experiment to experiment.

Empirical investigations into programming approach features provide examples of the types of materials-choice problem encountered by researchers and their attempts to overcome the latter. These investigations have used material that includes different sets of languages, natural language (Miller, 1974), small sub-sets of a programming language (Sime et al, 1999), complete languages (Wiedenbeck et al., 1999; Burkhardt et al., 2006a, b) and a special purpose query language (Reisner, 1977). Each type of material allows researchers to focus on the specific issue being investigated, thereby avoiding any bias due to differences in subject training. Obviously, the use of specific material should be based on the specific programming aspect under investigation. Siddiqi (1984) has given an example of Gannon and Horning (1975) work, which evaluated TOPPS and TOPPS II (a pair of statically and dynamically typed languages developed at the University of Maryland for teaching programming and studying the design of programming languages). Siddiqi argued that this work provides a starting point for the systematic comparison of two different (but syntactically similar) programming languages. He pointed out that when it is necessary to investigate the interaction of language features, the latter must be evaluated in the context in which they are used. The author advanced a clear rational in the choice of experimental material for detecting existing differences and made a reasonable case for their findings.

In studies attempting to investigate the comprehension of different programming approaches (Pennington 1987a; Ramalingam and Wiedenbeck 1997; Wiedenbeck, et al 1999; Wiedenbeck and Ramalingam 1999; Khazaei and Jackson 2002), the researchers focused on assigning the subjects' comprehension of a specific program implemented in different programming approaches. The experimental materials used consisted of multiple, diverse, programs examples, each of which was implemented in different programming approaches. Comprehension was evaluated in the context of the mental representations, which represent different types of knowledge, constructed by subjects during comprehension of given treatment. This was considered an appropriate method to investigate how different programming approaches have different effects on program comprehension. The comprehension questions were designed to ask about particular relations between program parts and thus represent different types of knowledge incorporated in the program text. However, no explicit details were given about the characteristics of the problems used and the criteria followed in choosing them. Also it was not clearly stated what type of solution decompositions these studies used in implementing these problems. Moreover, it was unclear if this set of solutions decompositions/programs could unwittingly bias the results of one type of example over the other (Alardawi et. al., 2011a, b).

In Siddiqi's (1984) study, type of task, general problem characteristics, and solution decomposition were the most important elements considered in designing the experimental materials. For example, his study was restricted to problems whose general characteristics were similar to each other. This choice was made to avoid undue emphasis either on input data content or on processing requirement. Limiting the study to a specific problem arena gave the advantage of allowing a more detailed conclusion. In this thesis, characteristics of the problems chosen and their relevant solution decompositions as experimental materials were also limited to a specific problem arena.

In some other cases, devising experimental material may require experimental subjects to undergo special training to adapt to the physical setting of the experiment. The presence of this effect pertaining to the experimental materials is a further factor that could produce an unwanted bias by increasing the already large variance between subjects that is present in programming tasks (see for example Green et al., 1975). Another different example of these effects is the difference in experimental program length highlighted by Ramalingam and Wiedenbeck 1997; Wiedenbeck et al., 1999; Wiedenbeck and Ramalingam 1999; Khazaei and Jackson 2002. This difference is due to the different programming approaches used in implementing experimental materials, for example, the overhead of defining classes and constructors of classes in the materials implemented using OO programming approach. However, no significant effect was reported in this context.

3.5.3 Measures

The final methodological concern is the choice of measures. Human factor investigations in programming have a variety of experimental metrics. These metrics seem to have resulted from a combination of necessity and a carte-blanche application of the principle "to measure is to know". Most experimental researchers would claim that their choice is based on necessity. However, some concerns have been expressed as to the relevance of some of the metrics in contributing to the understanding of the program comprehension process (for example, Brooks, 1980; Pennington 1987a, b; Wiedenbeck et al., 1999). Brooks (1980) argued that most innovations in programming approaches can influence the ease with which programs can be constructed and/or the ease with which existing programs can be comprehended. The experimental tasks used in these relevant studies will, therefore, be aimed at measuring changes in either or both of these properties.

Choosing an experimental measure is mainly based on the nature of the claim being assessed. Software science is, to paraphrase Yeh (1979), a unified and coherent field in which attributes of a computer program, such as implementation time, clarity, structure; error rates, language levels, etc. can be derived from metrics based on intrinsic characteristics of the program itself. Such metrics measure what Shneiderman (1980) terms as the "logical complexity" of the program. These include functions of frequencies of operators and operands in a program. Such metrics have the obvious advantage of facilitating automatic computation of measures from the program text, and the gathering of quantitative evidence that readily lends itself to hypothesis-testing methods. Investigation by Curtis et al. (1979) using Halstead and McCabe's metrics reveal that "these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance" (i.e., they measure the difficulty in comprehending programs which have been written in an "unstructured" manner). Such experiments exhibiting high correlations between factors and their proposed metrics, therefore, can offer useful quantitative evidence. However, because these measures are based on intrinsic properties of the program, they take no account of the interaction between the program and the programmer.

Program comprehension measures are frequently used in empirical studies as a means of establishing the level of comprehension a subject has of a program. Such measures are useful in a wide range of studies involving technological developments, for example new languages/concepts, methods of documentation, approaches to visualisation etc. However; the wide variety of approaches to measuring comprehension means that it is difficult to compare measures and have confidence in the reliability and accuracy of measures. Although there is a variety of metrics, the effect being measured in most cases has been the ease with which existing programs can be comprehended. There is a need for an accurate and reliable measure of program comprehension. Experimentation involving program comprehension tasks usually takes the form of comparing two groups of subjects: a control group and a group undergoing the treatment being investigated. Moreover, the choice of a model of measure will have a decided impact on the construction of experimental materials and on the interpretation of the obtained results. For example, mental simulation measure model is consistent with an interpretation of comprehension of a program as complete knowledge of all the details of the program's construction. Dunsmore and Roper (2000) claim that there is not one established measure of comprehension, and the studies which try to make use of and measure levels of comprehension use a diversity of techniques (often this is because the technique used is related to the particular form of comprehension that the researcher is trying to investigate). For example, there is no direct way to measure a person's comprehension of a piece of code, so all comprehension measures should be more properly referred to as "indirect" or "proxy" measures. The authors identified four essential comprehension measures: *"maintenance"*, *"mental simulation"*, *"static"*, and *"subjective (self-ranking)"*. They empirically found that a measure based on mental simulation is the most reliable comprehension measure. It is also the one that is the most easily controlled (very roughly, by increasing the amount of information that has to be kept in mind, the comprehension task similarly increases). The followed reliable measure is the maintenance based tasks, however; care must be taken with the type of maintenance required. Static tasks appear to be notoriously unreliable and hardest to control. A subjective measure is cheap and worth using along with another measure.

Mental simulation, also known as recall test technique, has been used in many empirical studies for measuring comprehension of program (see for example, Shneiderman, 1977; Pennington 1987a; Ramalingam and Wiedenbeck 1997; Wiedenbeck et al., 1999; Wiedenbeck and Ramalingam 1999; Khazaei and Jackson 2002; Burkhardt et al., 2006a, b; Affandy et al., 2011). These studies used similar methodology, where the technique usually involves presenting a subject with a segment of code and allowing them to study it for an allotted time, estimated by the experimenters. Subjects were then asked to recall as much of the code as possible. In some cases, both of these steps were repeated several times (for example, Wiedenbeck et al., 1999; Affandy et al., 2011). This task is also known as "memory-based" comprehension task. Evidence from these studies supports the use of this technique as a metric for measuring program comprehension. However, Brooks (1980) argued that the problem with the mental simulation approach is that it is applicable only to isolated modules or to toy or student programs. Even though a programmer is thoroughly familiar with, say, a typical compiler, he or she will certainly be unable to reproduce it literally. For systems of realistic size, instructions which encourage subjects to behave consistent with a reconstruction model will probably be more appropriate. The drawback of such instructions is that it will be necessary to develop a scoring scheme that compares programs on underlying structure, rather than on literal equivalence. If the goal of the study is primarily to assess comprehension of a program, then an attractive task is simply to have the subject study this program and then to respond to questions about it. These tasks may be used alone or in conjunction with program construction or program maintenance tasks. The kind of questions used can range from completely open-ended (i.e. "How does this program work?") to completely structured multiple-choice. Open-ended and short-answer questions have an advantage in that it is fairly easy to construct a comprehensive set of questions. On the other hand, scoring is often difficult; unless an elaborate formal scoring scheme has been created, it is often difficult to tell how much more accurate one description of how a program works is from another. The primary drawback to multiple-choice questions is that construction of a sufficiently large set of questions often requires considerable effort, since the questions must satisfy a number of criteria. First, while the correct response should be the one that is chosen most often, the other responses should also be chosen a moderate fraction of the time. Second, if the goal of the experiment is to measure comprehension of the entire program, then the questions must cover all aspects of the program to approximately the same degree. Finally, the content of one question must not inadvertently reveal the answer to another. Given the difficulty of balancing all these requirements, extensive pretesting is virtually a necessity in the use of multiple-choice questions.

In a program comprehension context, subjective rating is simply the subject's own judgement of how much they found a piece of code easy/difficult to comprehend. Although the subjective measure has been criticised in some quarters, it has continually been used for measuring comprehensibility (Shneiderman, 1977, Daly 1996). The primary drawback of a subjective rating measure is that subjective levels could be attributed to the naivety of novice subjects, as they might feel less inhibited to offer a realistic opinion of their abilities. Dunsmore and Roper (2000) suggested using subjective measures can only be useful along with another measure.

Another measure technique most commonly used for measuring the effort required to accomplish a certain programming task is the time taken. This technique has been widely used especially to measure programming tasks such as construction, maintenance, modification, etc. (see for example Brooks, 1980; Siddiqi, 1984, Daly, 1996). Nevertheless, this measure suffers from the problems that are common to time measures in general; for example, difficulties in excluding irrelevant behaviour and irrelevance of some parts of the program to the hypothesis under test (see Brooks, 1980 for more detailed discussion).

An additional problem of ensuring accurate measurement arises if subjects are allowed to set their own work periods. For these reasons, time measures must be supplemented with other measures. For example, Brooks (1980) suggested using the number of debugging runs or the ratio of total lines written to final program size; Daly (1996) used a debriefing questionnaire to produce another alternative interpretation to his results. In studies measuring program comprehension, the rate at which subjects are able to perform the task of reading a program and responding to the related comprehension questions reflects the time required to comprehend this program. Time required to perform a comprehension task can be used as a supplementary measure, as recommended, in this investigation. It gives an indicator about the comprehension of the given program (i.e., an easy-to-comprehend program will take less time than an equivalent difficult-to-comprehend program).

The previous discussion has not advocated any particular experimental measure as being uniformly superior. The choice of a particular measure will primarily depend on the claim being assessed, experimental question, subject population, and resources available to the experimenter. To ensure the accuracy of the assessment method, the current investigation aimed to use a combination of the mentioned metrics (mental simulation, subjective rating, and time) in measuring program comprehension.

Finally, whilst it is desirable to conduct an "ideal experiment" (i.e., one in which unwanted bias due to between-subject variation, non-uniform characteristics in experimental materials, and/or inaccuracies in metrics, is negligible) so that the results obtained can be attributed solely to the treatment under investigation rather than anything else, in practice this is extremely difficult to achieve when investigating the complex human tasks involved in programming. The options are to choose either what Green (1980) describes as the utopian solution, that is, "once psychologists have taken the wrinkles out of a theory of thinking, programming can be treated as a special case and it will be obvious how to make it easier", or to conduct experiments as methodologically precise as is practically achievable so as to "chip away" at the problem under investigation. The latter option was followed in this thesis

3.6 Hypothesis Testing

A statistical test procedure is a decision mechanism, founded on the principles of mathematical probability theory, which transforms the experimental hypothesis and the set of collected observations by means of a decision statistic into an outcome that accepts or rejects that hypothesis. Siddiqi (1984) has reported that Leach (1979) notes the similarity between the mechanics of a statistical procedure and the reasoning used in a court of law provides a useful analogy to explain the force of argument used in the former. Daly (1996) reported that a common method of conducting an experiment is to use statistical significance testing of the Neyman-Pearson type: the form of rejecting or accepting a null hypothesis (denoted H_0), where the null hypothesis is stated simply for the purpose that it may be rejected. The researcher then accepts the alternative hypothesis (denoted *Hi)* and concludes that an effect exists. The standard procedure for carrying out a statistical test is as follows:

1. Posit the validity of the null Hypothesis (i.e., assume that there is no relationship between the variables being investigated).
2. Choose the decision statistic to be used.
3. State the level of significance.

4. Compute, using the decision statistic chosen, the probability of obtaining the observed sample, this probability being denoted by p
5. Reject the null hypothesis (and accept the experimental hypothesis) provided the computed probability exceeds the significance level.

Once the researcher has stated the null and alternative hypotheses, a significance criterion (p*)* should be set. The level of significance is the smallest probability value for the collected observations that would result in the null hypothesis being accepted (Sawyer and Ball, 1981). In theory, the value chosen is at the discretion of the experimenter and may vary from experiment to experiment depending on the degree of assurance required. However, in practice, the sole purpose of experiment is to verify the desired hypothesis and demonstrate the occurrence of an effect. Therefore, the smaller the significance level, the greater the confidence that an effect has occurred (Bailey, 2008). The most frequently used value for the significance level in experimental psychology, so that the researcher can conclude that the observed effect is not the result of chance variation, is 0.05. However, many studies adopt the convention of using the value of the computed probability p, asserting that the result is significant at that level; for example in Sheil (1981) the significant level chosen is $p<0.2$. There is, however, an even greater danger in choosing "appropriate" significance levels in such a manner, because the computed value for p is an estimate that an effect has occurred and not an estimate of the size of an effect. The commonly quoted values within software engineering are 0.05 or 0.1. The empirical study is then conducted, the collected data analysed, and statistical tests applied. If the researcher achieves a statistical result that is less than the preset p value, the null hypothesis is rejected and the alternative hypothesis accepted. From the many articles read by the researcher, it is clear that researchers within software engineering use this type of significance testing as their primary means to detect the presence of an effect within the phenomena being empirically investigated.

3.7 Statistical Methods of Analysis

Several decision statistics were employed to analyse the quantitative data gathered in the studies. Descriptive statistics were also used to describe and summarise data, and more complex techniques, parametric and non-parametric (based on the results of normality tests), were used to make inferences and test the hypotheses advanced. Pallant (2010) stated that the choice of an appropriate statistical technique requires consideration of several factors, including the following: the type of question being addressed, the type of items and scales chosen, the nature of the data, and, finally, the assumptions required for each particular technique.

Daly (1996) argued that the assumptions underlying parametric statistics are rarely used in software engineering studies (e.g., the assumption of normality). Parametric tests require assumptions about the format of the data, and usually normality is assumed for the data. Parametric tests rely on estimating and testing values of parameters. In contrast, a non-parametric test, also called a distribution-free test, does not require any assumption about the distribution of the data (Clark-Carter, 2009). However, each particular test, even a non-parametric one, requires certain criteria to be met. The main advantage of non-parametric tests is that they can be used for small samples where there is no information about the distribution of the sample available (Cohen et al., 2013). However, these tests are less powerful than parametric tests and also less sensitive in that they may not detect differences which actually exist (Walliman, 2005).

The particular techniques employed in this investigation are: Mann-Whitney U unrelated samples test, Hodges-Lehman estimate test, and a kruskal-wallis "chi-square". Consideration and details about these tests will be given in Chapter 5, where the results of specific tests are presented.

3.8 Ethical Issues

Saunders et al. (2007) stated that, in any experimental research, ethical considerations are a significant issue. However, the ethical issues raised by empirical methods have received little attention in software engineering literature (Shull et al., 2008). This is even more important in the current thesis since all studies took place within an academic environment and involved human subjects. Carver et al, (2003) provided a practical guide to ethical research involving humans.

Every ethical and legal issue involved in this research, such as obtaining subjects' consent and academic approval, and conforming to educational principles, was appropriately considered beforehand, and copies of correspondence and approved forms from the Universities, where subjects took place in any of the research's studies, have been attached as Appendix (C). The ethical issues that are relevant to this investigation are outlined below:

- prior to the start of each study, the appropriate academic authorities were informed. In some cases, in order to proceed with the study and have access to the resources of the academic institution, the approval of corresponding paperwork was required;

- in all cases, it was ensured that the participation of the students had no effect on their academic evaluation and grading in any course/module; furthermore, participation did not earn them any course credit;
- participation was voluntarily and students had the right to withdraw from the study if they felt uncomfortable;
- all students were informed orally of the experimental purposes, the procedures, and the task that were to be conducted. Moreover, students were informed of how the data were to be collected and used;
- finally, confidentiality and anonymity was offered to the students. This is especially recommended by the course tutors of the students in order to encourage them to be more sincere and open.

3.9 Chapter Summary

The discussion presented in this chapter has outlined important aspects of the methodology and empirical considerations taken into account in conducting empirical software engineering experiments. It provided an account of the rationale for the choice of a comparative empirical evaluation paradigm and the way in which the method was employed. The methodological concerns and justifications for adopting a controlled comparative experiment in the present investigation were also discussed and the research framework identified. Methodological issues for the specific investigation, such as subjects, materials, and measures were discussed. Moreover, hypothesis testing, statistical methods of analysis, and ethical concerns of this study were also outlined.

Whilst this investigation acknowledges that the evidence obtained using the scientific method is not irrefutable, it does, however, take as axiomatic the view that using this method can provide a probability measure of the comparison being representative of the phenomena under investigation, so that the latter's significance can be assessed. Moreover, a model or theory based on the results from such comparisons then constitutes a proposed explanation of the nature of the phenomenon under investigation. The research was faced with the problem of applying the broad principles of scientific method, rather than a suitably designed experimental methodology. However, the unwanted bias introduced because of this problem can be controlled by judiciously augmenting the scientific method with guidelines based on methodological decisions made in previous empirical investigations. Therefore, it was decided to make effective use of such guidelines so that an increased level of confidence could be placed in the results obtained.

In conclusion, the specific research objectives were to investigate:

- the difference in ease in comprehension between OO programs and non OO programs;
- how elements of class concept, problem characteristics, and solution decomposition influence the comprehension of different types of knowledge in OO programs.

Chapter 4 Specific of the Investigation

4.1 Introduction

The aim of this chapter was to detail the specifics of the investigation. Before these were detailed, consideration was gave to two important aspects: first, the identification of the context within which studies were performed, and hence within which the investigation findings were interpreted; second, the description of the experimental methodology employed - in particular, subjects, materials, and measures - and the steps taken to provide a methodology tailored to the need of the investigation.

4.2 Experimental Context

The experimental context specifics of the investigation described is given here. Several factors contributed to the experiment context. The most significant include: the population under investigation, the physical setting, and the size of the problems to be investigated. Ideally, it was felt desirable to conduct the investigation so that the findings:

- applied to a large cross-section of the programming community whose members' characteristics varied considerably with regard to ability, experience, training, etc.;
- were obtained from an experimental environment which closely resembled the physical setting within which programmers work;
- related to "realistic" programming problems.

In practice, empirical research (whether conducted in an industrial or an academic environment) on a complex programming activity such as program comprehension (an area in which there is a scarcity of empirical investigation) can have little hope of arriving at a satisfactorily complete solution. However, there is a difference between investigations in industrial and academic environments. The former often involve large-scale experiments, whereas the latter are frequently constrained to small-scale experimentation. Therefore, academic studies are open to the often-voiced criticism that such studies deal with "toy" rather than "life-size" programs, and use subjects from academic rather than production environments performing tasks in artificial settings. The reason for this disparity between academic and industrial investigations is often attributable to differences in availability of finance, resources, and subjects.

The circumstances surrounding this investigation were such that no provisional arrangements had been agreed either for industrial co-operation (i.e., there were no commercial organisations who had agreed to supply volunteer subjects and/or make available resources) or for financing programmers to act as volunteers. In addition, at the academic establishments where students are usually willing to be participants, there was no precedent for their being used as experimental subjects, which ruled out any serious possibility of organising experiments in students' free time. Moreover, subjects' tutors were concerned that experimentation should be performed only during one tutorial/practical session (i.e., a period of approximately forty minutes) per term.

These circumstances dictated that:

1. Unpaid subjects should be used.

2. Since subjects' availability was restricted to infrequent, short periods, the size-related complexity of experimental materials should be relatively small.
3. Due to the necessity for adequate numbers of experimental participants, experimentation had to be performed across different institutions.

Nevertheless, it was considered that, despite these practical constraints, an experimental context in which computer-science undergraduate and postgraduate students were asked to comprehend relatively "small" programs under experimental conditions, rather than test-like conditions, could constitute a meaningful research framework. This view could simply be justified on the principle that, because of the scarcity of research in program comprehension, any contribution – even one with severe constraints - could be worthwhile. The provision of a reassurance that subjects were participating in an experiment rather than a test would help to motivate subjects. However, a stronger case can be advanced:

- the chosen subjects represent a significant proportion of the programming community. Subjects were from different institutions across different countries, as well as being potential future professional programmers;
- the specific objectives of the investigation meant that a number of important elements influencing OO program comprehension could be investigated;
- the "reward" of being allowed access to the outcome of the investigation would help to overcome possible adverse effects associated with the artificial setting of experimental conditions.

The overall direction that any programming research project using the scientific method follows is an investigative path combining exploration and evaluation. In an approach where the former is emphasised, the intention is to "discover" from a human-factors standpoint what features of a program make its specification, construction, verification etc. more comprehensible. However, in an approach where emphasis is on evaluation, the investigator posits, prior to experimentation, certain elements which are believed, or assumed, to be of interest; the aim then becomes to "measure" the effect of those elements. Investigations on programming approach and program design by Siddiqi (1984) and Khazaei (1990) provide examples of the exploration approach, whilst the evaluation approach is exemplified by, for example, Good, 1999; Wiedenbeck et al., (1999) and Burkhardt, (2006a, b). The current research essentially followed an evaluation path. The investigative approach followed in this research assumes - based on claims existing in the literature - that elements such as class concept, problem characteristics, and solution decomposition affect ease of comprehension of OO programs. One of the consequences of this decision was that the approach could make it easier to identify evaluative experimental hypotheses (i.e., factors under investigation could be transformed into specific experimental aims and hypotheses). Another reason is that there is an existing number of relevant prior experimental studies which could be used as a starting point and their outcomes built upon (for example, Pennington, 1987a, b; Wiedenbeck et al., 1999; Khazaei and Jackson 2002; Burkhardt et al., 2006a, b).

The investigation can be viewed as two sets of studies. Each was associated with a particular problem characteristic and was designed to study the influence of the mentioned elements. Initially, a comparative experiment was preferred, where the overall aim was to discover the influence of only one factor on program comprehension. For example, the influence of problem characteristics as a factor was discovered by varying the problems used in each study, the influence of solution decomposition was discovered by varying the solution decomposition in each problem. The discussion that now follows details the methodological issues involved in choosing subject, materials, and measures.

4.3 Choice of Subjects

Two previously mentioned factors concerning training and payment restricted the population from which subjects could be chosen to that of computer science students trained in considerably broad principles of programming. Subjects were willing to be unpaid volunteers. The criterion of choosing subjects from this population was dependent on the fact that the studies were comparative in nature. The comparative studies had specific aims of establishing differences for a particular aspect of program comprehension between two or more groups of subjects; this meant that the criterion was the need for homogeneity of subjects' characteristics.

The techniques considered in order to control the *carryover effect* produced by using within-subject-design were between-subjects-design, matched pairs and random assignment of treatments. Use of the first technique meant devising problems of variance in training and intelligence levels between subjects The obvious difficulties in undergoing all experimental treatments and the consequent carryover effect produced (equal to the number of treatment levels) ruled out this technique. However, producing two equivalent experimental subject groups in terms of intelligence level could be problematic.

The second technique would have involved the pairing of subjects in relation to characteristics that might contribute to subject variance. *A stratified random sampling* approach was used to allocate the subjects into two experimental groups. Stratification is the formation of categories or strata from a population.

Every member of the population is assigned to only one stratum that is relatively homogenous with regards to the characteristic or attributes forming the stratum (Black, 1999). Considering the case of this investigation, the two subject groups were equivalent or balanced regarding the characteristics that affect them. One approach, in order to have similar and comparable groups that would ensure high precision in a study, is to match the two groups on a significant variable(s) to the results in the study. The main concern involved in grouping subjects was that the two groups should be similar. Theoretically, this could be achieved by matching on length or course of study undertaken by subjects and course grades obtained. However, this was only partly possible because, in practice, it was not known prior to experimentation which of the subjects would volunteer. Therefore, the homogeneity assumption was based on choosing subjects from the same course (i.e., matching differences due to length and type of training), as well as using *a stratified sampling approach* to allocate subjects into two matched groups. This allocation was based on the course grades that subjects obtained (i.e., assuming that effects of other factors such as skills levels would be randomly distributed across experimental groups). By using *a stratified sampling approach*, in each experiment, subjects were placed into four groups of similar ability – ranked as adequate, medium, good or excellent – and then the members of each of these groups assigned to one of the two groups in a random fashion. Finally, from the two groups, each group was then exposed to different experimental treatment. Subjects also did not know the details of the treatment or experimental procedure to be followed until the start of the assigned task.

4.4 Choice of Problems Used and Solutions Delivered

In terms of experimental materials, it was essential to consider carefully a number of factors which can influence the choice of problems and the related solution decompositions as experimental materials.

One important factor which influences the choice of experimental materials is deciding the type of task to be performed. Two possible choices are program construction and program comprehension tasks. The latter type was considered more appropriate to the needs of the investigation. Program comprehension activity provides an obvious means of investigating several programming tasks (i.e., program development, modification, maintenance, and reuse). It is considered as a necessary prerequisite activity that plays a key role in several programming tasks (von Mayrhauser and Vans, 1995). The activity is usually employed in a comparative studies approach. The obvious preference for comparative experimentation necessitates devising experimental programs in which the variable under investigation is a treatment rather than an attribute. The material to be used for each experiment consisted of functionally equivalent software programs written in two different programming approaches and a set of corresponding comprehension questions reflecting different types of knowledge. The task to be performed broadly involved comprehending the given program by responding to the comprehension questions.

Another most significant factor in choosing experimental materials was the decision to restrict the scope of the investigation to problems whose general characteristics are as following:

- the problems chosen should be built on existing sets of problems used in prior related studies. Thus the investigation findings could be compared to a wide range of existing related studies and able to build on existing knowledge;
- due to subjects' availability, as the experimental subjects were undergraduate and postgraduate computer sciences students who are represented as novice and experienced programmers respectively, the investigation was restricted to problems whose general characteristics do not require any specialised domain knowledge. Thus, chosen problems should also be within the spectrum of programming example programs, and well-suited to different programming paradigms;
- each problem used should have different characteristics from the other. More precisely, the problems used should differ in terms of tangibility of the problem's entities, complexity level, and richness in possessing different solution decompositions. The idea behind using problems with these different characteristics was an attempt to produce empirical evidence concerning the influence of problem characteristics and solution decompositions in the ease of comprehension of OO programs. A decision was made in this study so that one problem could be considered as "*trivial*" (its problem entities are relatively tangible and exist in the real world, are less complex, and can posses one possible solution decomposition). However, the other problem could be considered as "*rich*" (its problem entities are relatively intangible and do not exist physically in the real world, it is comparatively more complex than the *trivial* one, and can possess different solution decompositions);

- additionally, and to be more consistent, the problems chosen should fall into well-known standard problem frames and be drawn from different problem domains. Jackson's problems classification was identified as a good classification standard of problem types to be followed (Jackson 1995; 2005). This classification was also considered in distinguishing between the experimental problems used in this thesis.

The problems chosen were considered to satisfy the above-mentioned requirements. The reason for choosing this approach was that it would have the advantage of reaching more detailed conclusions that - although derived from relatively limited problems scope - could with circumspection be extrapolated to a family of programming examples.

For the first study, the problem used was derived, and then adopted, from Ramalingam and Wiedenbeck (1997) “Car” problem as a basis for the first study's material (specified in Appendix A). This problem had been used in other similar studies (Wiedenbeck et al., 1999; Wiedenbeck and Ramalingam, 1999; Khazaei and Jackson, 2002). The Car problem was considered to be a good starting point for the investigation. It meets the investigation requirements as a *trivial* problem. It also possesses one type of solution decomposition. Moreover, according to Jackson's classification, this problem is considered as an example of an information display problem, where the information machine is required to monitor the state and behaviour of a concrete "real world" problem entity, in this case a car, and to display information about it, in this case a speed, on a display.

It was considered important to the investigation to choose another problem with characteristics *rich* enough to be perceived from different perceptions, which can thus possess different solution decompositions. This was a pre-requisite to investigate the influence of problem characteristics and solution decompositions on OO program comprehension.

Naur's "Line-Edit" problem (Naur, 1969), which is considered as a *rich* problem, was used as a basis for the second study's material (specified in Appendix B). This problem was also used in Siddiqi's (1984) study. The problem differs from the Car problem as it is relatively complex and its entities are comparably intangible. It is also richer than the Car problem in that it can possess different types of solution decompositions. Considering Jackson's classification, a Line-Edit problem is a good example of the workpiece problem. A user edits a text document using problem entities such as character, line, and word. These problem entities are considered as less tangible than those of the Car problem. The requirement is that the edit commands issued by the user should effect appropriate corresponding changes in the workpiece. Figure 4.1show the problems used and how they differ in their characteristics.

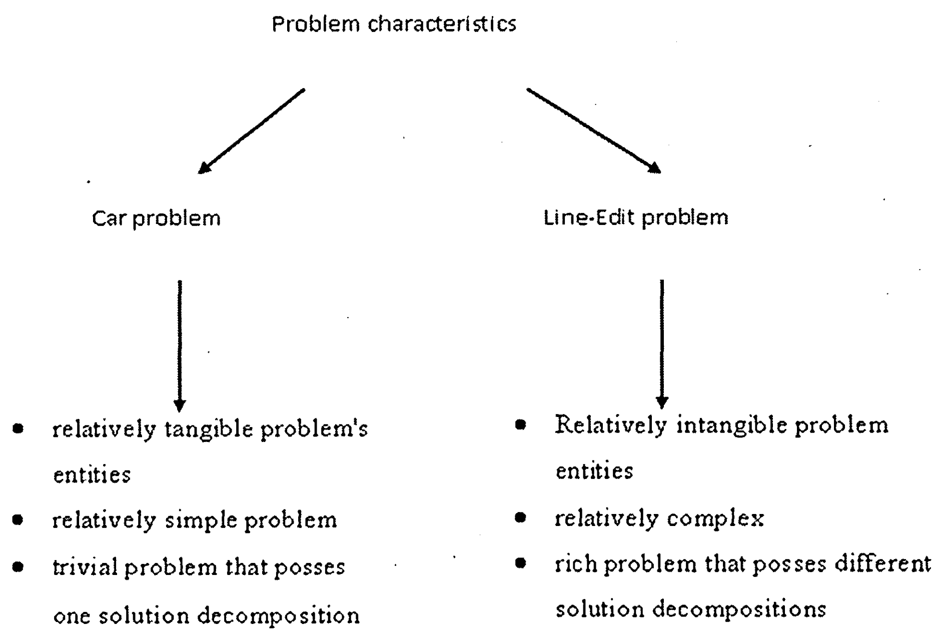
Problem characteristics
Car problem
Line-Edit problem
relatively tangible problem's entities
relatively simple problem
trivial problem that posses one solution decomposition
Relatively intangible problem entities
relatively complex
rich problem that posses different solution decompositions

Figure 4.1: Classification of the problems used in the research

Since one aspect of this section concerns choice and design of the experimental materials, it is considered relevant and vital to detail how the investigation proceeded in designing these experimental materials, and how the solution decompositions for each experimental problem (Car and Line-Edit) were achieved. Different experimental materials were developed for the purposes of the investigation. These represent the correct solution decompositions of the corresponding problems, each of which uses a different programming approach. The aim was to produce, for the Car problem, two functionally equivalent program versions in which each program version is based on the same solution decomposition but implemented in different programming approaches. One program version containing classes, hereafter called "*Object* based" program, while the other does not, hereafter called "*Non-Object* based" program. For the Line-Edit problem, there are two functionally equivalent program versions in which each program version is based on different types of solution decompositions but implemented in a different programming approach. One program version, hereafter known as "*Object* based" program, contains classes, while the other, hereafter known as "*Non-Object* based" program, does not. In order to achieve this aim, for the Car problem, *primitive* solution decomposition was used in designing *Object* based and *Non-Object* based program versions containing and not containing classes respectively. For the Line-Edit problem, different solution decompositions were applied. A *primitive* solution decomposition was used in designing the *Non-Object* based program version that does not contain classes. An *Abstract* decomposition solution was used in designing the *Object* based program version containing classes.

In decomposing Car and Line-Edit problems, an informal "top-down" exposition is presented. The function/process is used as a basis for the design of the problems solutions. Initially, each solution is characterised in terms of an "item-type-to-be-processed", hereafter referred to as "*item*". Siddiqi (1984) defined the *item* as:

> *"a perception obtained from consideration of input data and/or processing requirements (including output) which becomes pivotal to the subsequent decomposition of the problem." (p 79)*

The various alternative *item* based solutions to each problem, if they exist, are mapped onto characteristic process structure pairs for each problem.

In designing the Car solutions, only one possible *item*, which is "car status" process, can be considered as a perception that dominates the elaboration of the *primitive* solution decomposition. This solution decomposition is then implemented in two different forms (*Non-Object* based and *Object* based). Thus, we considered the Car problem as a *trivial* problem.

In designing the line-edit solutions, Siddiqi (1984) reported that there are three possible alternatives of *item,* namely, "character", "line", or "word". Each of these perceptions can dominate the elaboration of equivalent solutions. Siddiqi (1984) reported that a solution based on more *abstract* perception (viewing the *item* as a word) is superior to those based on more *primitive* perceptions (viewing the *item* as a character). Taking these findings into consideration, the decision was made to consider the character *item* as perception that dominates the elaboration of the *primitive* solution decomposition. This *primitive* solution decomposition was implemented in the form of a *Non-Object* based program. The decision was also made to consider the word *item* as an appropriate perception that dominates the elaboration of the *abstract* solution decomposition. This *abstract* solution decomposition was implemented in the form of an *Object* based program. Therefore, the Line-Edit problem was considered to be a *rich* problem.

As result, four solution decompositions implemented in different forms (specified in Appendices A and B) are defined as following:

1. CD1: the *Non-Object* based *primitive* solution decomposition of the Car problem.
2. CD2: the *Object* based *primitive* solution decomposition of the Car problem.
3. LD1: the *Non-Object* based *primitive* solution decomposition of the Line-Edit problem.
4. LD2: the *Object* based *abstract* solution decomposition of the Line-Edit problem.

The rest of this section details the way in which the study designed a set of program versions for each experimental problem.

4.4.1 Designing Car Problem Solutions

Considering the Car problem adopted from Ramalingam and Wiedenbeck (1997), and after much consideration, the decision was made to extend the program by adding more details. However, to be consistent with the prior related studies, to avoid any bias in the findings produced, and to make the findings more comparable, these additional details do not impact on the functionality of the evolved programs. The added details represent the addition of two more entities, an "*engine*" and a "*body*", to the problem in the new *Object* based program version. This in turn resulted in adding two more classes to the original OO program (see Ramalingam and Wiedenbeck 1997 for the original OO program version). The justification for this was to utilise the use of classes. However, in the *Non-Object* program version, these problem entities were represented as data variables. Moreover, both new program versions offered additional output messages back to the user.

For the CD1, where "car status" perception dominates the elaboration of the decomposition, the solution involves a sequence of two processes (“input car details” and “car status”). The "input car details" process involves inputting details of the added specifications of the problem entities *engine* and *body*. The elaboration of the "car status" process consists of the two composite processes "process number of passengers" and "process check speed limit". The complete refinement of the CD1 is presented in figure 4.2.

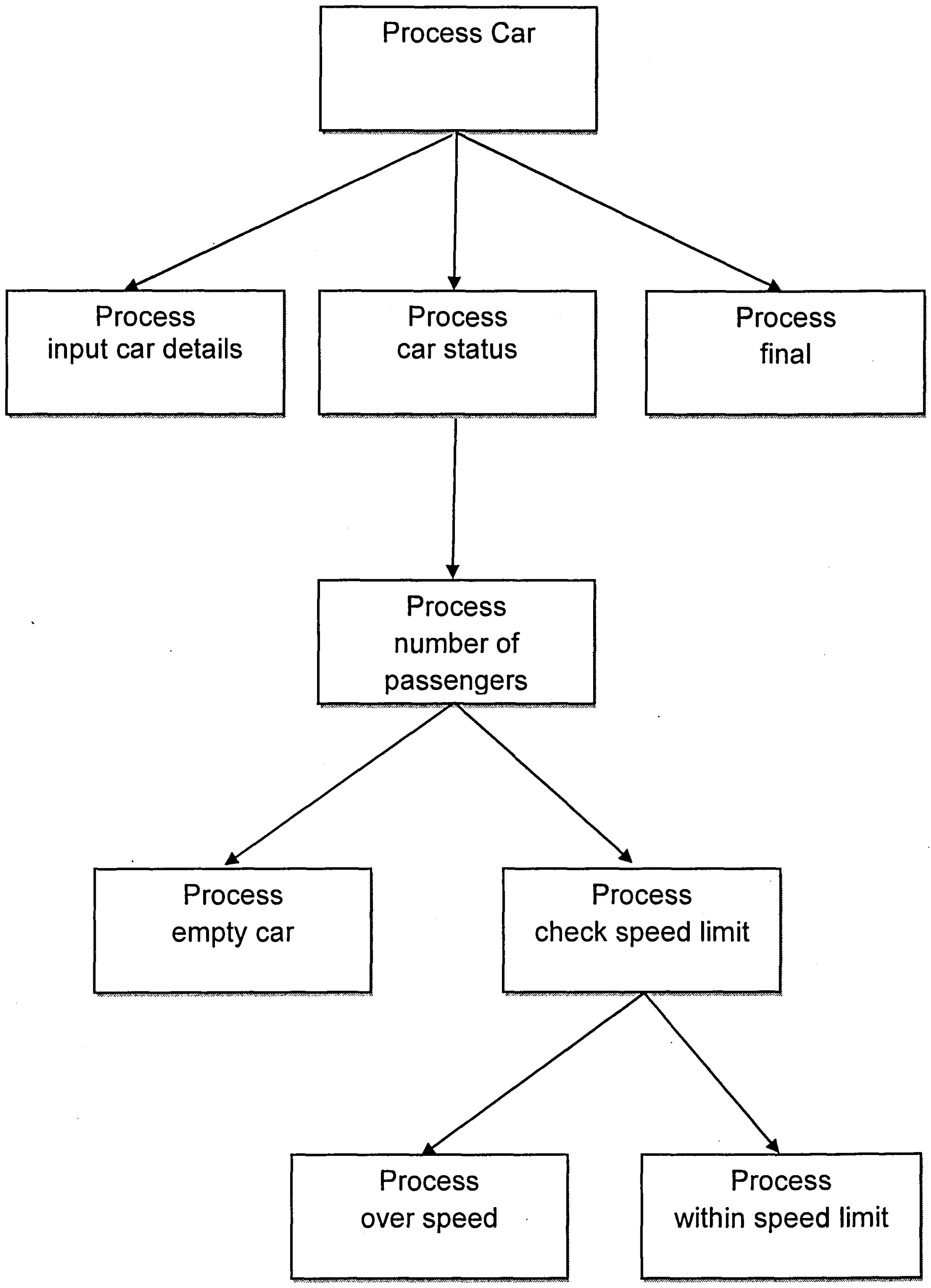
Process Car
Process input car details
Process car status
Process final
Process number of passengers
Process empty car
Process check speed limit
Process over speed
Process within speed limit

Figure 4.2 The complete refinement of the CD1

For the CD2, where also "car status" perception dominates the elaboration of the decomposition, a different criterion was applied. This criterion was fundamental to meet the design specifications of Object based version of the Car problem. The solution was characterised in terms of a set of objects required to perform certain processes. The "input car details" process involved defining the specifications of the problem entities *body* and *engine* as separate objects. The object *engine* has the attribute "power" and performs operations "set-engine" and "get-engine". The object *body*" has the attribute "brand" and performs operations "set-body" and "get-body". Both *engine* and *body* are composed into the object "*car*", as they are parts of this object. The "car status" process was incorporated as an operation in the object *car*. The object *car* has the attributes "speed" and "passengers" and performs operations called "set-car", "car-status", and "get-car". Figure 4.3 illustrates the complete refinement of the CD2.

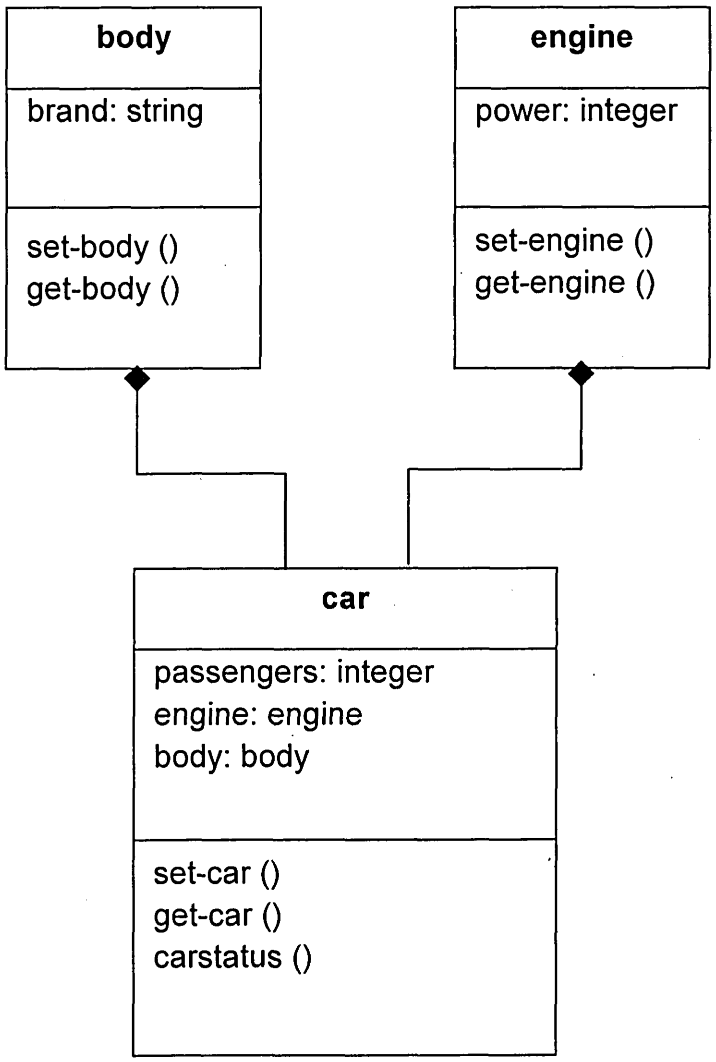

Figure 4.3 the complete refinement of CD2

Although CD1 and CD2 were differently implemented, the way in which the "car-status" perception is processed is similar in both CD1 and CD2. The "car status" was incorporated as a process in CD1 and was incorporated as an operation embodied within the object *car* in CD2. More precisely, the perception of input data and processing requirements was similar even though the implementation differed. This is because the characteristic of the car problem was not rich enough to be perceived from different perceptions so alternative *items* could be introduced.

4.4.2 Designing the Line-Edit Solution

For LD1, where “character” perception dominates the elaboration of the decomposition, the solution was borrowed from Siddiqi's L2-type solution. The solution involves two processes, "non-space character" and "space character". The elaboration of the "non-space character" process consists of two elementary actions: adding a character to a word, and incrementing the size of the word. The refinement of the "space character" process needs to distinguish between the cases when a space is either redundant or acts as control character. The complete refinement of LD1 is presented below in figure 4.4.

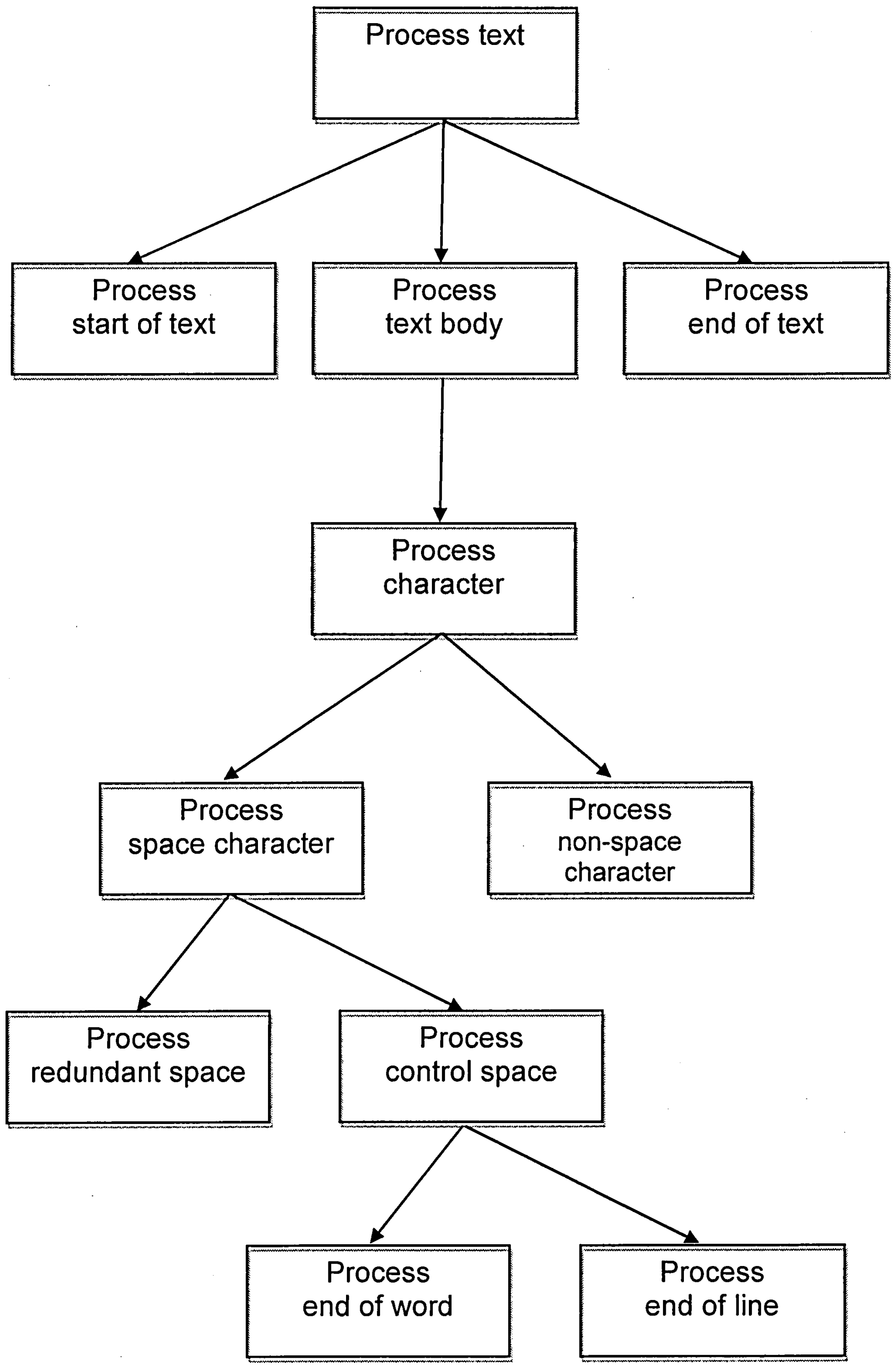

Figure 4.4 The complete refinement of LD1

LD2 was an adoption of Siddiqi's L3-type solution. This fundamental adoption was made to meet the design specifications of an Object based version of the Line-Edit problem. Since word perception dominates the elaboration of the decomposition, word was considered as problem entity. This entity was implemented as object called "word" with its relevant possible attributes and operations. There was a need for another object to manipulate the given text, by use of the "word" object, in the word basis. This object was called "buildingword". Figure 4.5 illustrates the complete refinement of LD2

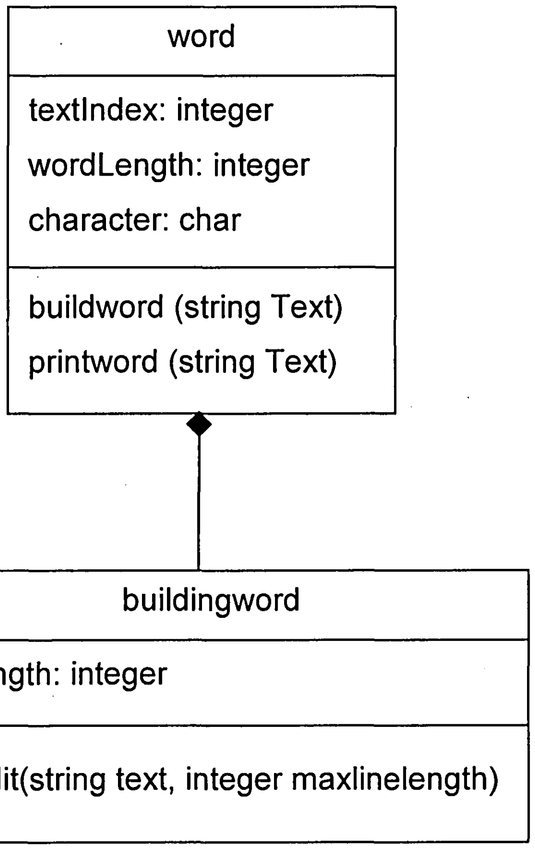

Figure 4.5: The complete refinement of LD2

LD1 and LD2 were differently designed and equivalent alternative solutions were produced. The perception dominating the elaboration of the decomposition was different in LD1 and LD2. The characteristic of the line-edit problem was *rich* enough to be perceived from different perceptions, thus alternative solutions were achieved.

4.4.3 Designing the Experimental Materials

This investigation carried out two empirical studies (hereafter known as Car study and Line-Edit study). For each study, suitable experimental materials were specifically devised based on the above design settings, this comprising outline programs with corresponding lists of comprehension questions (hereafter known as "experimental treatments"). The experimental treatments for the Car study and the Line-Edit study are provided in Appendices A and B respectively. It is considered worthwhile to mention that the term "experimental treatment" throughout this thesis represents: the program being comprehended, the corresponding list of comprehension questions, the ranking question, and the background questionnaire.

For the Car study, two functionally equivalent treatments were developed. The first treatment was developed based on CD1 (see figure 4.2), with the absence of classes, (known as *Non-Object* based program). The second treatment was developed based on CD2 (see figure 4.3), with the presence of classes, (known as *Object* based program).

The *Object* based program contains three classes, *engine*, *body*, and *car*, each class consisting of private data member(s) and public interface containing declarations of member functions (see figure 4.3). The execution starts in the program's main function, which begins by creating instances of classes, *engine* and *body*. These instances objects are composed into the class *car*. The main program function creates an instance of class *car* and calls the other objects' methods in which the principal computations were carried out (see Appendix A for the *Object* based program of the Car study).

The alternative, functionally equivalent, *Non-Object* based program does not use classes; rather it was created by removing all classes (see figure 4.2). Entities of *engine* and *body* were represented as data variables. The *Non-Object* based program initialises variables, and then it carries out the principal computations of the program in the program's main function (see Appendix A for the *Non-Object* based program of the Car study).

Similarly, for the Line-Edit study two functionally equivalent treatments were developed. The first treatment was developed based on LD1 (see figure 4.4), with the absence of classes (known as *Non-Object* based program). The second treatment was developed based on LD2 (see figure 4.5), with the presence of classes, (known as *Object* based program).

The *Object* based program contains two classes, *word* and *buildingword*, each class consisting of private data member(s) and public interface containing declarations of member functions (see figure 4.5). The execution starts in the program's main function, which begins by creating instance of class *word*. This instance object is composed in *buildingword* class. The main program function creates an instance of class *buildingword* which then calls the *word* object's functions in which the principal computations were carried out (see Appendix B for the *Object* based program of the Line-Edit study).

The alternative, functionally equivalent, *Non-Object* based program does not use classes, rather it was created by removing all classes (see figure 4.4). The *Non-Object* based program initialises variables, and then it carries out the principal computations of the program in the program's main function (see Appendix B for the *Non-Object* based program of the Line-Edit study).

All the experimental programs (*Non-Object* and *Object* based programs of both Car and Line-Edit studies) were developed to a level such that complete run programs were obtained. The general format of the experimental materials (including program and list of comprehension questions) was designed to be consistent with previous related studies (see for example, Siddiqi, 1984; Ramalingam and Wiedenbeck, 1997; Wiedenbeck et al., 1999; Khazaei and Jackson, 2002; Burkhardt et al., 2006a, b). The experimental procedure involved subjects having to read a given program and then respond to a list of respective comprehension questions. Furthermore, in order to ensure that the effect of any significant differences could be attributed to the experimental treatment rather than alternative sources of variation, the following criteria were considered in developing the experimental treatments:

- the programming languages used were from the subjects' main programming languages. Since subjects were from different institutions, JAVA and Visual Basic.net (hereafter known as JAVA and VB respectively) were used. Possible effects of the syntactical differences between the programming languages used were minimised as much as possible;
- the positioning and size of the programs were such that no implied significance could be attached to them regarding the readability of the code. Each program was fitted onto only one page;
- the stylistic rules used regarding formatting and discrimination of key-words, choice of variables and procedure names, indentation, comments etc. were

in accordance with the conventions for program clarity as advocated on their programming code.

One methodological problem is the variation in programs' sizes, in term of number of lines of code. Table 4.1 shows the number of lines of code (hereafter known as LOC) for all programs in each experiment.

Table 4.1: The corresponding LOC of each program in each study

the study	programming language	program version	LOC
Car	VB	Non-Object	24
		Object	60
	JAVA	Non-Object	35
		Object	48
Line-Edit	VB	Non-Object	37
		Object	55
	JAVA	Non-Object	29
		Object	45

The *Non-Object* based programs in Car and Line-Edit studies were slightly shorter than the corresponding *Object* based programs. This basically was due to the overhead of class and function definitions in the *Object* based programs.

For each experimental treatment, a set of corresponding comprehension questions was formulated. The comprehension questions were formulated to address the six different knowledge categories. A mixture of all knowledge categories introduced in Pennington's model (Pennington, 1987a) with one knowledge category added from Burkhardt's model (Burkhardt et. al., 2006a, b) composing the model of OO program comprehension was used in this investigation., see table 4.2. Due to the scope of the subjects' ability and the nature of the Non-Object based programs, the rest of Burkhardt's model knowledge categories were excluded.

Table 4.2: Correspondence between knowledge categories, knowledge structures, and mental representation of OO program comprehension model used in this investigation

knowledge category	**knowledge structures**	**mental representation**
elementary operations (EO)	**text structure knowledge**	**dynamic and functional views**
control flow (CF)	**text structure knowledge**	**dynamic view**
data flow (DF)	**plan knowledge**	**dynamic and functional views**
program goals (GOAL)	**plan knowledge**	**functional view**
state (STATE)	**plan knowledge**	**dynamic and functional view**
problem classes (CLASS)	**problem and plan knowledge**	**object view**

Each knowledge category explained as follow:

- **elementary operations** form part of the text microstructure, and constitute basic text units usually consisting of one or few lines of code. The feature of this category is that it is directly available in the program text, thus it represents low-level knowledge;

- **control flow** forms part of the text microstructure, and constitutes the links between text units, which is sequential in the simplest case or, in complex situations, involves looping or calls to subprograms; thus this knowledge is procedural in nature and represents low-level knowledge;

- **data flow** relates to communication between data variables, corresponds to data flow relationships connecting units of local plans within a routine and also changes that occurs to data variables while they pass through the program execution. The transformations of the data are, thus, at the heart of whatever useful action a program achieves. This knowledge is considered to be high-level knowledge;

- **program goals** explain the goal of the whole program, what the program accomplishes in terms of the problem situation it addresses. Program goals knowledge expresses what the program does in terms of entities, relationships, and actions in the world; this knowledge is usually not directly available in a program text, but must be inferred from the program text in combination with knowledge of the real world problem domain of the program. Thus, it represents high-level knowledge;

- **state** comprises the state of all aspects of the program at the time a given action occurs in a program. It is considered as high-level knowledge;

- **problem classes** are objects which directly model entities existing in the problem domain and represent high level knowledge.

To illuminate any discrepancy with other related prior studies, the syntax/wording of the comprehension questions was similar to that used in Wiedenbeck (1997), Good (1999), and Burkhardt et al. (2006a, b). However, the wording of the questions was reviewed by the tutors of the relevant courses from which the subjects participating in the studies were drawn. This was to ensure subjects were familiar with the terms used and to improve their response to the questions. In terms of the type of response, most related studies had used questions with yes/no answers (Ramalingam and Wiedenbeck, 1997; Wiedenbeck et al., 1999; Wiedenbeck and Ramalingam, 1999; Burkhardt, 2006a, b). This type of question indicates that answers at a level of 50% can be considered barely above chance. It is also difficult to predict whether subjects are just guessing their answers or these answers are their real responses. To overcome the problem of subjects guessing the answer, each question was presented with the option of three responses (yes/no/don't know). This idea was used by Khazaei and Jackson (2002). The `don't know` response was considered as a `no` response in analysing the results.

A ranking question was also positioned at the end of the comprehension task and was excluded from the time recorded for the task. Subjects were asked to explain how well they understood the given program. Introducing this question was an attempt to gather more data and therefore possibly to offer alternative interpretations of the results obtained via the comprehension questions.

For the purpose of the studies, a background questionnaire was used at the beginning of each experimental session. The aim of this questionnaire was to collect demographic data to highlight any possible existence of significant differences in prior programming experience among subjects. The questionnaire asked subjects to rank their previous programming experience at one of three levels (novice/intermediate/expert). To ensure consistency with other relevant studies, the questionnaire format, which can be found in Appendices A and B, was similar to the background questionnaire used in Good's (1999) study.

In order to conduct the Car study in Libyan institutions, translation of the experimental materials (including the background questionnaire and comprehension questions) into Arabic was undertaken by the researcher. The Arabic versions were also revised by corresponding tutors who are teaching programming courses where part of the car experiment took place. This revision was made to illuminate any possible terminology differences that Libyan subjects may be unfamiliar with.

4.5 Choice of metrics

Deciding upon suitable metrics depends largely on the variable being investigated and the type of task being performed (Siddiqi, 1984; Daly, 1996). In the experiments where subjects comprehend an existing program, it was necessary to devise an investigation rationale/framework so that subjects' attempted comprehension could be analysed.

In the classification of attempted comprehension, the quantitative metric in analysing subjects' attempts at comprehension would be the proportion of comprehension based on different programs. It was considered that this metric would effectively quantify subjects' "preference" for a particular program and would therefore be a useful contribution to the investigation.

In both the Car and Line-Edit studies, there was a specific aim of investigating the ease of comprehension of OO programs. Subjects were asked to comprehend specific programs that represent problems' different delivered solutions; these *Non-Object* based programs and *Object* based programs (hereafter known as program versions) were developed for this purpose. It was necessary to devise metrics to analyse subjects' attempted comprehension. Comprehension was measured in terms of the total correct responses to the respective comprehension questions about one of the program versions of a given study (*Non-Object* based or *Object* based of either Car or Line-Edit problem). The rationale for using this measure is that the average of the total correct responses reflects the amount of knowledge subjects have gathered from a given program. Moreover, the justification for such a view is as follows: it can be assumed that, in comprehending a program, a subject gathers different types of knowledge about the program under comprehension. This is consistent with Pennington's ideas of cognitive representation and mental representations in the area of program comprehension (Pennington 1987a).

Considering the quality of the metrics used in this investigation, Dunsmore and Roper (2000) evaluated the mental simulation metric as the most reliable and accurate comprehension measure. Furthermore, this measure has been widely used in a number of empirical related studies (for example, Ramalingam and Wiedenbeck 1997; Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al., 1999; Good, 1999; Khazaei and Jackson, 2002; Burkhardt et al., 2006a, b; Affandy et al., 2011). It follows therefore that the ease/difficulty with which subjects comprehend a given program version will be dependent upon the degree to which the latter "mirrors" their comprehension. On this basis, it was found reasonable to use this metric to assess the comprehension of different program versions with respect to the total correct responses (hereafter known as "*performance')* to the respective comprehension questions.

Another metric used for measuring subjects' comprehension was the *"time"* required to comprehend the given program version. The *time* required was measured in terms of the total time taken to accomplish a given task. The rationale of using this measure is that the time which subjects spend on performing the task of reading the program and responding to the related comprehension questions reflects the ease/difficulty of the task. It was considered necessary to use this metric, in conjunction with *performance,* to assess the ease of comprehension of different program versions with respect to the *time* required for their comprehension.

It was also considered important to use a subjective rating measure (hereafter known as "ranking") along with performance measure. A subjective *ranking* question was used as a supplementary metric alongside *performance* and *time* in this investigation. The rationale for using this measure is that it reflects subjects' own judgment about the level of the comprehensibility of a piece of given program. Moreover, it could yield extra results/information that could support and complement results from the other measures.

Performance, *time*, and *ranking* measure the ease of comprehension in each program version, whereas knowledge category performance measures the extent to which these knowledge categories are present and interact as hypothesised by the model. According to all the above, the metrics used to analyse the investigation results are:

- ***time***: the time each subject spends to accomplish the comprehension task.
- ***performance***: the subjects' correct responses to all the corresponding comprehension questions.
- ***ranking***: the level at which subjects rank their comprehension of the given program version.

The first metric was used to evaluate the significance of the rate at which subjects accomplished the given task. The second metric was used to evaluate the significance of the subjects' correct responses to the questions. The third metric was used to evaluate the significance of subjects' judgement of their comprehension experiences.

4.6 Chapter Summary

In this chapter, the experimentation context was identified. Factors such as population, physical settings, and problem size were considered and detailed. In relation to the aim of the investigation, an evaluation investigation approach was followed in this thesis.

The chapter has also discussed concerns relating to specific methodological issues. It reported how the experimental methodology was tailored to the needs of the investigation. The chapter detailed how subjects, materials and metrics were chosen and the circumstances surrounding these choices. The next chapter details the plan, execution, and analysis of the investigation involving the Car and Line-Edit studies.

Chapter 5 Report of the Investigation

Class structure represents one of the essential concepts of OO approach and therefore, a good understanding of this concept will positively affect the effectiveness of OO programmers. To investigate the phenomenon in a controlled manner, decision was made to conduct controlled comparative empirical studies. The problems chosen in these studies were considered appropriate to the needs of the investigation because they contain explicit references to both primitive and abstract problem specification features. The Car problem used in the Car study was considered trivial problem that posses only one primitive possible solution decomposition. However, the Line-Edit problem used in the Line-Edit study was considered rich problem that posses primitive and abstract solution decompositions. This approach was considered appropriate to the need of the investigation.

This chapter reports the investigation of the research involving the Car study and the Line-Edit study. The studies were designed to assess the ease of comprehension of OO programs by varying problem characteristics and their possible solution decompositions in each study. The class concept, problem characteristics, and solution decomposition are elements considered pertinent to the objective of obtaining insight into subjects' comprehension. A group of subjects was asked to perform an identical comprehension task on a simple *Object* based program; another equivalent group of subjects was asked to perform the same task but with a functionally equivalent *Non-Object* based program. A background questionnaire was also collected to highlight any significant differences among subjects' previous programming experiences. The remainder of this chapter details the two studies.

5.1 Case Study 1: The Car Study

5.1.1 Aim

The aim of this study was to assess the ease of comprehension OO program of the Car problem's solution. The problem considered contained *trivial* specifications. This aim was achieved through focusing on comprehension of different sets of knowledge categories of an *Object* based program of the Car in contrast to an equivalent *Non -Object* based program of the Car.

5.1.2 Subjects

Due to subjects' availability, the study was performed over three different academic institutions, and two different OO programming languages were used. The study was conducted in three different experimental sets (hereafter known as expset1, expset2, and expset3 respectively). Table 5.1 illustrates the institutions and the programming language used for each experimental set.

Table 5.1 : Institutions and programming language taught in each experimental set

experimental set	**institution**	**programming language**
expset1	Sheffield Hallam University-UK	VB
expset2	Faculty of Electronics-Libya	VB
expset3	Faculty of Computer Technologies-Libya	JAVA

For expset1, the subjects were first year undergraduate students studying a Programming Concepts module using VB at Sheffield Hallam University in the UK. For expset2, the subjects were first year undergraduate students also studying Programming Concepts using VB at the Faculty of Electronics in Libya. Both modules introduce programming concepts particularly in Event-Driven style. For expset3, the subjects were first year undergraduate students studying an Introduction to Programming module using Java at the Faculty of Computer Technologies in Libya. The aim of recruiting these three sets was to gather data from a large number of subjects.

As a result, a total of 353 undergraduate first year computer science students, from three different institutions, participated in the study, all of whom had completed 15 to 16 weeks' study of programming using either VB or JAVA.

Demographic data collected from the background questionnaire showed that the subjects' gender ratio was 37% males to 62% females. The average age was about 20 years. The majority of the participants had no previous experience in OO programming and the only significant programming languages currently experienced were VB or JAVA. However, all subjects were studying programming concepts when the study took place. Prior to each experimental session, a stratified random sampling approach was used to matching the experimental subjects. The subjects were then randomly allocated into one of two matched groups (hereafter known as *Non-Object* group and *Object* groups). For all experimental sessions, 176 subjects were allocated in the *Non-Object* group and 177 subjects were allocated in the *Object* group. Matching was based on the subjects' grades on the courses they were attending. Table 5.2 shows subjects' group allocations in each experimental set.

Table 5.2 : Group allocations for each experiment set of the Car study

Group	expset1	expset2	expset3	totals
Non-Object	25 subjects	52 subjects	99 subjects	176 subjects
Object	25 subjects	54 subjects	98 subjects	177 subjects

5.1.3 Materials

Each subject was supplied with the following experimental materials:

- a copy of the background questionnaire;
- a copy of either the *Non-Object* based or the *Object* based program version, each of which contains a copy of the program code;
- a list of the corresponding comprehension questions with the option of three responses for each question (YES, NO, DON'T KNOW);
- the comprehension ranking question.

VB or JAVA versions of the *Non-Object* and *Object* based programs were supplied depending on the course the subjects enrolled in. Appendix A provides a copy of all the above experimental materials.

5.1.4 Procedure

The experiment was paper-based; each experiment session was conducted during a lab session for each experimental set and over different time periods. In each experiment session, the experimental instructions were explained verbally by the researcher at the beginning. Subjects were also informed that they were participating in an experiment and they were assured that they were not being assessed. The experimental instructions followed for each experimental session were:

1. All subjects were verbally informed about the purposes and the procedure of the experiment (see Appendix A).

2. Each subject was asked to fill out a background questionnaire at the beginning of the experimental session.

3. Once all subjects had responded to the background questionnaire, each subject was presented with a hard copy of either the *Non-Object* or *Object* based program version with its corresponding list of comprehension questions depending on the group the subject was allocated to. The start time of the experiment was recorded.

4. After completing the comprehension task, the end time was recorded for each subject individually. Then, subjects were asked to respond to the ranking question at the end. As the experiment is timed, the ranking question was excluded from the recorded duration of the experimental session.

Since the experiment was paper-based, data was collected manually by the researcher. The data collected from conducting each experimental session for any given subject were: (i) answers to the background questionnaire, (ii) responses to both the corresponding comprehension questions and the ranking question, and (iii) the start and end time of the task.

A pilot study was performed using four experienced programmers, one for each program version for each programming language. No significant issues were encountered during the pilot study. However, there was a need for clarification of several points in the experimental instructions (i.e., in the first run of the experiment, three programmers mentioned that the experimental instructions were not clear enough and should be explained in more detail). These suggestions were incorporated into the experimental instructions, which made the instructions clearer in the subsequent experimental runs.

5.1.5 Metrics and Experimental Hypotheses

Three metrics were used to analysis the data. These are:

- ***time*** is measured by time taken to accomplish the comprehension task;
- ***performance*** is measured by correct responses to all knowledge categories;
- ***ranking*** is measured by the level at which subjects ranked their comprehension.

Standard significance testing was adopted for the stated null hypotheses. These were:

H_{01}: There is no significant difference in terms of ease of comprehension between *Non-Object* based program and *Object* based program by (i) *time*, (ii) *performance*, and (iii) *ranking.*

H_{02}: There is no significant difference in terms of ease of comprehension in knowledge categories between *Non-Object* based program and *Object* based program.

H_{03}: There is no significant difference in terms of ease of comprehension in knowledge categories by (i) All group (ii) *Non-Object* group, and (iii *Object* group.

To be rejected in favour of the alternatives hypotheses:

H_{11}: There is significant difference in terms of ease of comprehension between *Non-Object* based program and *Object* based program by (i) *time*, (ii) *performance*, and (iii) *ranking*.

H_{12}: There is significant difference in terms of ease of comprehension in knowledge categories between *Non-Object* based program and *Object* based program

H_{13}: There is significant difference in terms of ease of comprehension in knowledge categories by (i) All group (ii) *Non-Object* group, and (iii) *Object* group.

No direction has been specified in the alternative hypotheses: it was not predicted whether the effect on comprehension would be positive or negative. This was attributed to the varying opinions expressed in the program comprehension literature about the ease of comprehension of OO programs.

The experimental design provided two independent variables and three dependent variables. The "program version" and "knowledge category" were the independent variables. The dependent variables were: *time*, *performance,* and *ranking*.

5.1.6 Experimental Results of the Car Study

Since data distribution is shown to be non-normal (see appendix d), to be conservative, corresponding non-parametric statistical tests were applied. For the case of having only two unrelated samples (*Non-Object* and *Object* groups), a Mann-Whitney U ranks (unrelated) test was calculated. However, in the case of having more than two unrelated samples (i.e. the differences in performances in knowledge categories) the Kruskal-Wallis test was calculated. Kruskal-Wallis, equivalent to ANOVA (Analysis of variance) parametric test, is used to calculate the differences between more than two unrelated sets of data (Hinton, 2004 and De Sá, 2007).

Preliminary statistical analysis was done to determine whether there was a significant difference in ease of comprehension among the three different experiment sets. A Kruskal-Wallis test was run. This tests whether the *time* and *performance* for the three different experiments were significantly different. An "experimental sets" (expset1, expset2, and expset3) was the independent variable and "*time*" and "*performance*" were the dependent variables. The result was not significant. Therefore, *time* and *performance* among the three different experimental sets was not significantly different. Thus the "experiment sets" was not included as a variable in further analysis. Since different programming languages were used in the study, different specific details of notations (such as presentations of classes and syntax) were expected to be a factor that might affect the comprehension. Therefore, testing this factor could be instructive at this stage. Another preliminary analysis was done to determine whether the programming languages affected the *performance* and the *time.* A Mann-Whitney was run with "programming language" (VB and java) as the independent variable and "*time*" and "*performance*" as the dependent variables. The result was not significant. There was no significant effect of the programming languages on the *time* and *performance*. Thus "programming language" was not included as a factor in further analysis. These preliminary analyses have established that hereafter the experimental data from the three different experimental sets and two different programming languages could be grouped together and combined as one data set for data analysis, beginning with the whole of the three experiment sets.

The rest of the analysis was divided into five main levels. In the first level, *time* was tested. This level of analysis was to compare the *time* between the program versions. The second level of analysis was done to compare the *performance* between the program versions. The third level of analysis aimed to compare the *ranking* between the program versions. These three levels of analysis aim to these the first null hypothesis. The fourth level of analysis aim to test the second null hypothesis, this level was to compare the *performance* in knowledge category between program versions. The last level of analysis was to compare *performance* in each knowledge category for different sets of subjects groups (All group, *Non-Object* group, and *Object* group), this last level aim to test the third null hypothesis.

5.1.6.1 Comparison of the Time

This level of the analysis was to compare the *time* required to accomplish the given task. The descriptive analysis of the timing data is presented in table 5.3. Column two gives the number of observed times (N), Columns three and four give the minimum and maximum times, column five gives the mean time, and column six gives the standard deviation. First row presents the summary of *Non-Object* group and second row represents the summary of *Object* group. Note that the mean times for the *Non-Object* and *Object* groups are very similar which indicates that there was no program version effect in terms of *time*.

Table 5.3: Statistical summary of the time of the Car study

descriptive statistics					
	N	min	max	mean	SD
time of *Non-Object* group	176	6	18	12.61	2.500
time of *Object* group	177	7	18	12.58	2.501

Statistical tests were then applied to test the first null hypothesis. The difference in *time* between subjects' times to complete the *Non-Object* based program and the *Object* based program was calculated. A Mann-Whitney U test was run with "program version" (i.e. *Non-Object* and *Object* based programs) as the independent variable and "*time*" as the dependent variable. The result was not significant (U=15533.000, p=.964>0.05 2-tailed). The first null hypothesis H_{01} was accepted in terms of *time*. There is no significant difference between the *Object* based and the *Non-Object* based programs in relating to *time*. The sum and means of the ranks for the subjects are shown in table 5.4 and the Mann-Whitney test results are shown in table 5.5.

Table 5.4: Mean ranks of time in the Car study

ranks				
	program version	N	mean rank	sum of ranks
time	*Non-Object* based	176	176.76	31109.00
	Object based	177	177.24	31372.00
	total	353		

Table 5.5: Mann-Whitney test results of time in the Car study

test statistics	
	time
Mann-Whitney U	**15533.000**
asymp. Sig. (2-tailed)	**.964**

Since the result was not significant, no further analysis was considered.

5.1.6.2 Comparison of the Performance

Analysis was carried out to find the effect of the program version on *performance*. The descriptive analysis of the *performance* data is presented in table 5.6. Examination shows that the mean performance of the *Object* based group is higher than the mean performance of the *Non-Object* based group. This indicates that there is program version effect in terms of *performance*.

Table 5.6: Statistical summary of the performance of the Car study

descriptive statistics					
	N	min	max	mean	SD
performance of *Non-Object* Group	176	10.53	94.74	54.54	17.59
performance of *Object* Group	177	10.53	100.00	63.81	18.23

In comparing the two different groups each of which undergoing one of the two different "program versions" (as independent variable) and *performance* (as dependent variable), a Mann-Whitney U test was applied. The result revealed that there was a significant difference (U=10784.500, p=.000<0.05 2-tailed). The *performance* of *Non-Object* group and *Object* group is significantly different supporting the H_{11}. The sum and means of the ranks for the subjects are shown in table 5.7 and the Mann-Whitney results are shown in table 5.8.

Table 5.7: Ranks of performance in the Car study

ranks				
	program version	N	mean rank	sum of ranks
performance	Non-Object based	176	149.78	26360.50
	Object based	177	204.07	36120.50
	total	353		

Table 5.8: Mann-Whitney test result of performance in the Car study

test statistics	
	performance
Mann-Whitney U	10784.500
asymp. Sig. (2-tailed)	.000

We wanted to estimate the difference in *performance* between the two groups, so a Hodges-Lehman estimate test was run as a follow-up. The test is used to measure the size of the difference between the subjects (Hinton, 2004; Chris, 2004).

5.1.6.3 Comparison of the Ranking

This level of analysis was to assess subjects' judgment about their comprehension of the given program version (*Non-Object* based or *Object* based). The assessment was based on the subjects ranking the given program in terms of its comprehensibility. Figure 5.1 shows the mean of subjects' responses to the ranking categories *(Not very well, fairly to moderated well,* and *Well to very well)* broken down by the programs versions.

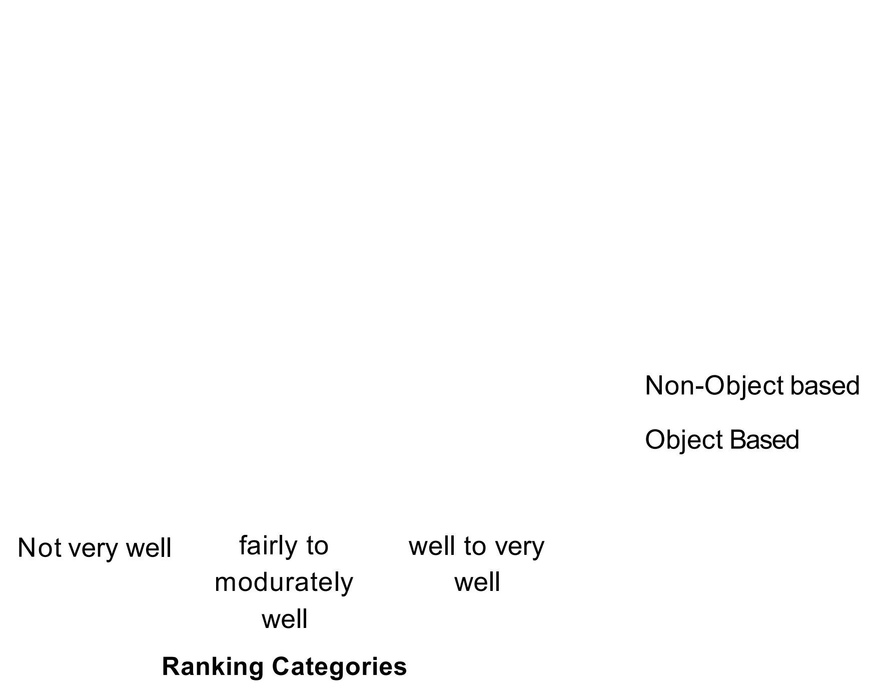

Figure 5.1: Graphical representation of the subjects' ranking for each program version of the Car study

Examination of the data confirms that the *Object* based program was often assessed as being easier to comprehend than the *Non-Object* based program. Moreover, comparing subjects' ranking of the given program version showed that the *Non-Object* group differed from the *Object* group in their responses.

44% of the *Non-Object* group graded their program as *Not very well.* However, only 22% of the *Object* group graded their program as *Not very well.* Moreover, 15% of the *Non-Object* group graded their program as *Well to very well*, while 31% of the *Object* group graded their program in this ranking category. This gives substantial agreement and supports the above findings from the *performance* measure that the *Object* based program appears to be more comprehensible than the *Non-Object* program. Moreover, this also shows that *ranking* is a possible measure in assessing the ease of comprehension of OO programs.

On the basis of the above three measures used (*time*, *performance*, and *ranking*), there is a significant difference between *Non-Object* based and *Object* based programs as measured by *performance* (the number of correct responses) and *ranking* (subjects' judgment about the comprehensibility), but not by *time* (the time taken to accomplish the given task). One could reasonably argue that *performance* and *ranking* are better indicators of comprehension. However, *time* is not as good an indicator in measuring comprehension, and therefore, *time* will not be used further in the investigation.

5.1.6.4 Comparison of Performance in Knowledge Category between Program Versions

This analysis was undertaken to account for effect of program version (*Non-Object* based and *Object* based) on each knowledge category (*elementary operations, control flow, data flow, program goals, state,* and *problem classes*). Figure 5.2 shows the mean of the correct responses to each knowledge category broken down by the programs versions.

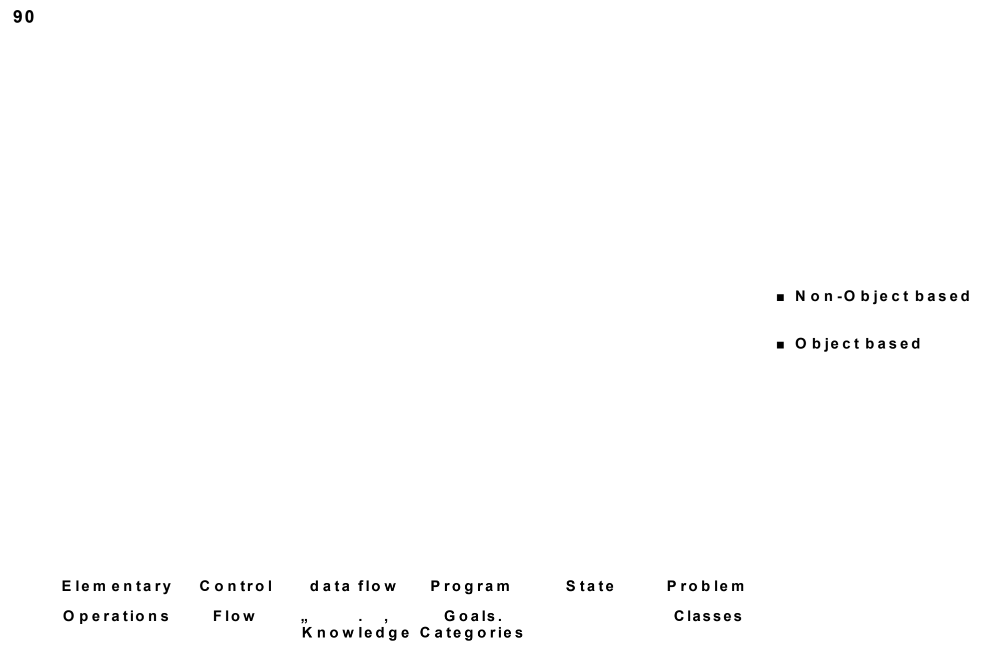

Figure 5.2: Graphical representation of the performance in knowledge categories for each program version of the Car study

In comparing the two experimental groups where each of which undergoing one of the two different Car program versions (as an independent variable) with *performance* in each knowledge category (as a dependent variable), a Mann-Whitney U test was applied to test the second null hypothesis $H0_2$- The result revealed that *Data Flow* and *Problem Classes* knowledge categories were significantly different supporting H i2. The sum and means of the ranks for the groups are shown in table 5.9 and the Mann-Whitney results for each knowledge category are shown in table 5.10.

Table 5.9: Mean ranks of performances in knowledge categories between program version of the Car study

ranks				
knowledge categories	program version	N	mean rank	sum of ranks
Elementary Operations	Non-Object based	176	171.20	30131.50
	Object based	177	182.77	32349.50
	total	353		
Control Flow	Non-Object based	176	176.81	31118.50
	Object based	177	177.19	31362.50
	total	353		
Data Flow	Non-Object based	176	187.52	33004.00
	Object based	177	176.54	31177.00
	total	353		
Program Goals	Non-Object based	176	169.67	29862.00
	Object based	177	184.29	32619.00
	total	353		
State	Non-Object based	176	183.93	32371.50
	Object based	177	170.11	30109.50
	total	353		
Problem Classes	Non-Object based	176	125.09	22015.50
	Object based	177	228.62	40465.50
	total	353		

Table 5.10: Mann-Whitney test results of performances in knowledge categories between program versions of the Car study

test statistics						
	Elementary Operations	Control Flow	Data Flow	Program Goals	State	Problem Classes
Mann-Whitney U	14555.50	15542.50	13724.00	14286.00	14356.50	6439.500
asymp. sig.(2-tailed)	.261	.971	.069	.150	.131	.000

To estimate the difference in *performance* between the two groups for the significant *Problem Classes* knowledge category, a Hodges-Lehman estimate test was run as a follow-up for these two significant categories. The Hodges-Lehman indicated that the *Object* group performed 30% better than the *Non-Object* group in the *Problem Classes* knowledge category.

One could say that subjects using the *Object* based program version outperformed subjects using the *Non-Object* based program by nearly a third in the *Problem Classes* knowledge category. However, there appeared to be no difference between the groups in the other knowledge categories. Therefore, the easiest result of H_{12}, relating to significant difference between subjects' performance in relation to the two program versions, is that it is largely attributable to the *Problem Classes* knowledge category.

5.1.6.5 Comparison of Performance in Knowledge Categories

We measured comprehension by *performance* and *ranking* and showing the ease of comprehension of the *Object* based program version over the *Non-Object* based program version. Taking the knowledge categories, which composed the model used in this investigation, into account, our main interest in comprehension accuracy is in the differences that might occur between different knowledge categories, that is, between questions asked about different types of knowledge in the given programs versions. We assume that higher error rates for questions in a particular knowledge category imply that the knowledge in that category is less easily comprehended. This level of analysis aimed to test whether these knowledge categories exist and, if so, how they interact, and whether they help in explaining any difference found in *performance*. This was done by comparing the six knowledge categories, mentioned above, in different sets of groups (All group, *Non-Object* group, and *Object* group).

5.1.6.5.1 Comparison of Performance in Knowledge Categories for All Group

The descriptive analysis of data for the All group's *performance* is presented in table 5.11. Examination shows that the mean *performance* of the *State* knowledge category is the highest among all other knowledge categories. However, *performance* of the *Data Flow* and the *Program Goals* knowledge categories were the lowest among other knowledge categories.

Table 5.11: Statistical summary of the performance in each knowledge category of All groups of the Car study

descriptive statistics					
knowledge categories	N	min	max	mean	SD
Elementary Operations	353	10.53	100.00	60.33	29.31
Control Flow	353	10.53	100.00	59.86	30.81
Data Flow	353	10.53	100.00	53.96	38.78
Program Goals	353	10.53	100.00	53.35	28.24
State	353	10.53	100.00	76.34	35.35
Problem Classes	353	10.53	100.00	57.22	30.34

Statistical tests were then applied to test the third null hypothesis H_{03} for All group. A Kruskal-Wallis test was run. The independent variable was the "knowledge categories". The dependent variable was the *performance* in each knowledge category of the All group. The test revealed a significant difference among knowledge categories ($\chi^2=128.12$, $p=.000<0.05$). Means of the ranks for All group's *performance* in each knowledge category are shown in table 5.12 and the Kruskal-Wallis results are shown in table 5.13.

Table 5.12: Ranks of All group's performance in each knowledge category of the Car study

ranks			
	knowledge categories	N	mean rank
performance of for All group	Elementary Operations	353	1050.52
	Control Flow	353	1051.64
	Data Flow	353	965.53
	Program Goals	353	925.14
	State	353	1372.35
	Problem Classes	353	991.81
	Total	2118	

Table 5.13: Kruskal-Wallis test result of All group's performance in each knowledge category in the Car study

test statistics	
	performance for All group
chi-square	128.12
df	5
asymp. sig.	.000

The result revealed that, for All group, *performance* in each knowledge category was significantly different. Moreover, the knowledge category *State* had the highest score value, whilst the knowledge categories *Program Goals* and *Data Flow* were amongst the lowest mean scores. Since the results were significant, a pairwise comparison test was run as a follow-up to investigate the interaction between the knowledge categories.

Table 5.14: Pairwise comparison of All groups' performance in each knowledge category of the Car study

Sample1-Sample2	Test Statistic	Std. Error	Std.Test Statistic	Sig.	Adj.Sig.
Elementary Operations-State	-321.834	44.958	-7.158	.000	.000
Control Flow-State	-320.708	44.958	-7.133	.000	.000
Program Goals-State	-447.208	44.958	-9.947	.000	.000
Data Flow-State	-406.820	44.958	-9.049	.000	.000
Problem Classes-State	-380.537	44.958	-8.464	.000	.000
Program Goals-Control Flow	126.500	44.958	2.814	.005	.073
Program Goals-Elementary Operations	125.374	44.958	2.789	.005	.079
Data Flow-Control Flow	86.112	44.958	1.915	.055	.832
Data Flow-Elementary Operations	84.986	44.958	1.890	.059	.881
Program Goals-Data Flow	40.388	44.958	.898	.369	1.000
Problem Classes-Control Flow	59.829	44.958	1.331	.183	1.000
Elementary Operations-Control Flow	-1.126	44.958	-.025	.980	1.000
Data Flow-Problem Classes	-26.283	44.958	-.585	.559	1.000
Program Goals-Problem Classes	-66.671	44.958	-1.483	.138	1.000
Problem Classes-Elementary Operations	58.703	44.958	1.306	.192	1.000

From table 5.14, we can see that the five pairwise combinations all involving the *State* knowledge category have a significant interaction with the other five knowledge categories. This is to be expected given the considerably higher value of the mean *performance* score of *State* knowledge category found in table 5.11. Therefore, for All group, State knowledge plays a significant positive role in program comprehension for groups using both *Non-Object* based and *Object* based versions of programs. The dominant role of this reinforced by its high value in the first test as well as its continued presence in the interaction with other knowledge categories.

5.1.6.5.2 Comparison of Performance in Knowledge Categories for Non-Object Group

The descriptive analysis of data for the *Non-Object* group's *performance* is presented table 5.15. Examination shows that the mean *performance* of the *State* knowledge category is the highest among all other knowledge categories. This was expected as it was found in All group's *performance*. However, examination also shows that the *Problem Classes* knowledge category represents the lowest *performance* of all other knowledge categories.

Table 5.15: Statistical summary of the performance in each knowledge category of *Non-Object* group in the Car study

descriptive statistics					
knowledge categories	N	min	max	mean	SD
Elementary Operations	176	10.53	100.00	58.52	29.65
Control Flow	176	10.53	100.00	59.84	30.30
Data Flow	176	10.53	100.00	58.23	38.21
Program Goals	176	10.53	100.00	51.13	27.80
State	176	10.53	100.00	79.26	33.53
Problem Classes	176	10.53	100.00	42.14	25.93

A Kruskal-Wallis test was run with the independent variable "knowledge categories" and dependent variable *performance* in each knowledge category for the *Non-Object* group. The test revealed a significant difference in *performance* among knowledge categories (χ^2=127.91, p=.000<0.05). Means of the ranks for *Non-Object* groups' *performance* are shown in table 5.16 and the Kruskal-Wallis results are shown in table 5.17.

Table 5.16: Ranks of *Non-Object* group's performance in each knowledge category of the Car study

ranks			
	knowledge categories	N	mean rank
performance for *Non-Object* group	Elementary Operations	176	529.07
	Control Flow	176	546.12
	Data Flow	176	531.59
	Program Goals	176	464.07
	State	176	721.99
	Problem Classes	176	378.17
	Total	1056	

Table 5.17: Kruskal-Wallis test result of *Non-Object* group's performance in each knowledge category of the Car study

test Statistics	
	performance for *Non-Object* group
chi-square	127.91
df	5
asymp. sig.	.000

The result revealed that, for the *Non-Object* based group, there is a significant difference in *performance* between knowledge categories. The *State* knowledge category has a considerably higher mean score, whilst the *Problem Classes* knowledge category has a lower mean score.

A pairwise comparison test was also run as a follow-up to investigate the interaction between the knowledge categories for *Non-Object* group.

Table 5.18: Pairwise comparison of *Non-Object* group's performance in each knowledge category of the Car study

Sample1-Sample2	Test Statistic	Std. Error	Std.Test Statistic	Sig.	Adj.Sig.
Elementary Operations-State	-192.923	31.759	-6.075	.000	.000
Control Flow-State	-175.872	31.759	-5.538	.000	.000
Problem Classes-Control Flow	167.949	31.759	5.288	.000	.000
Problem Classes-Data Flow	153.420	31.759	4.831	.000	.000
Program Goals-State	-257.915	31.759	-8.121	.000	.000
Data Flow-State	-190.401	31.759	-5.995	.000	.000
Problem Classes-Elementary Operations	150.898	31.759	4.751	.000	.000
Problem Classes-State	-343.821	31.759	-10.826	.000	.000
Problem Classes-Program Goals	85.906	31.759	2.705	.007	.102
Program Goals-Control Flow	82.043	31.759	2.583	.010	.147
Program Goals-Data Flow	67.514	31.759	2.126	.034	.503
Program Goals-Elementary Operations	64.991	31.759	2.046	.041	.611
Data Flow-Control Flow	14.528	31.759	.457	.647	1.000
Elementary Operations-Control Flow	-17.051	31.759	.591	.591	1.000
Elementary Operations-Data Flow	-2.523	31.759	.937	.937	1.000

From table 5.18, we can see that the *State* knowledge category has a significant interaction with all other five knowledge categories. However, *Problem Classes* knowledge category has a significant interaction with *Elementary Operations, Control Flow*, *Data Flow*, and *State* knowledge categories. This is to be expected given the considerably higher value of the mean *performance* score of *State* knowledge category and considerably lower value of the mean *performance* score of the *Problem Classes* knowledge category found in table 5.15. We can conclude that, for the *Non-Object* group, there is a significant difference in *performance* between knowledge categories. This difference is largely attributable in a positive manner to *State* knowledge category. In relation to *Problem Classes* knowledge category, it plays a significant negative role in program comprehension for the group using a *Non-Object* version of the program. The different dominant roles of these knowledge categories are emphasised by their high and low values in the first test as well as their continued presence in the interaction with other knowledge categories.

5.1.6.5.3 Comparison of Performance in Knowledge Categories for Object Group

Comparing only the *Object* based group, table 5.19 shows the descriptive analysis of data for the *Object* groups' *performance*. Examination shows that the mean *performance* of both *State* and *Problem Classes* knowledge categories are higher than all other knowledge categories. However, examination also shows that the *Data Flow* knowledge category represents the lowest *performance* of all other knowledge categories.

Table 5.19: Statistical summary of the performance in each knowledge category of *Object* group of the Car study

descriptive statistics					
knowledge categories	N	min	max	mean	SD
Elementary Operations	177	10.53	100.00	62.59	28.66
Control Flow	177	10.53	100.00	59.88	31.45
Data Flow	177	10.53	100.00	49.71	38.98
Program Goals	177	10.53	100.00	55.55	28.57
State	177	10.53	100.00	73.44	36.94
Problem Classes	177	10.53	100.00	72.22	26.80

A Kruskal-Wallis test with "knowledge categories" as independent variable and *performance* in each knowledge category as dependent variable was revealed a significant result (χ^2=71.72, p=.000<0.05). Means of the ranks for the *Object* group's *performance* are shown in table 5.20 and the Kruskal-Wallis results are shown in table 5.21.

Table 5.20: Ranks of Object group's performance in each knowledge category of the Car study

ranks			
	knowledge categories	N	mean rank
performance for *Object* group	Elementary Operations	176	517.41
	Control Flow	176	502.89
	Data Flow	176	433.69
	Program Goals	176	457.39
	State	176	649.34
	Problem Classes	176	610.26
	Total	1056	

Table 5.21: Kruskal-Wallis test result of *Object* group's performance in each knowledge category of the Car study

test statistics	
	performance for *Object* group
chi-square	71.72
df	5
asymp. sig.	.000

The result revealed that, for the *Object* based group, there is a significant difference in *performance* between knowledge categories. The *State* and *Problem Classes* knowledge category have considerably higher mean scores than other knowledge categories. To investigate the interactions between the knowledge categories for *Object* group, a pairwise comparison was run.

Table 5.22: Pairwise comparison of *Object* group's performance in each knowledge category of the Car study

Sample1-Sample2	Test Statistic	Std. Error	Std.Test Statistic	Sig.	Adj.Sig.
Elementary Operations-State	-221.497	31.802	-6.965	.000	.000
Control Flow-State	-143.372	31.802	-4.508	.000	.000
Elementary Operations-Problem Classes	-187.372	31.802	-5.895	.000	.000
Program Goals-State	-187.866	31.802	-5.907	.000	.000
Data Flow-State	-214.852	31.802	-6.756	.000	.000
Data Flow-Problem Classes	-180.827	31.802	-5.686	.000	.000
Program Goals-Problem Classes	-153.841	31.802	-4.837	.000	.000
Control Flow-Problem Classes	-109.347	31.802	-3.438	.001	.009
Elementary Operations-Control Flow	-78.125	31.802	-2.457	.014	.201
Data Flow-Control Flow	71.480	31.802	2.248	.025	.369
Program Goals-Control Flow	44.494	31.802	1.399	.162	1.000
Elementary Operations- Data Flow	-6.645	31.802	-.209	.834	1.000
Data Flow-Program Goals	-26.986	31.802	-.849	.396	1.000
Problem Classes-State	-34.026	31.802	-1.070	.285	1.000
Elementary Operations-Program Goals	-33.631	31.802	-1.057	.290	1.000

Table 5.22 showed that both *State* and *Problem Classes* knowledge categories have a significant interaction with all other four knowledge categories. This is to be expected given the considerably higher value of the mean *performance* score of *State* and *Problem Classes* knowledge categories. We can conclude that, for the group using the *Object* based program version, *performance* scores differed significantly in the knowledge categories. It could be argued that *State* and *Problem Classes* play a significant positive role in program comprehension for the subjects using *Object* based version of program.

In summarising findings from the Car study, we could argue that, in measuring the ease of comprehension of different programs version, *performance* and *ranking* would be better indicators of comprehension, whilst *time* could be considered as an inappropriate indicator of comprehension. Therefore, *time* will not be used further in this investigation. Groups of subjects given the *Object* based program outperformed (i.e., found the program easier to comprehend) those given the *Non-Object* based program by nearly a third. This is largely attributed to the *Problem Classes* knowledge category. In investigating the interaction between knowledge categories in different sets of groups, we could say that *State* knowledge category has a positive dominant effect in program comprehension for the all sets of groups (All group, *Non-Object* group, and *Object* group); *Problem Classes* knowledge category has also a positive dominant effect in program comprehension only for the *Object* group. However, it has a negative strong effect for the *Non-Object* group.

5.2 Case Study2: The Line-Edit Study

5.2.1 The Rationale of the Second Study

Considering the Car study, one possible contributory factor that was advanced to explain the ease with which the *Object* based program was comprehended was the type of solution decomposition (CD2) used in implementing the *Object* based program version. It is almost axiomatic that problem characteristics will influence the types of solution decomposition produced and hence the ease/difficulty with which a program is comprehended. This formed the basis for the second study. The Car problem example can be considered to be more amenable to OO program comprehension. However, laying on example programs that have precisely this characteristic could, unwittingly, lead to limiting the validity of the investigation (Alardawi et al, 2011a, b). To make the investigation more valid, and to identify characteristics of problems that their possible solutions are best comprehended in their OO form, care was taken not to limit the investigation to problems that are classified and used as common OO example programs, as mentioned in Chapter 4. A decision was made to conduct another empirical study that using problem with different characteristics. A Line-Edit problem was used in the second study. The problem chosen was considered appropriate to the needs of the investigation because it contains explicit references and can posses both *primitive* and *abstract* solution decompositions, thus could meet the above-mentioned requirement.

5.2.2 Aim

The aim of the study was to assess the ease of comprehension of OO program for a Line-Edit problem. This aim was similar to the Car study. However, the problem here is considered different in terms of its specification features.

5.2.3 Subjects

The complexity of the program versions produced for the purpose of the study (*Non-Objects* and *Object* based program versions of the Line-Edit problem) may require subjects to have a comparatively higher level of programming experience than the subjects of the Car study. For this reason, postgraduate software engineering students were recruited in this study. A total of 56 subjects, all from Sheffield Hallam University UK, were participated in the experiment.

Demographic data from the background questionnaire showed that the subjects' gender ratio was 89% males and 11% females. The average age was about 23 years. All subjects had previous programming experience, particularly in event-driven programming using VB and OO programming using C++ and JAVA. All subjects had completed 18 weeks studying OO programming when the experiment took place. The experiment was conducted over two lab sessions. Prior to each session, a stratified random sampling approach was used to match the subjects. Matching was based on the subjects' grades on the courses they were attending. The subjects were then randomly allocated to one of two matched groups (*Non-Object* and *Object* groups). For the two sessions, 28 subjects were allocated to the *Non-Object* group and 28 subjects to the *Object* group.

5.2.4 Materials and procedure

These were as for the Car study except that each subject was supplied with a copy of either the *Non-Object* based or *Object* based program version of the Line-Edit programs instead of the Car programs.

5.2.5 Metrics and Experimental Hypotheses

Metrics and hypotheses were as for first study.

5.2.6 Experimental Results of the Line-Edit Study

Normality test (see appendix d) showed that data to be non-normal. Corresponding non--parametric statistical tests were applied. The analysis was divided into four levels. The first and second levels were done to compare the *performance* and *ranking* between the program versions respectively, thus, to test the first null hypothesis. The third level aim to test the second null hypothesis, it was to compare the *performance* in each knowledge category between program versions. The last level of analysis aim to test the third null hypothesis, it was done to compare *performance* in knowledge categories for different sets of subjects' groups (All group, *Non-Object* group, and *Object* group).

5.2.6.1 Comparison of the Performance

The analysis accounted for the effect of the program version on *performance*. The descriptive analysis of the *performance* data is presented in table 5.23. Examination shows that the mean *performance* of the *Object* based group is higher than the mean *performance* of the *Non-Object* group. This indicates that there is program version effect in terms of *performance*

Table 5.23: Statistical summary of performance in the Line-Edit study

descriptive statistics					
	N	min	max	mean	SD
performance of *Non-Object* group	28	18.75	81.25	50.66	16.25
performance of *Object* group	28	50.00	93.75	69.86	9.77

In comparing all groups undergoing the two different Line-Edit programs versions (as independent variable), *performance* (as dependent variable), a Mann-Whitney U test was applied. The result revealed that there was a significant difference (U=131.500, p=.000<0.05 2-tailed). The *performance* of *Non-Object* group and *Object* group is significantly different supporting the H_{11}. Sums and means of the ranks for the subjects are shown in table 5.24 and the Mann-Whitney results are shown in table 5.25.

Table 5.24: Ranks of performance in the Line-Edit study

ranks

	program version	N	mean rank	sum of ranks
performance	Non-Object based	28	19.20	537.50
	Object based	28	37.80	1058.50
	total	56		

Table 5.25: Mann-Whitney test result of performance in the Line-edit study

test statistics

	performance
Mann-Whitney U	131.500
asymp. sig. (2-tailed)	.000

5.2.6.2 Comparison of Ranking

In assessing subjects' judgment about their comprehension of the given program version *(Non-Object* based and *Object* based), Figure 5.3 shows the means of subjects' responses to the ranking categories broken down by the program versions.

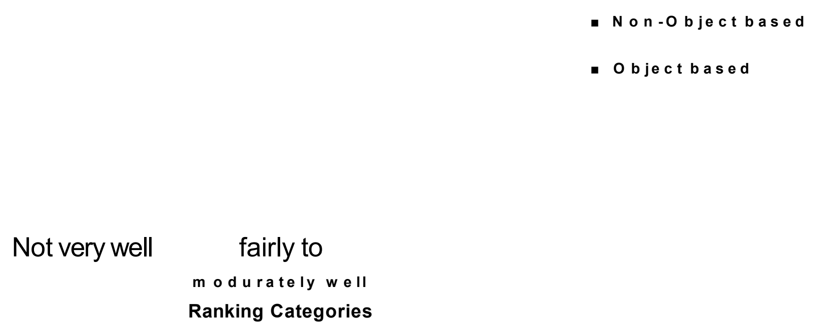

Figure 5.3: Graphical representation of the subjects' ranking of each program version in the Line-Edit study.

Examination of the data confirms that the *Object* based program was often assessed as being easier to comprehend than the *Non-Object* based program. More specifically, 39% of the *Non-Object* group ranked their comprehension as *Not very well*, while only 14% of the *Object* group graded comprehending their program as *Not very well.* The percentage of both *Non-Object* and *Object* groups in ranking their related program as *Fairly to moderate well* was similar. Only 17% of the *Non-Object* group graded their program as *Well to very well*, while 39% of the *Object* group graded their program in this ranking category. This gives substantial agreement and supports the above findings, by using *performance* measure, that the *Object* based program was easier to comprehend than the *Non-Object* based program. On this basis, the *Object* based program appears easier to comprehend if it is measured by *performance* and *ranking*. In comparing the comprehension of the subjects with *Object* based and *Non-Object* based programs versions, the results revealed that the *Object* based group outperformed the *Non-Object* based group. These results concurred with what was found in the Car study.

5.2.6.3 Comparison of Performance of Knowledge Categories between Program Versions

This analysis accounted for the effect of program version on each knowledge category. Figure 5.4 shows the mean of the correct responses to each knowledge category broken down by the programs versions.

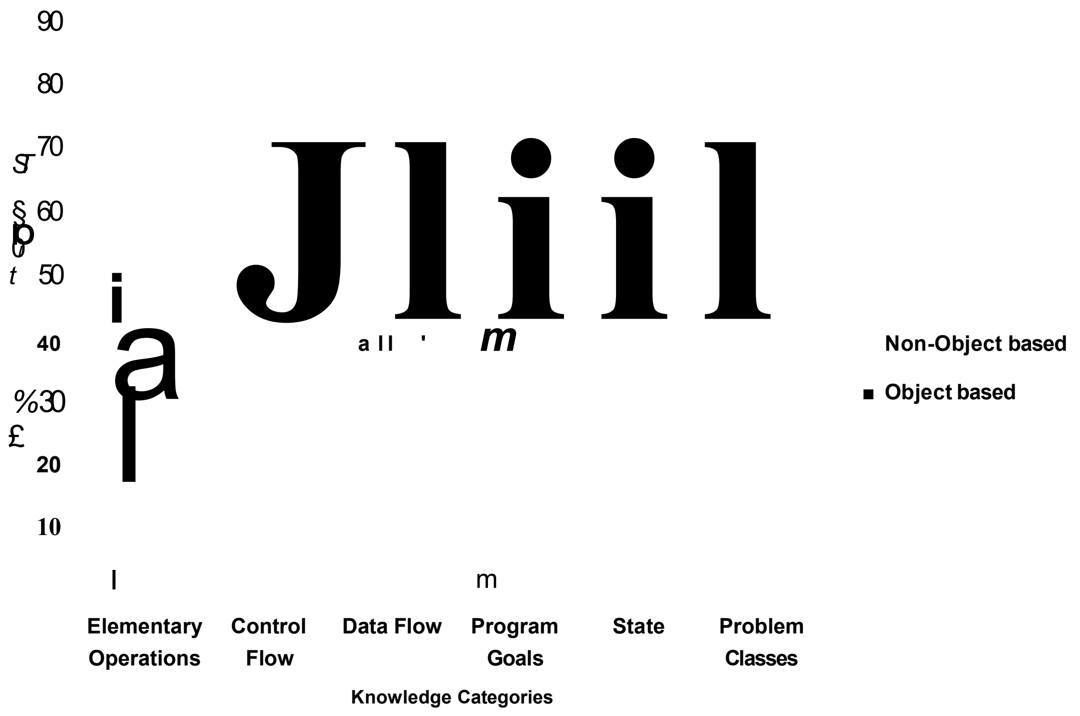

Figure 5.4: Graphical representation of performance in each knowledge category for each program version in the Line-Edit study

In comparing the two groups each of which undergoing one of the Line-Edit programs versions (as an independent variable) with *performance* in each knowledge category (as dependent variable), a Mann-Whitney U test was applied. The result revealed that *Control Flow, State,* and *Problem Classes* knowledge categories were significantly different supporting Hi2. The sums and means of the ranks for the groups are shown in table 5.26 and the Mann-Whitney results for each knowledge category are shown in table 5.27.

Table 5.26: Mean ranks of performances of each knowledge category between program versions of the Line-Edit study

ranks				
knowledge categories	program version	N	mean rank	sum of ranks
Elementary Operations	Non-Object based	28	27.32	765.00
	Object based	28	29.68	831.00
	total	56		
Control Flow	Non-Object based	28	23.84	667.50
	Object based	28	33.16	928.50
	total	56		
Data Flow	Non-Object based	28	30.25	847.00
	Object based	28	26.75	749.00
	total	56		
Program Goals	Non-Object based	28	26.89	753.00
	Object based	28	30.11	843.00
	total	56		
State	Non-Object based	28	23.82	667.00
	Object based	28	33.18	929.00
	total	56		
Problem Classes	Non-Object based	28	23.82	667.00
	Object based	28	33.18	929.00
	total	56		

Table 5.27: Mann-Whitney test results of performance of each knowledge category between program versions in the Line-Edit study

test statistics						
	Elementary Operations	Control Flow	Data Flow	Program Goals	State	Problem Classes
Mann-Whitney U	359.000	261.500	343.000	347.000	261.000	60.000
asymp. sig.(2-tailed)	.513	.010	.383	.388	.027	.000

A Hodges-Lehman indicated that *Object* group performed 30% better than the *Non-Object* group in the *Problem Classes* knowledge category.

In comparing the two groups for different program versions (*Non-Object* based and *Object* based) in terms of *performance* in each knowledge category, the results indicated that the *Object* group outperformed the *Non-Object* group by nearly a third in *Problem Classes* knowledge category. This was similar to the Car study, where the knowledge category contributing to the difference was the *Problem Classes* (i.e., nearly third more in the mean score). However, both *Control Flow* and *State* knowledge categories were also significant and contributed to the difference that was revealed in the Line-Edit study.

5.2.6.4 Comparison of Performance in Knowledge Categories

The level of analysis accounted for compare the knowledge categories in different sets of groups (All groups, *Non-Object* group, and *Object* group).

5.2.6.4.1 Comparisons of Performance in Knowledge categories for All Group

The descriptive analysis of data for All group's *performance* is presented in table 5.28. Examination shows that the mean *performance* of the *Program Goals* knowledge category is the highest among all other knowledge categories. However, *performance* of the *Problem Classes* knowledge category is the lowest among all other knowledge categories.

Table 5.28: Statistical summary of the performance in each knowledge category of All groups of the Line-Edit study

descriptive statistics					
knowledge categories	N	min	max	mean	SD
Elementary Operations	56	0.00	100.00	63.39	26.09
Control Flow	56	0.00	100.00	56.25	28.70
Data Flow	56	0.00	100.00	64.28	38.98
Program Goals	56	0.00	100.00	66.07	27.14
State	56	0.00	100.00	62.05	30.89
Problem Classes	56	0.00	100.00	54.02	31.54

A Kruskal-Wallis test, with all six knowledge categories questions (as independent variable) and *performance* in each knowledge category (as dependent variable), revealed no significant difference among knowledge categories ($\chi^2=9.320$, $p=.097>0.05$). The third null hypothesis H_{03} was accepted. Means of the ranks for All groups' *performance* are shown in table 5.29 and the Kruskal-Wallis results are shown in table 5.30.

Table 5.29: Ranks of All group's performance in each knowledge category of the Line-Edit study

ranks			
	knowledge categories	N	mean rank
performance for All group	Elementary Operations	56	157.26
	Control Flow	56	153.39
	Data Flow	56	187.04
	Program Goals	56	181.05
	State	56	181.91
	Problem Classes	56	150.35
	Total	336	

Table 5.30: Kruskal-Wallis test result of All groups' performance in each knowledge category of the Line-Edit study

test statistics	
	performance for All group
chi-square	9.320
df	5
asymp. sig.	.097

One can conclude in this case that, for All group, the *performance* in knowledge categories was not significantly different. Indeed, even the difference in the highest mean score in *Program Goal* (66%) and the lowest mean score in *Problem Classes* (54%) was marginal.

5.2.6.4.2 Comparisons of Performance in Knowledge categories for Non-Object Group

The descriptive analysis of data for *Non-Object* group's *performance* is presented in table 5.31. Examination shows that the mean *performance* of the *Data Flow* knowledge category is the highest among all other knowledge categories. However, the knowledge category *Problem Classes* represents the lowest *performance* of all other knowledge categories.

Table 5.31: Statistical summary of the performance in each knowledge category of *Non-Object* group in the Line-Edit study

descriptive statistics					
knowledge categories	N	min	max	mean	SD
Elementary Operations	28	0.00	100.00	60.71	28.40
Control Flow	28	0.00	100.00	46.42	26.97
Data Flow	28	0.00	100.00	67.85	41.30
Program Goals	28	0.00	100.00	62.50	29.26
State	28	0.00	100.00	52.67	31.43
Problem Classes	28	0.00	100.00	31.25	25.11

The Kruskal-Wallis test revealed a significant difference among knowledge categories (χ^2=24.55, p=.000<0.05). Means of the ranks for *Non-Object* group's' *performance* in each knowledge category are shown in table 5.32 and the Kruskal-Wallis results are shown in table 5.33.

Table 5.32: Ranks of *Non-Object* group's performance in each knowledge category of the Line-Edit study

ranks			
	knowledge categories	N	mean rank
performance for *Non-Object* group	Elementary Operations	28	94.68
	Control Flow	28	75.41
	Data Flow	28	104.61
	Program Goals	28	97.11
	State	28	83.39
	Problem Classes	28	51.80
	Total	168	

Table 5.33: Kruskal-Wallis test result of *Non-Object* group's performance in each knowledge category of the Line-Edit study

test statistics	
	performance for *Non-Object* group
chi-square	24.55
df	5
asymp. sig.	.000

Results revealed that, for the *Non-Object* group, there is a significant different in *performance* between knowledge categories. The *Problem Classes* knowledge category has a considerably lower mean score. A pairwise test was run as a follow-up.

Table 5.34: Pairwise comparison of *Non-Object* group's performance in each knowledge category of the Line-Edit study

Sample1-Sample2	**Test Statistic**	**Std. Error**	**Std.Test Statistic**	**Sig.**	**Adj.Sig.**
Problem Classes-Data Flow	52.804	12.175	4.337	.000	.000
Problem Classes-Program Goals	45.304	12.175	3.721	.000	.003
Problem Classes-Elementary Operations	42.875	12.175	3.522	.000	.006
Problem Classes-State	31.589	12.175	2.595	.009	.142
Control Flow-Data Flow	-29.196	12.175	-2.398	.016	.247
Problem Classes-Control Flow	23.607	12.175	1.939	.053	.788
Control Flow-Elementary Operations	19.268	12.175	1.583	.114	1.000
Program Goals-Data Flow	7.500	12.175	.616	.538	1.000
Control Flow-State	-7.982	12.175	-.656	.512	1.000
State-Elementary Operations	11.286	12.175	.927	.354	1.000
State-Program Goals	13.714	12.175	1.126	.260	1.000
Control Flow-Program Goals	-21.696	12.175	-1.782	.075	1.000
State-Data Flow	21.214	12.175	1.742	.081	1.000
Elementary Operations-Data Flow	-9.929	12.175	-.815	.415	1.000
Elementary Operations-Program Goals	-2.429	12.175	-.199	.842	1.000

From table 5.34, the result in comparing *performance* between knowledge categories for the *Non-Object* group revealed that there was a significant different between knowledge categories. Moreover, investigation of these knowledge categories' contributions and their interactions revealed that

Problem Classes knowledge category has a significant interaction with the *Elementary Operations*, *Data Flow*, and *Control Flow* knowledge categories. This was expected given the considerable lower value of the mean *performance* score of *Problem Classes* found in table 5.31. Therefore, *Problem Classes* once again plays a significant negative role in program comprehension for the subjects using the *Non-Object* based program version of the Line-Edit problem.

5.2.6.4.3 Comparisons of Performance in Knowledge categories for Object Group

The descriptive analysis of data for *Object* group's *performance* is presented in table 5.35. Examination shows that the mean *performance* of the *Problem Classes* knowledge category is the highest among all other knowledge categories. However, *Data Flow* knowledge category represents the lowest *performance* than all other knowledge categories.

Table 5.35: Statistical summary of performance in each knowledge category of *Object* group of the Line-Edit study

descriptive statistics					
knowledge categories	N	min	max	mean	SD
Elementary Operations	28	50.00	100.00	66.07	23.77
Control Flow	28	.00	100.00	66.07	27.39
Data Flow	28	.00	100.00	60.71	36.91
Program Goals	28	50.00	100.00	69.64	24.86
State	28	25.00	100.00	71.42	27.81
Problem Classes	28	50.00	100.00	76.78	17.90

The Kruskal-Wallis test, with "knowledge categories" as the independent variable and *performance* in each knowledge category as dependent variable, revealed a non-significant difference among knowledge categories (χ^2=5.22, p=.389>0.05). Means of the ranks for *Object* group's *performance* are shown in table 5.36 and the Kruskal-Wallis results are shown in table 5.37.

Table 5.36:Ranks of *Object* group's performance in each knowledge category of the Line-Edit study

ranks			
	knowledge categories	N	mean rank
performance for *Object* group	Elementary Operations	28	76.25
	Control Flow	28	79.64
	Data Flow	28	78.11
	Program Goals	28	84.46
	State	28	89.11
	Problem Classes	28	99.43
	Total	168	

Table 5.37:Kruskal-Wallis test result of *Object* group's performance in each knowledge category of the Line-Edit study

test statistics	
	performance for *Object* group
chi-square	5.22
df	5
asymp. sig.	.389

In comparing *performance* of knowledge categories for the *Object* group, we can conclude that there is no significant difference. The highest mean score of *Problem Classes* (77%) and the lowest mean score of *Data Flow* (61%) means that there is a marginal difference.

In summarising the findings of Line-Edit Study, we can conclude that the group given the *Object* based program version outperformed the group given the *Non-Object* based program version. This outperformance is largely attributable to the *Problem Classes* and perhaps *Control Flow* and *State* knowledge categories. Investigating the interaction between knowledge categories in different sets of subjects groups, we can conclude that the most significant negative effect in program comprehension was found in the *Non-Object* group and was caused by the *Problem Classes* knowledge category. However, the interaction between knowledge categories in the All group and the *Object* group was found to be marginal.

5.2.7 Summary of the Investigation Results

This chapter reported two sets of empirical studies (Car and Line-Edit) assessing the ease of comprehension of OO programs with different sets of problem characteristics and different solution decompositions. Both *performance* and *ranking* were found to be good indicators. However, *time* was not a good indicator for measuring the ease of comprehension and was therefore omitted from the investigation. Table 5.38 details the summary of the findings from both Car and Line-Edit studies.

Table 5.38: Summary of the studies' findings

studies	**Car study**		**Line-Edit study**	
Performance	Object based group outperformed the Non-Object based group		Object based group outperformed the Non-Object based group	
Knowledge categories that contributed to the difference between program version	Problem Classes		Control Flow State Problem Classes	
Performance in each knowledge category	positive dominant	negative dominant	positive dominant	negative dominant
All group	State	none	none	none
Non-Object group	State	Problem Classes	none	Problem Classes
Object group	State Problem Classes	none	none	none

Looking at the second and third row in table 5.38, clearly, the similarity between the two studies indicates that the *Object* group outperformed (i.e., found the program easier to comprehend) the *Non-Object* group in both studies. The knowledge categories contributing to these differences differ in each study. In the Car study, *Problem Classes* knowledge category is the highly contributor to the difference between program versions. However, *Control Flow*, *State*, and *Problem Classes* knowledge categories are high contributors to this difference in the Line-Edit study.

In investigating *performance* in each knowledge category, represented in the last three rows in the above table, it was found that, in the Car study, the *State* knowledge category has a highly positive dominant effect upon *performance* for All groups. However, when considering them separately, for the *Non-Object* group, *State* knowledge category had a positive dominant effect whilst *Problem Classes* knowledge category had a negative dominant effect. In contrast, for the *Object* group, *State* and *Problem Classes* knowledge categories had a positive dominant effect. We can therefore clearly infer that the *State* knowledge category is important and plays a key role in comprehension for both *Non-Object* and *Object* based program versions. Moreover, *Problem Classes* knowledge also plays a key role in comprehension, but it has opposite effects on the two program versions. For the Line-Edit study, we found that there was no significant difference for All group or *Object* group. However, for the *Non-Object* group, *Problem Classes* knowledge category has the greatest negative effect.

In conclusion, the *Problem Classes* knowledge category contributes to the difference in comprehension between groups in both studies, whilst *Control*

Flow and *State* knowledge categories contribute to this difference only in the Line-Edit study. In terms of ease of comprehension of knowledge categories, it is clear that the first study indicates that *Problem Classes* has a positive effect on the *Object* based program and a negative effect on the *Non-Object* based, whilst *State* knowledge category has a positive effect in both program versions. The counterpart of this in the second study is that *Problem Classes* has a negative effect only on the *Non-Object* based program.

Chapter 6 Discussion

6.1 Introduction

The investigation has focused on whether OO programs are easier to comprehend than non OO programs. The investigation considered the influence of *class concept*, *problem characteristics*, and *solution decompositions* on the comprehension of different sets of knowledge categories. In order to achieve this, the investigation was built on and adapted from existent empirical works in the field of OO program comprehension.

The obtained findings are positive in extending and tailoring empirical works, based on established principles of the scientific method, done by Wiedenbeck and Ramalingam (1999) for the Car problem and by Siddiqi (1984) for the Line-Edit problem. The adaptation allowed us to improve the experimental materials to carry out this investigation. For example, incorporating three classes in the Car problem and implementing *abstract* solution decomposition of the Line-Edit problem in the form of *Object* based program helped in investigating the effect of *class concept* better and brought interesting findings to the field of empirical studies of OO program comprehension. Using different sets of problems allowed us to investigate the influence of *problem characteristics* in OO program comprehension. Moreover, utilising Siddiqi's different solution decompositions of the Line-Edit problem helped in investigating the influence of *solution decompositions* in comprehension of OO programs. This has also made it possible to investigate these solutions in the sense of a program being comprehended rather than a program being designed. The investigation has also obtained a rich view of the comprehension of a set of different types of knowledge, which are taken from the best-known models of program comprehension in this field (Pennington, 1987a, Burkhardt et al 2006a, b). The shift in emphasis from a memory-based task to a search-based task, which was considered necessary for the needs of this investigation, has also helped in obtained these findings. The investigation was also able to use and evaluate different measures of comprehension; these are *time*, *performance*, and *ranking*. These measures were used in different related studies. All the above adaptations have improved the experimental materials used in this investigation. These improvements have allowed us to relate the investigation findings to a wide range of related studies. Thus, the findings are interesting and add to the body of knowledge about empirical work on OO program comprehension.

Section 6.2 interprets the findings of the two empirical studies reported in Chapter 5 and evaluates the model used in this investigation. Section 6.3 discusses the proposed empirically grounded based model of OO program comprehension along with the limitations of the model. Section 6.4 then discusses methodological limitations of the studies conducted and how they might affect the findings. Finally, possible pedagogical issues are highlighted in section 6.5.

6.2 Interpretation of the Studies' Findings

The investigation assessed comprehension using *time*, *performance*, and *ranking* measures. In terms of *time*, it was found that this was not a good indicator in measuring comprehension. This was similar to Siddiqi's finding (Siddiqi, 1984). *Performance* has been widely used by most related studies (for example, Pennington 1987a; Ramalingam and Wiedenbeck, 1997; Wiedenbeck et al., 1999; Wiedenbeck and Ramalingam, 1999; Good, 1999; Khazaei and

Jackson, 2002, Burkhardt et al., 2006a, b; Affandy, 2011). It is found that *performance* is a better indicator of comprehension in the studies reported in this thesis too. The investigation has also found *ranking* as a good indicator and has provided substantial supporting indicators for the finding of *performance*.

Assessing the ease of comprehension of OO programs, the investigation found that, despite the variation in the problems' characteristics between Car and Line-Edit studies, the *Object* based programs were easier to comprehend than the *Non-Object* based programs. The overall comprehension was advantageous to *Object* based programs.

Interpreting this finding in terms of building mappings between program and problem domains, this finding supports Détienne's (2006a) claim about the ease of comprehension of the OO approach. However, the comprehension process involves application of a number of different types of knowledge. It is, therefore, difficult to ascertain this claim from the overall comprehension. In order to assess such a claim, it is more reasonable to consider these different types of knowledge rather than the overall comprehension of the programs. This work provides supporting empirical evidence as it is found that comprehension of certain types of knowledge is easier than of the other types of knowledge. These better-comprehended types of knowledge are highlighted and discussed further in this section. Looking to other related studies, our finding does not always concur. While Khazaei and Jackson (2002) found that comprehension of OO and event-driven programs have a lot in common, Ramalingam and Wiedenbeck (1997), Wiedenbeck and Ramalingam (1999), and Wiedenbeck et al., (1999) found OO programs are more difficult to comprehend than their corresponding imperative and procedural programs. In their explain of their findings, they argued that their findings may be more a reflection that the types of comprehension questions used have more meaning for procedural programs than for OO programs rather than an indication of difficulty of comprehension in OO programs.

The rest of this section is divided into two parts. It first discusses the types of knowledge that contributed most to the difference between *Object* based programs and *Non-Object* based programs. The second part evaluates the model used in this investigation.

Problem Classes knowledge was found to contribute most in showing the difference in comprehension between the program versions in both studies. Moreover, *Control Flow* and *State* types of knowledge also contributed to this difference, but only in the Line-Edit study.

The knowledge that contributed most was the *Problem Classes.* The high performance in *Problem Classes* knowledge in the *Object* based programs reflects how easy it was to comprehend this related knowledge from the program text in both studies. This supports Burkhardt et al.'s (2006a, b) findings. This high *performance* could be attributed to the clarity of classes' declarations and their related attributes in the *Object* based programs, which in turn makes *Problem Classes* knowledge easier to comprehend regardless of the variations between the problem characteristics and the solution decompositions used in each study. The *Problem Classes* knowledge of the *Non-Object* based programs was the most difficult knowledge to comprehend in both studies. This finding represents the main similarity between the studies in terms of the negative effect on comprehension. It was assumed that comprehension of *Problem Classes* knowledge in the *Non-Object* based programs is more related to the understanding of problems' entities. Thus, the problem characteristics could be a factor that plays a role in the ease of comprehension of *Problem Classes* knowledge. On this basis, comprehension of *Problem Classes* knowledge would be easier in the Car study, where *car*, *engine*, and *body* are considered more tangible, than of Line-Edit study, where *word* and *buildingword* can be argued to be relatively intangible. The findings support this assumption; it was found that the *Non-Object* group in the Car study performed better than the *Non-Object* group in the Line-Edit study.

The second greatest contribution was made by *Control Flow*. This type of knowledge was found easier to comprehend in the *Object* based program than the *Non-Object* based program in the Line-Edit study. This finding is the opposite to what was found in Ramalingam and Wiedenbeck's (1997), Wiedenbeck and Ramalingam's (1999), and Wiedenbeck et al.'s (1999) studies. These studies assume that the control flow is mainly based on the program execution order (sequential vs. non-sequential). They found comprehension of execution of non OO programs was better than of the OO programs. Building on our finding, the difference in comprehension of *Control Flow* knowledge could reasonably be attributed to the difference in control structures used in designing the Line-Edit program versions (*Non-Object* and *Object* based). Since the solution decomposition (*primitive* vs. *abstract*) used in implementing each program version was different, the selection of the control structure was mainly influenced by this difference. We found that the more *abstract* the solution decomposition, which represents the *Object* based program, the easier it was to comprehend the *Control Flow* knowledge. Thus, it could be argued that not only the execution order of the program affects the comprehension of *Control Flow* knowledge, but also the type of solution decomposition used in designing the program versions has a significant influence.

The last type of knowledge found which contributed to comprehension is the *State* knowledge. It also played a role in the ease of comprehension of *Object* based programs in the Line-Edit study. This finding contradicts other related studies. For example, Ramalingam and Wiedenbeck (1997) and Wiedenbeck and Ramalingam (1999) argue that the difficulty in comprehension of *State* knowledge is almost attributed to the indirect representation of this knowledge in the program text. It is worth mentioning here that these studies used programs written in C++. Khazaei and Jackson (2002), who had used VB and JAVA in their programs, attributed the ease of comprehension of the *State* knowledge to the nature of the *State* comprehension questions used in their study. These questions asked about the state change for a specific variable at the time when a certain action occurred, the action involves the output statement in relation to the value of the variable within a conditional statement. The authors argue that these questions were relatively easier to answer as they can easily be spotted in the program text. With respect to all these, it is reasonable to argue that the ease of comprehension of *State* knowledge in the *Object* based program found in this investigation is possibly more attributable to the high readability nature of the structure of the *Object* based programs over the corresponding *Non-Object* based programs, rather than to the indirect representation of the knowledge or to the nature of the *State* questions. More precisely, the relatively easier control structure used in the *Object* based program, which is influenced by the *abstract* type of solution decomposition, made tracking the program's certain action and the changes on its associated variable easier in the *Object* based program than the *Non-Object* based program, which is influenced by the *primitive* type of solution decomposition.

From all discussed above, it seems that *Object* based programs are easier to comprehend than their corresponding *Non-Object* based programs for *Control Flow*, *State*, and *Problem Classes* types of knowledge. These types of knowledge seem to be significant in showing the difference in ease of comprehension between program versions.

In evaluating models of program comprehension, most related studies base their evaluations on distinguishing between two distinct but interrelated models. We called this the "*two-stage model*" of program comprehension. These models are: program model and situation model. Each model combines different sets of knowledge. While the program model encompasses knowledge related to *Elementary Operations* and *Control Flow,* the situation model combines knowledge related to *Data Flow* and *Program Goals, t*he *State* knowledge fall in between these program and situation models. In order to evaluate the model used in this investigation, we did not follow the *two-stage model* approach. Instead, we treated each type of knowledge individually. The advantage of this approach is that we can then incorporate the most significant types of knowledge that play a role in the ease of comprehension of OO program into a new proposed model. There is less of a need to include the non-significant types of knowledge.

In the Car study, it was found that *State* is the most easily comprehended knowledge which positively affected the comprehension of both our program versions. *Problem Classes* knowledge was found as the least comprehended knowledge which negatively affected the comprehension of our *Non-Object* based program. However, *Problem Classes* knowledge was the most easily comprehended knowledge which positively affected the comprehension of the *Object* based program. The different effects of the *Problem Classes* in different program versions were expected. For the *Object* based program, the knowledge of classes is directly relevant, whilst this knowledge is less so in the *Non-Object* based program. Therefore, *State* and *Problem Classes* are the two important types of knowledge that played a significant role in program comprehension. Our findings regarding *State* are consistent with Khazaei and Jackson's (2002) findings, whilst they somewhat contradict Ramalingam and Wiedenbeck (1997) and Wiedenbeck and Ramalingam (1999) findings. It is worth mentioning that programming languages used in the Car study were similar to that used in Khazaei and Jackson (2002) but were different from what was used in Ramalingam and Wiedenbeck (1997) and Wiedenbeck and Ramalingam (1999). Thus, it could be argued that the difference in the programming languages could be a factor that has affected the comprehension of *State* knowledge. The program listings and the *State* comprehension questions used in all of the related studies as well as our own Car study were the same.

In the Line-Edit study, it was found that *Problem Classes* knowledge was the least comprehended knowledge and it negatively affected the comprehension of the *Non-Object* based program. This was also expected, as *Problem Classes* knowledge is probably less relevant in the *Non-Object* based program. This also consistent with the findings in the Car study, where *Problem Classes* negatively affected the comprehension of the *Non-Object* based program. In terms of the rest of the types of knowledge, although *Control Flow* and *State* knowledge categories were significant in showing the difference in comprehension between *Non-Object* based and *Object* based programs, none of these types of knowledge, as well as the rest of types of knowledge, was found to contribute significantly to the comprehension in the Line-Edit study.

To summarise, in comparing the ease of comprehension between program versions, *Control Flow*, *State*, and *Problem Classes* significantly contributed to this difference with advancing to *Object* based program version. In evaluating the model used in this investigation, *State* and *Problem* Classes are the most dominant types of knowledge that affected comprehension. Therefore, it can be argued that *Control Flow*, *State*, and *Problem Classes* knowledge are the most important types of knowledge that should be taken into consideration in proposing a new model of OO program comprehension. The next section discusses the proposed empirically grounded based model of OO program comprehension along with its limitations.

6.3 An Empirically Grounded based Model of OO Program Comprehension

Software practitioners and human factor researchers, whose goal is that of facilitating the programmers' task, have used notions from cognitive psychology, problem-solving, and text understanding to produce models of programmer behaviour for various programming-related tasks. For instance, Shneiderman and Mayer (1979) propose a syntactic/semantic model of programmer behaviour, Brooks (1983) introduced a conceptual model of program comprehension, and Pennington (1987a, b) and Burkhardt et al. (2006a, b) propose mental models of program comprehension. Although these models have provided good frameworks in the field of program comprehension, empirical works that attempted to assess comprehension and validate these models have been shown to be simplistic and insufficient in different contexts. For example, empirical works done to evaluate Pennington's model showed its inability to account for the OO programming approach (See, for example, Ramalingam and Wiedenbeck, 1997; Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al., 1999; and Khazaei and Jackson, 2002).

The model proposed here is based on the synthesis of previous models and is enriched by the findings of the empirical investigation reported in this thesis. This investigation raised a number of novel issues in OO program comprehension that had previously remained unexplored. In terms of relevance to the theory of program comprehension, the types of knowledge used in this investigation, which are found in Pennington's (1987a) and Burkhardt et al.'s (2006a, b) models of program comprehension (see table 4.2), have provided good frameworks to assess the ease of comprehension of OO programs. More precisely, the types of knowledge that were found to contribute significantly to comprehension were *Control Flow*, *State*, and *Problem Classes*. These types of knowledge will be the basis for the proposed model of OO program comprehension.

6.3.1 Formulation of the Model

The starting point for formulating the model is the diagram of software comprehension models provided by O'Brien (2003). He contended that, despite the variations in emphasis between comprehension models, all of them consist of four common elements, namely, a 'knowledge base', a 'mental model', 'external representation', and some form of 'assimilation process'. Figure 6.1 illustrates these elements and shows how they relate to each other.

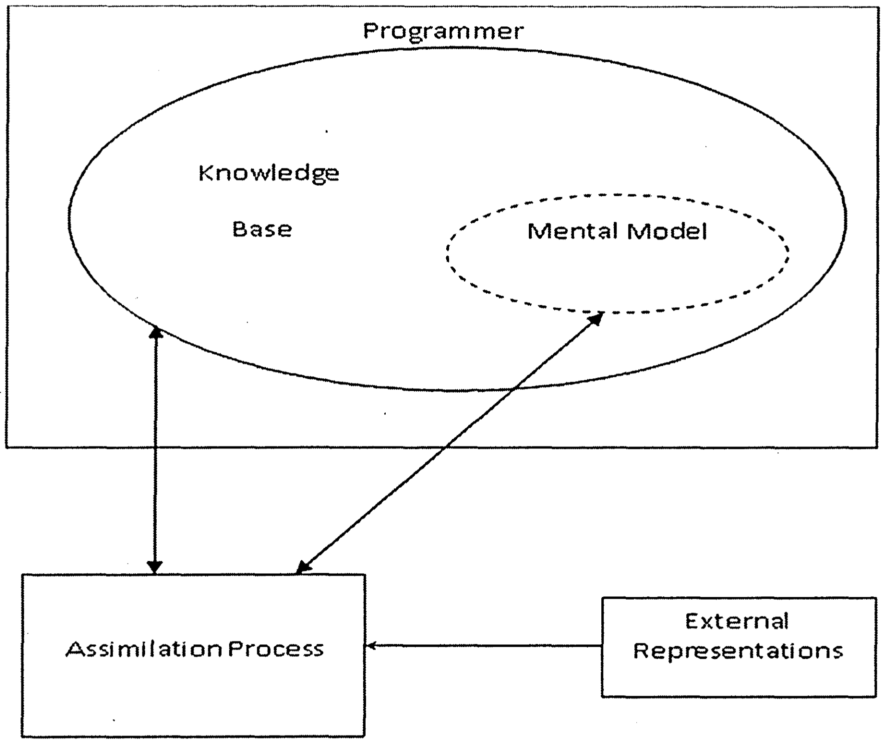

Figure 6.1: Elements of software comprehension models (source: O'Brien, 2003)

An operational overview of the model in the extension to our experimental settings can be defined as follows:

- the knowledge base is defined as programmers' previous knowledge before they comprehend the given code. This knowledge may consist of previous programming experience and domain knowledge;
- the mental model refers to the programmers' current understanding of the program code. It represents the output of the model and encompasses different types of knowledge. In the context of this thesis, these are: *Elementary Operations, Control Flow, Data Flow, Program Goals, State,* and *Problem Classes.* These types of knowledge have contributed differently in program comprehension. They provide detailed descriptions of different aspects of the programs;
- external representations can essentially be defined as any external supports in the form of system documentation, advice from other programmers

familiar with the problem domain, or, indeed, program code itself. External representations represent the input to a model. In our experimental settings, it is known as *treatments* which consist of the different program versions and their corresponding lists of comprehension questions used in the Car and the Line-Edit studies. Three more elements are incorporated into the *treatments*. These are: *class concept*, *problem characteristics*, and *solution decompositions*. The *class concept* was investigated by being present or absent in the *treatments*. Two problems with different characteristics were used in the *treatments*. These are represented as *trivial* in the Car study and as *rich* in the Line-Edit study. For the Car study, there was only one possible solution decomposition, which is *primitive*. The solution was implemented in two different forms (*Non-Object* based program and *Object* based program). In the Line-Edit study, two alternatives solution decompositions, which are *primitive* and *abstract,* were implemented in two different forms. The *primitive* type of solution decomposition was implemented as *Non-Object* based program and the *abstract* type of solution decomposition was implemented as *Object* based program. Thus, the input to the model is two different programs versions for each problem and *solution decompositions* is therefore depicted on the model as *treatments*;

- the assimilation process is defined as the actual strategy the programmer employs to comprehend the program. Thus, the assimilation process depicts the program comprehension process that yields a description of text-to-be-understood. This comprehension process represented as subjects *performance* of the six types of knowledge mentioned above.

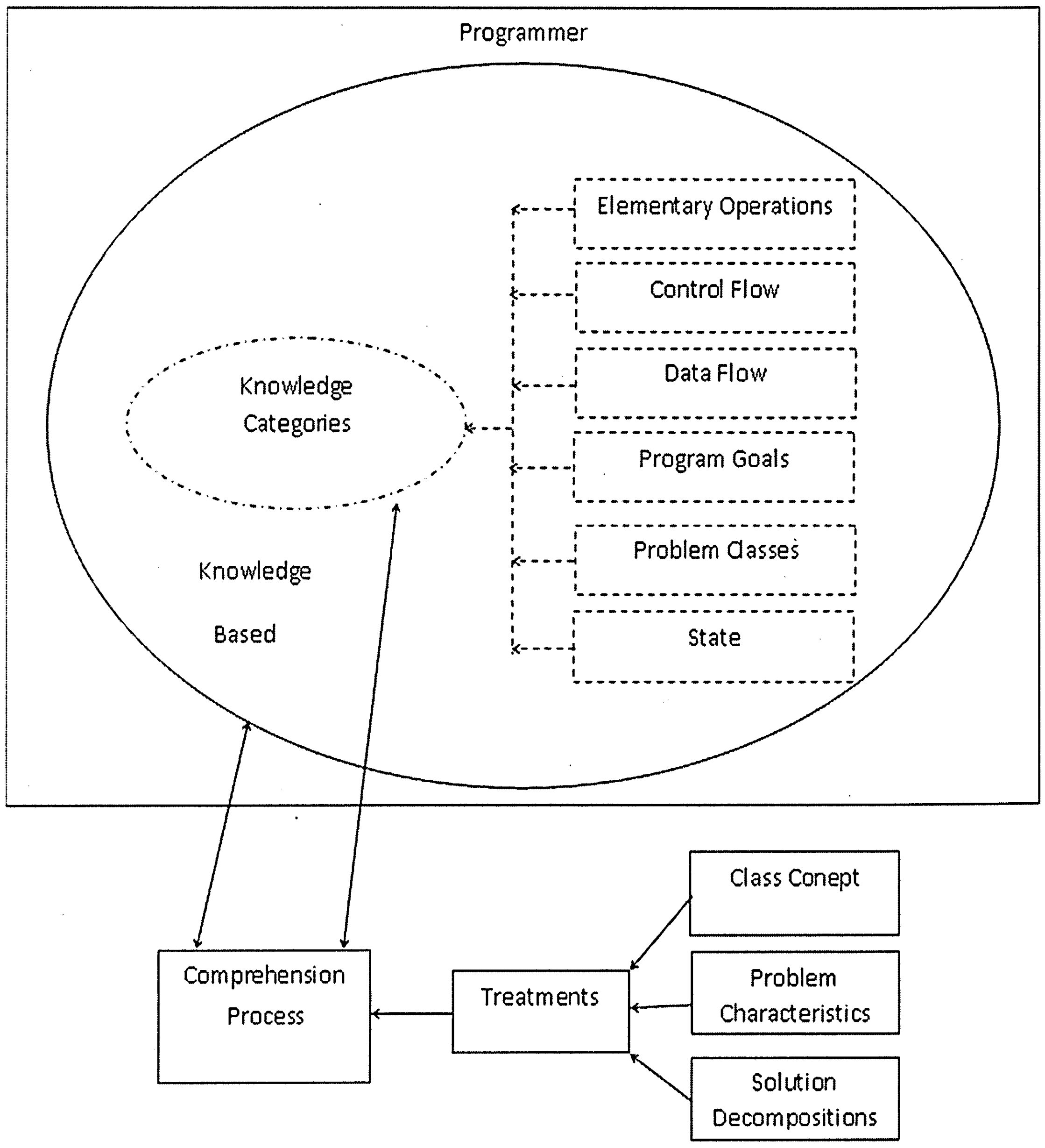

Figure 6.2The operational view of the model used in this investigation

Figure (6.2) illustrates the tailored operational view of the model used in this investigation. The model is intended to describe the influence of *treatments* on the comprehension process. The *treatments* are: *class concept*, *problem characteristics*, and *solution decompositions.* The comprehension process represents the contributions of different types of knowledge to comprehension. These are: *Elementary Operations, Control Flow, Data Flow, Program Goals, State,* and *Problem Classes*. The influences of the *treatments* are captured in the next few paragraphs.

For the *class concept*:- the classes play a key role in facilitating comprehension of OO programs. The key feature of a class is that its special structure and representation carries a lot of the meaning for the program text. Classes can easily be seen in the text of an OO program. Just skimming through a program listing leads to identifying classes during comprehension. The evidence for this was reported in both studies, where the *Problem Classes* contributed most to ease of comprehension of *Object* based programs.

In terms of the *problem characteristics:* - this element has shown a strong influence in facilitating the comprehension of *Problem Classes* knowledge. A *problem characteristic* was a significant factor in recognising the classes, their boundaries, their static data members, and their related functions. This recognition was more evident in the Car study, where the problem entities were relatively tangible. In the Line-Edit study, the problem entities were relatively intangible. Our findings showed that knowledge related to *Problem Classes* of *Object* based programs in the Car study was easier to comprehend than in the Line-Edit study. Moreover, comparing ease of comprehension of the *Non-Object* based programs in both studies; there is also evidence about the effect of the tangibility of the problem entities on comprehension of *Problem Classes* knowledge. Our findings showed that the comprehension of *Problem Classes* knowledge was easier in the Car study than in the Line-Edit study. Thus, a *problem characteristic* is considered to be an important element in the proposed model of OO program comprehension.

In terms of *solution decompositions:* - there is also evidence to suggest that there is a great influence of this element on comprehension of the OO program in the Line-Edit study. Implementing *abstract solution decomposition* led to facilitating comprehension of the *Object* based program. The *Non-Object* based program, implemented based on *primitive solution decomposition*, was less easy to comprehend. The *abstract* solution made tracking the flow of the program execution and its associated changes in the program actions easier in the *Object* based program. The evidence for this is that knowledge related to *Control Flow* and *State* in the *Object* based programs was comprehended better. Therefore, *solution decomposition* was found empirically to be influential. Thus, it is another important element that should be included in the proposed model of OO program comprehension.

To summarise, the *treatments* incorporated in the model have shown a strong influence on comprehension of OO programs over non OO programs. Based on the empirical evidence found in both studies, we can conclude:

- where the *problem characteristics* are *trivial*, the *primitive Object* based solution type is comprehended better than the *primitive Non-Object* based solution type. This especially influences comprehension of the *Problem Classes* knowledge;
- where the *problem characteristics* are *rich*, the *abstract Object* based solution type is comprehended better than the *primitive Non-Object* based solution type. This especially influences the comprehension of *Control Flow, State,* and *Problem Classes* knowledge.

However, these *treatments* did not show a strong influence on comprehension of the other types of knowledge in our studies. These are: *Elementary Operation, Data Flow*, and *Program Goals.* Therefore, in proposing a new empirically grounded based model of OO program comprehension, shown in figure 6.3, the following points were considered:

- incorporating *class concept*, *problem characteristics*, and *solution decompositions* as *treatments* is important as they were shown to be influential on comprehension of OO programs. These are highlighted as bold in the new model;
- the *Control Flow*, *State*, and *Problem Classes* should be considered as "*primary*" types of knowledge. They showed significant difference in comprehension between different program versions (*Non-Object* based and *Object* based). They also contributed significantly and showed positive effects on the comprehension of OO programs. These *primary* types of knowledge are highlighted as bold in the new model;
- *the Elementary Operations, Data Flow*, and *Program Goals* should be considered as "*secondary*" types of knowledge. They did not show any significant difference in comprehension between different program versions. Thus these *secondary* types of knowledge are not highlighted bold in the new model as they would be considered less important.

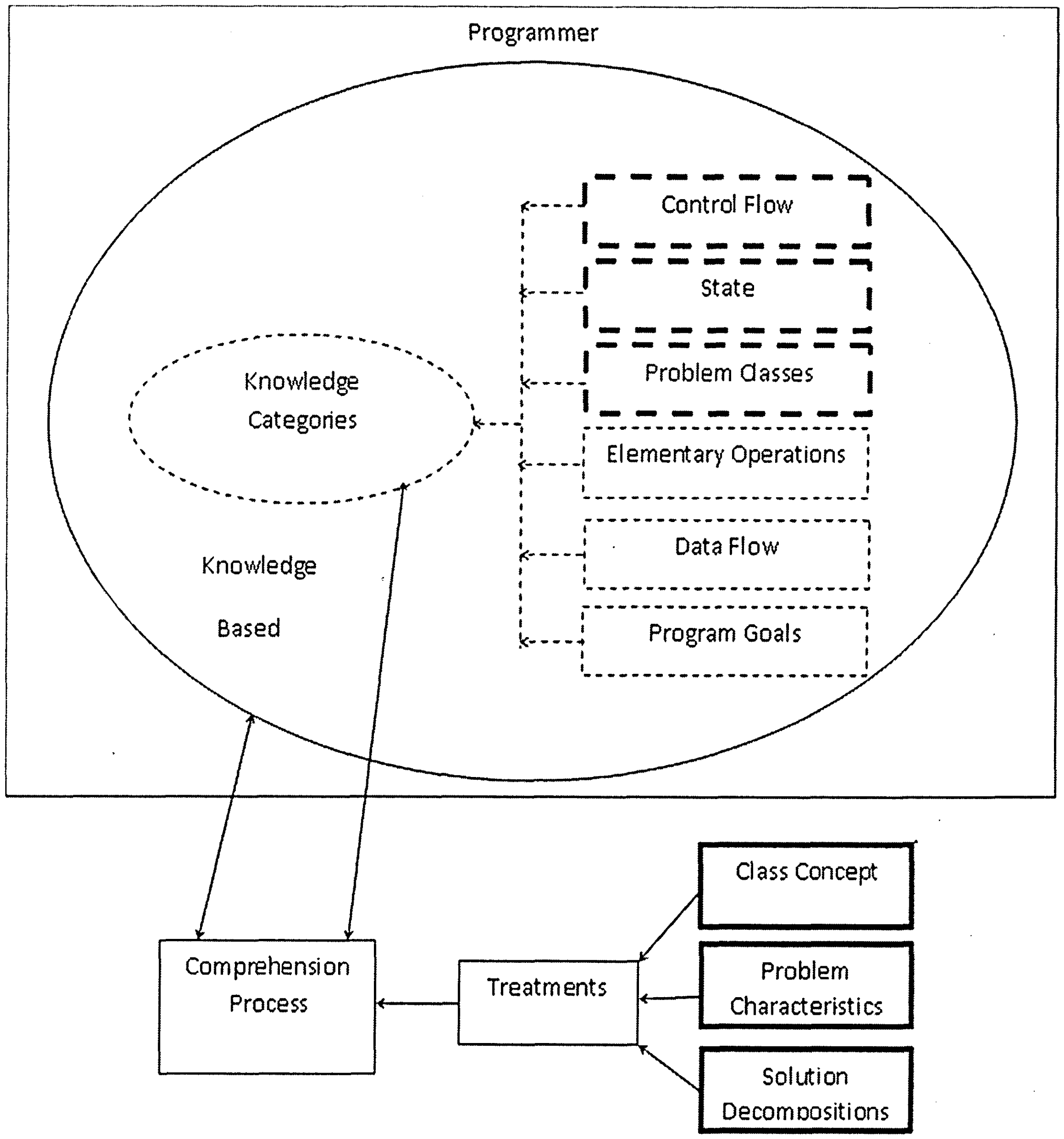

Figure 6.3: The empirically grounded based proposed model of OO program comprehension

The model proposed here is empirically based and could provide a good framework to the field of empirical studies of OO program comprehension. It could be a starting point for further empirical work in this field. However, as in any model of program comprehension, this new model can also be considered simplistic, for any other context and findings are only limited to our experimental settings. The next section elaborates on this limitation and suggests an extension to the proposed model.

6.3.2 Limitation and Possible Extension of the Proposed Model

The contributory types of knowledge found in this investigation may be sufficient to account for comprehension of OO programs used here. However, it may be simplistic for these types of knowledge to be described as the "*critical aspects*" of OO programs. In order to achieve a thorough understanding of OO program comprehension, there is a need to review *Control Flow* knowledge and expand its definition from "a sequential execution of the program" to "the way in which objects interact with each other". Burkhardt et al. (2006, b) defined this interaction as the dynamic aspects of the *Problem Classes* and any other objects used in the program. The dynamic aspects can be represented as client-server relationships via message passing, objects composition, and inheritance relationships. These aspects were introduced and empirically tested by Burkhardt et al. (2006a, b). There is also a need to reconsider *State* knowledge not only as "a state of a specific variable" but also as a "state of specific object", where this object is holding state and changing through its behaviour. The proposed model has limits in describing these *critical aspects* of *Control Flow*, *State,* and *Problem Classes*.

An extended model should probably distinguish between these *critical aspects*. Developing an extended model that includes these *critical aspects* in the form of a set of knowledge categories will be difficult if we base it on empirical investigation that compare non OO and OO program versions. More precisely, the questions relating to *critical aspects* of *Control Flow*, *State*, and *Problem Classes* knowledge can only be limited to those spanned in *Object* based programs but not included in the *Non-Object* based programs. In order to go further with investigating these types of knowledge based on the broad definition given above, there is a need to realise the limitation of experiments comparing OO programs with non OO programs. This proposal is that a further empirical study must compare two OO programs for these critical aspects.

The next section discusses methodological issues raised from this investigation and how they might affect the investigation's findings.

6.4 Methodological Issues of the Investigation

Adopting Pennington's (1987a) and Burkhardt et al.'s (2006a, b) models of program comprehension in this investigation proved successful, yet devising a specific tailored experimental methodology to empirically assess the ease of comprehension of OO program was not easy and needed very careful consideration of many different issues. While the investigation was able to assess the ease of comprehension of OO programs, some methodological issues may have played a role in this assessment. Two methodological issues are highlighted in this investigation: these are: the lack of a criterion for comparability between program versions, and the use of additional cues. This section discusses these issues and the way in which they may have affected the investigation's findings.

In conducting an empirical study, it is important to illuminate the effect of the *extraneous* variables on the outcome of the study. We were able to keep all the variables constant except the variables under investigation (i.e., keep comprehension questions the same in both program versions). However, one methodological issue which may be questioned is the lack of a criterion for comparability between the program versions (*Non-Object* base and *Object* based) used in each study. Since the programs were developed based on the existence/absence of class, there was no criterion of comparability to be used to argue about their "equivalence". The *Object* based programs were systematically slightly longer than the corresponding *Non-Object* based programs due to the overhead of declaring classes. Thus, such variation can be difficult to avoid. However, all corresponding programs were equivalent in terms of their functionality. Considering the comprehension questions used in each program version, the criterion of their comparability was based on the availability of related knowledge in both program versions, thus, except for the *Problem Classes* questions, the same comprehension questions were asked in both program versions. Although the investigation was able to produce equivalent counterpart questions for all types of knowledge under investigation, the equivalence of comprehension questions related to the *Problem Classes* knowledge is questionable.

Meaningful variable names and comments were used in the program versions of the two studies. Additionally, as the task here is search-based, the programs were available to subjects during the experiment session. This is a distinction different from the memory-based task used by other related studies. By the time, subjects may well have formulated prior ideas about what the program did and how it worked; for example, one expectation could stem from the variables names, such as, "*speed*", "*passengers*", "*wordlength*", and "*newcharacter*" and comments, such as, "*assign the speed value of the Car*" and "*output the word on current or new line*". Good (1999) argued that the availability of the program gave subjects an additional source of information on which to base their comprehension, by using the program text more as part of a hypothesis verification process than anything else. This in itself does not necessarily explain why differences in comprehension between program versions were found and it was difficult to avoid. Using these additional cues in the experimental settings could also be an issue that influences the findings. The possible solution to this problem was to include the approach of using non-meaningful variable names and the approach of fewer textual comments about the program. However, our pilot study suggested that introducing such approaches was considered difficult and found beyond the ability of our subjects. These approaches may be possible directions for future investigation, provided that subject group is made up of more experienced programmers.

6.5 Pedagogical Issues

The outcome of this thesis suggests several pedagogical issues to consider when teaching OO programming. Identifying what types of knowledge novice programmers found difficult to comprehend in OO programs will contribute to the theory of OO program comprehension as well as helping educators to teach OO program comprehension skills. Wiedenbeck and Ramalingam (1999) emphasised the importance of understanding how OO programs affect novice programmers in comprehending different types of knowledge. In terms of pedagogy, the investigation points to the need for careful attention to knowledge of *Data Flow* and *Program Goals* in teaching OO program comprehension. Further empirical studies are needed in order to determine the reason for the comprehension difficulties with these two types of knowledge in OO programs. It also calls for a thorough understanding of the comprehension of the *critical aspects* of *Control Flow*, *State*, and *Problem Classes* knowledge.

Another pedagogy issue is the lesson learned from using different types of example programs in teaching OO programming approach introduced by

Börstler et al. (2010. 2011). The authors have identified properties of what they called "*quality factors*" OO example programs. These properties are: technically correct, readable, promote "object-oriented thinking". This investigation suggests that emphasising types of knowledge, investigated in this thesis, as well as the proposed *critical aspects* of these types of knowledge, should also be added to the property list. It seems not possible to incorporate all these types of knowledge in one small program example. Rather, different program examples would highlight and, thus, facilitate comprehension of different types of knowledge. For example, as in this investigation, the *Object* based program version of the Car study facilitated comprehension of *Problem Classes* knowledge. Therefore, a program example such as the one used in the Car study is a good example program to highlight and teach this type of knowledge. Similarly, the *Object* based program version of the Line-Edit study made *Control Flow*, *State* and *Problem Classes* knowledge easier to comprehend than the other types of knowledge. So perhaps we should use Line-Edit as an example program in teaching these types of knowledge. The findings of differences in comprehension among different types of example program in this investigation make further study of such a question interesting. What type of problems/program examples emphasis what type of knowledge is a good area of future empirical research? This research has, therefore, succeeded in producing the first categorisation of programming examples. We call it "*knowledge-based*" program example categorisation. This categorisation acts as an effective educational tool for OO educators to improve OO program comprehension skills.

Chapter 7 Conclusions and Further Work

The broad aim of the thesis was to investigate the ease of comprehension of OO programs versus the non OO program. Therefore, the original motivation for conducting empirical studies was simply to gather empirical data that would reinforce or refute the claim regarding the ease of comprehension of OO programs. This chapter reports the primary contributions and findings of the thesis and relates them to the main thesis questions put forward in Chapter 1. It then discusses the achievements and limitations along with suggestions for future work. A summary comes at the end.

7.1 Findings and Contributions

In endeavouring to answer the thesis questions about the ease of comprehension of OO programs over the non OO programs, the investigation's findings can be summarised as follows:

- the investigation provides empirical evidence that supports claims about the ease of comprehension of OO programs. Different types of knowledge contributed to the ease of comprehension of OO programs in each of the studies. Only *Problem Classes* knowledge contributed to the difference in comprehension between the *Object* based program and the *Non-Object* based program in the Car study; *Control Flow*, *State*, and *Problem Classes* also contributed to the difference in comprehension between the *Object* based program and Non-Object based program in the Line-Edit study;

- the *performance* and the *ranking* measures were found to be good indicators in measuring comprehension, whilst *time* was not;
- the *State* and *Problem Classes* knowledge were the most dominant positively affecting comprehension of the *Object* based program in the Car study. However, *Problem Classes* knowledge had a negative dominant effect on comprehension of the *Non-Object* based programs in both Car and Line-Edit studies;
- the proposed model of OO program comprehension considered *Control Flow*, *State*, and *Problem Classes* as *primary* types of knowledge, whilst it considered *Elementary Operations*, *Data Flow*, and *Program Goals* as *secondary* types of knowledge;
- there is a strong influence of *class concept*, *problem characteristics*, and *solution decompositions* on comprehension of *primary* types of knowledge in the *Object* based programs. On the other hand, no significant influence was found on comprehension of *secondary* types of knowledge;
- due to the equivalence between experimental programs, the investigation's scope was restricted to assessing the ease of comprehension of types of knowledge that have equivalent counterpart questions in both program versions *(Non-Object* based and *Object* based);
- the use of additional cues as well as the lack of criteria for comparability between *Non-Object* based and *Object* based programs seem to be unavoidable methodological issues that can affect the investigation's findings;

- pedagogically, there is a need to give careful attention to critical aspects of *primary* types of knowledge as well as *secondary* types of knowledge. Moreover, introducing *knowledge-based* example programs would help in improving OO program comprehension skills.

The following contributions have been made:

1. Assessing the ease of comprehension of OO programs by devising a specific experimental methodology, based on previous empirical experiments on program comprehension but tailored to the need of our experimental settings. This led to considering different methodological issues, such as choice of subjects, materials, and metrics.
2. Conducting two main sets of empirical studies (Car and Line-Edit), based on the tailored methodology, aiming to assess the ease of comprehension of OO programs and in particular *class concept, problem characteristics, and solution decompositions.* This led to proposing a new empirically grounded based model of OO program comprehension.
3. Highlighting a number of methodological issues that affected the investigation's findings. Introducing different *critical aspects* that are considered important to OO program comprehension.
4. Suggesting a number of pedagogical issues that can be considered as supporting education tools in teaching OO programming. Proposing a categorisation of example programs to improve OO program comprehension skills.

7.2 Suggestions and Further Work

The investigation carried out in this thesis has shed some light on the field of empirical works in OO program comprehension. The investigation has also pointed to other areas of future empirical research in this field. This section discusses suggestions for further research in this files based on the limitations of this investigation.

The present investigation is one of what should eventually be an ensemble of empirical studies of OO program comprehension. The investigation established a foundation for further work in this field. The investigation suggests that other researchers can build up on this work by using other problem types as case study materials to cover all Jackson problem frames.

Although the investigation found OO programs easier to comprehend than non OO programs, especially for *primary* types of knowledge, it calls for further empirical studies to find out why *secondary* types of knowledge were found difficult to comprehend. In this context the investigation was able to introduce a new *knowledge-based* categorisation of program examples for teaching OO programming. These findings are representative and can be generalised for other program examples that have similar characteristics. Moreover, these findings raise new questions which may be pursued. The investigation suggests a future direction of empirical studies to find out what type of program examples emphasise what type of knowledge. One possible research direction is to investigate the ease of comprehension of program examples each of which is assumed to emphasise a certain type(s) of knowledge. The main goal here is to expand the *knowledge based* categorisation of program examples which can be used in improving OO program comprehension skills.

The investigation also has limits to obtain empirical data about the comprehension of the *critical aspects* of the primary types of knowledge. This limitation was attributed to the nature of the comparison approach followed by the investigation. In order to investigate all these critical aspects, the core direction of further research should focus on investigating these *critical aspects* within the OO approach rather than between different programming approaches. This also requires adapting the experimental settings and, in turn, will lead to expanding the scope of the investigation. A possible research direction would be to assess the ease of comprehension of these critical aspects in a particular problem that possesses different solution decompositions both of which can be implemented in the form of *Object* based programs. The assessment could then be discussed in terms of contributions to critical aspects of the comprehension process. The original intention here is to develop a teaching tool based on comprehension of the critical aspects and then to apply this tool in teaching OO programming.

Although the investigation involved subjects who are considered at novice and experienced programmers' level, the example programs addressed here are still appropriate to be scaled up to professional software developers' level. This can emerge another research direction. The direction focus on investigating how including advanced OO concepts, such as, inheritance and polymorphism, can affect the comprehension of types of knowledge used in this investigation. This, in turn, will help to enrich the knowledge-based categorisation of the program examples for professionals and improve their program comprehension skills.

7.3 Summary

The broad aim of the thesis was to investigate whether OO programs are easy to comprehend. Thus, the original motivation for this investigation was to look at the issue of OO program comprehension. The investigation focused on the influence of problem characteristics and solution decompositions on OO program comprehension. The idea of different types of knowledge was looked at in depth, both as a methodological tool for experimentation and as a basis for supporting OO program comprehension.

Although the thesis did not aim to investigate the stages of how different types of knowledge were comprehended, it was able to classify types of knowledge used based on their importance to OO comprehension (primary and secondary). The thesis also sought to uncover critical aspects of the primary types of knowledge which are considered important to OO program comprehension. It appears that elements of class concept, problem characteristics, and solution decomposition are influential in OO program comprehension. The thesis suggested different research directions to improve OO program comprehension skills by discovering the ease of comprehension of critical aspects of primary types of knowledge. It also proposed another research direction to expand the categorisation of knowledge based program examples by investigating the ease of comprehension of secondary types of knowledge

To conclude, this thesis focused on topics of interest in the domain of OO program comprehension. In doing so, it has considered the influence of different elements and methodological issues, and highlighted critical aspects relating to the way in which OO program comprehension might best be studied. It has described how the findings of the investigation might be used to provide useful support in the field of empirical work in OO program comprehension. As such, it has established a foundation for further work in this field.

References

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of experimental psychology: Learning, memory, and cognition,* **10** (3), 483.

Affandy Herman, N. S., Salam, S. B., & Noersasongko, E. (2011). A Study of Tracing and Writing Performance of Novice Students in Introductory Programming. In *Software Engineering and Computer Systems,* Springer Berlin Heidelberg, 557-570.

Alardawi, A., Khazaei, B., & Siddiqi, J. (2011a). Influence on Novices of Class Structure on Program Comprehension. In *Proceedings of 7th Work-in-Progress Workshop of the Psychology of Programming Interest Group*, PPIG, 1-3.

Alardawi, A., Khazaei, B., & Siddiqi, J. (2011b). Influence of Class Structure on Program Comprehension. *In Proceedings of the 23th Annual Workshop of the Psychology of Programming Interest Group,* PPIG, 10-23.

Ambler, A. L., Burnett, M. M., & Zimmerman, B. A. (1992). Operational versus definitional: A perspective on programming paradigms. *Computer*, **25**(9), 28-43.

Anderson, N. H. (2001). Empirical direction in design and analysis: scientific psychology series. *Mahwah, New Jersey: LEA.*

Atwood, M. R., & Jeffries, R. (1980). *Studies in Plan Construction I: Analysis of an Extended Protocol* (No. SAI-80-028-DEN). Science applications inc Englewood Co..

Bailey, R. A. (2008). *Design of comparative experiments.* Cambridge University Press, 25.

Baker, F. T. (2003). Chief programmer team. John Wiley and Sons Ltd , 209-210.

Basili, V. R. (1993). *The experimental paradigm in software engineering.* Springer Berlin Heidelberg. 1-12.

Basili, V. R. (1992). *Software Modelling and Measurement: The Goal/Question/Metric Paradigm. Technical Report. University of Maryland at College Park, College Park, MD, USA.*

Basili, V. R., & Reiter Jr, R. W. (1981). A controlled experiment quantitatively comparing software development approaches. *Software Engineering, IEEE Transactions on*, (3), 299-320.

Basili, V. R., & Selby, R. W. (1987). Comparing the effectiveness of software testing strategies. *Software Engineering, IEEE Transactions on*, (12), 1278-1296.

Bierre, K., Ventura, P., Phelps, A., & Egert, C. (2006). Motivating OOP by blowing things up: an exercise in cooperation and competition in an introductory java programming course. In *ACM SIGCSE Bulletin*, ACM, **38** (1), 354-358.

Biggerstaff, T. J., Mitbander, B. G., & Webster, D. (1993). The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering,* IEEE Computer Society Press, 482-498.

Black, T. R. (1999). Doing quantitative research in the social sciences: An integrated approach to research design, measurement and statistics. Sage Publications Limited.

Borgida, A., Greenspan, S., & Mylopoulos, J. (1985). Knowledge representation as the basis for requirements specifications, Springer Berlin Heidelberg. 152-169.

Börstler, J., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C., & Thomas, L. (2010). An evaluation of object oriented example programs in introductory programming textbooks. ACM SIGCSE Bulletin, **41**(4), 126-143.

Börstler, J., Nordstrom, M., & Paterson, J. H. (2011). On the Quality of Examples in Introductory Java Textbooks. *ACM Transactions on Computing Education*, 11- 21.

Briand, L., Arisholm, E., Counsell, S., Houdek, F., & Thévenod–Fosse, P. (1999). Empirical studies of object-oriented artifacts, methods, and processes:

state of the art and future directions. *Empirical Software Engineering*, **4**(4), 387-404.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, **9**(6), 737-751.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International journal of Man-Machine studies*, **18**(6), 543-554.

Brooks, R. E. (1980). Studying programmer behaviour experimentally: the problems of proper methodology. *Communications of the ACM*, **23**(4), 207-213.

Bryman, A. (2012). *Social research methods*. OUP Oxford.

Burkhardt, J. M., Détienne, F., & Wiedenbeck, S. (2006a). Mental representations constructed by experts and novices in object-oriented program comprehension. *arXiv preprint cs/0612018*.

Burkhardt, J. M., Détienne, F., & Wiedenbeck, S. (2006b). Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *arXiv preprint cs/0612004*.

Calhoun, C. (1995). Critical social theory: culture, history, and the challenge of difference. Wiley-Blackwell.

Capps, J. (2012). *Pragmatism: An introduction.* New York, Continuum Intl Pub Group.

Carver, J., Jaccheri, L., Morasca, S., & Shull, F. (2003). Issues in using students in empirical studies in software engineering education. In *Software Metrics Symposium, 2003. Proceedings. Ninth International,* IEEE, 239-249.

Chatel, S., & Détienne, F. (2007). Strategies in object-oriented design. *arXiv preprint cs/0703008.*

Cheatham, T. J., & Mellinger, L. (1990). Testing object-oriented software systems. In *Proceedings of the 1990 ACM annual conference on Cooperation,* ACM, 161-165.

Chris, D. (2004). *Calculating a nonparametric estimate and confidence interval using SAS software.* Glaxo Wellcome Inc., Research Triangle Park, NC.

Christensen, L. B. (2007). *Experimental methodology. Pearson/*Allyn & Bacon.

Clark-Carter, D. (2009). Quantitative psychological research: The complete student's companion. Psychology press.

Cohen, L., Manion, L., Morrison. K., (2013). Research methods in education. Routledge.

Cook, T. D., Campbell, D. T., & Day, A. (1979). *Quasi-experimentation: Design & analysis issues for field settings* (p. 405). Boston: Houghton Mifflin.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. In *ACM SIGCSE Bulletin,* AMC, **35** (1), 191-195.

Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, **28**(2), 294-306.

Corritore, C. L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension?. *International Journal of Human-Computer Interaction*, **3**(2), 199-222.

Corritore, C. L., & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, **50**(1), 61-83.

Creswell, J. W. (2011). *Educational research: Planning, conducting, and evaluating quantitative and qualitative research,.* 4th edition. Pearson Education.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., & Love, T. (1979). Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *Software Engineering, IEEE Transactions on*, (2), 96-104.

Daly, J. (1996). Replication and a multi-method approach to empirical software engineering research, PhD thesis, University Of Strathclyde.

De Sá, J. M. (2007). Applied Statistics: Using SPSS, Statistica, MATLAB, and R. Springer.

De Vaus, D. (2013). *Surveys in social research*. Routledge.

Détienne, F. (2006a). Assessing the cognitive consequences of the object-oriented approach: a survey of empirical research on object-oriented design by individuals and teams. *arXiv preprint cs/0611154*.

Détienne, F. (2006b). Design Strategies and Knowledge in Object-Oriented Programming: Effects of Experience. *arXiv preprint cs/0612008*.

Détienne, F., & Bott, F. (2002). *Software design--cognitive aspects*. Springer Verlag.

Dijkstra, E. W., Dijkstra, E. W., Dijkstra, E. W., & Dijkstra, E. W. (1976). *A discipline of programming*. Englewood Cliffs: prentice-hall, 1.

Dunsmore, A., & Roper, M. (2000). A comparative evaluation of program comprehension measures. *The Journal of Systems and Software*, **52**(3), 121-129.

Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering,* Springer London. 285-311.

Fenton, N. E., & Bieman, J. (2013). *Software metrics: a rigorous and practical approach*. Taylor & Francis Group.

Gannon, J. D., & Horning, J. J. (1975). The impact of language design on the production of reliable software. In *ACM SIGPLAN Notices*, ACM, **10** (6), 10-22.

Gray, D. E. (2009). Doing research in the real world. Sage.

Gilmore, D. J., & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, **21**(1), 31-48.

Good, J. (1999). *Programming paradigms, information types and graphical representation: Empirical investigation of novice program comprehension,* PhD thesis, The University of Edinburgh.

Green, T. R. G., Sime, M. E., & Fitter, M. (1975). *Behavioural experiments on programming languages: Some methodological considerations.* University of Sheffield. Dept. of Psychology. Medical Research Council Social and Applied Psychology Unit,

Green, T. R. G. (1980). Programming as a cognitive activity. *Human interaction with computers*, 271-320.

Grix. J., (2010). *The foundation of research.* 2nd Edition, Palgrave Macmilian.

Harrison, R., Counsell, S., & Nithi, R. (2000). Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, **52**(2), 173-179.

Hedrick, T. E., Bickman, L., & Rog, D. J. (1993). *Applied research design: A practical guide*. Newbury Park, CA: Sage.

Herbsleb, J. D., Klein, H., Olson, G. M., Brunner, H., Olson, J. S., & Harding, J. (1995). Object-oriented analysis and design in software project teams. *Human–Computer Interaction*, **10**(2-3), 249-292.

Hinton, P., Brownlow, C., & McMurray, I. (2004). *SPSS explained.* Routledge.

Hoc, J. M. (1981). Planning and direction of problem solving in structured programming: An empirical comparison between two methods. *International Journal of Man-Machine Studies, 15*(4), 363-383.

Jackson, M. (1995). *Software requirements & specifications.* New York: ACM Press, 8.

Jackson, M. (2005). Problem frames and software engineering. *Information and Software Technology, 47*(14), 903-912.

Johnson, R. E., & Foote, B. (1988). Designing reusable classes. *Journal of object-oriented programming, 1*(2), 22-35.

Johnson-Laird, P. N. (1986). Mental models: Towards a cognitive science of language, inference, and consciousness (No. 6). Harvard University Press.

Jonker, J., & Pennink, B. J. W. (2010). The Essence of Research Methodology: A Concise Guide for Master and PhD Students in Management Science. Springer.

Kant, E., & Newell, A. (1984). Problem solving techniques for the design of algorithms. *Information Processing & Management*, **20**(1), 97-118.

Khazaei, B., & Jackson, M. (2002). Is there any difference in novice comprehension of a small program written in the event-driven and object-oriented styles?. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia,* IEEE, 19-26.

Khazaei. B. (1990). *The determinations of program designer behaviour: An empirical study.* PhD thesis. University of Wolverhampton.

Kintsch, W., & Van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological review*, **85**(5), 363-394.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, **28**(8), 721-734.

Klein, H. K., & Myers, M. D. (1999). A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS quarterly*, 67-93.

Kölling, M., & Henriksen, P. (2005). Game programming in introductory courses with direct state manipulation. *ACM SIGCSE Bulletin*, **37**(3), 59-63.

Pfleeger, S. L. (2010). *Software Engineering: Theory and Practice,* Prentice-Hall. *Inc.,.*

Leach, C. (1979). Introduction to statistics: A nonparametric approach for the social sciences. New York: Wiley.

Liskov, B., Snyder, A., Atkinson, R., & Schaffert, C. (1977). Abstraction mechanisms in CLU. *Communications of the ACM*, **20**(8), 564-576.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., ... & Whalley, J. L. (2006, June). Research perspectives on the objects-early debate. In *ACM SIGCSE Bulletin.* ACM. **38** (4), 146-165.

Littman, D. C., Pinto, J., Letovsky, S., & Soloway, E. (1987). Mental models and software maintenance. *Journal of Systems and Software*, **7**(4), 341-355.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, **13**(3), 307-325.

Mendonça, M. G., Maldonado, J. C., de Oliveira, M. C., Carver, J., Fabbri, C. P. F., Shull, F., ... & Basili, V. R. (2008). A framework for software engineering experimental replications. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference. IEEE,* 203-212.

Meyer, B. (1988). *Object-oriented software construction.* 331-410. New York: Prentice hall, 2.

Meyer, B. (1997). *Object-oriented software construction.* Prentice hall New York., 2.

Miller, L. A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies*, **6**(2), 237-260.

Miller, S. (2006). *Experimental design and statistics.* Routledge, 1.

Moher, T., & Schneider, G. M. (1982). Methodology and experimental research in software engineering. *International Journal of Man-Machine Studies*, **16**(1), 65-87.

Naur, P. (1969). Programming by action clusters. *BIT numerical mathematics,* **9** (3), 250-258.

Neubauer, B. J., & Strong, D. D. (2002). The object-oriented paradigm: more natural or less familiar?. *Journal of Computing Sciences in Colleges*, **18**(1), 280-289.

O'Brien, M. P. (2003). Software comprehension–a review & research direction. Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report.

Pallant, J. (2010). SPSS survival manual: A step by step guide to data analysis using SPSS. Open University Press.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, **15**(12), 1053-1058.

Peirce, C. S., & Menand, L. (1997). Pragmatism: A Reader. New York, Vintage

Pennington, N. (1987a). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, **19**(3), 295-341.

Pennington, N. (1987b). Comprehension strategies in programming. In *Empirical studies of programmers: second workshop.* Ablex Publishing Corp, 100-113.

Pennington, N., & Grabowski, B. (1990). The tasks of programming. *Hoc et al, 307*, 45-62.

Peter, M. (1990). Expert programmers and programming languages. *Psychology of programming,* 103-115.

Porter, A. A., Votta Jr, L. G., & Basili, V. R. (1995). Comparing detection methods for software requirements inspections: A replicated experiment. *Software Engineering, IEEE Transactions on*, **21**(6), 563-575.

Quantrani, T. (2003). *Visual modeling with rational rose 2002 and uml.* Addison-Wesley Professional.

Ramalingam, V., & Wiedenbeck, S. (1997, October). An empirical study of novice program comprehension in the imperative and object-oriented styles. In

Papers presented at the seventh workshop on Empirical studies of programmers. ACM, 24-139.

Reisner, P. (1977). Use of psychological experimentation as an aid to development of a query language. *Software Engineering, IEEE Transactions on*, (3), 218-229.

Rist, R. (1996a). System structure and design. *In Proceedings of the Workshop on Empirical Studies of Programmers.* 163-194.

Rist, R. S. (1996b). Teaching Eiffel as a first language. *Journal of object-oriented programming*, **9** (1), 30-41.

Robson, C. (2002). *Real world research: A resource for social scientists and practitioner-researchers* . Oxford: Blackwell, 2.

Rosson, M. B., & Alpert, S. R. (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction*, **5**(4), 345-379.

Sajaniemi, J., & Kuittinen, M. (2007). From procedures to objects: What have we (not) done. In Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group, University of Joensuu, Department of Computer Science and Statistics, Citeseer, 86-100.

Sajaniemi, J., & Kuittinen, M. (2008). From procedures to objects: A research agenda for the psychology of object-oriented programming education. *Human technology,* **4** (1), 75-91.

Saunders, M. A., Liang, H., & Li, W. H. (2007). Human polymorphism at microRNAs and microRNA target sites. In *Proceedings of the National Academy of Sciences*, *104*(9), 3300-3305.

Saunders, M. N., Saunders, M., Lewis, P., & Thornhill, A. (2011). *Research Methods For Business Students, 5/e*. Pearson Education India.

Sawyer, A. G., & Ball, A. D. (1981). Statistical power and effect size in marketing research. *Journal of Marketing Research*, 275-290.

Schmalhofer, F., & Glavanov, D. (1986). Three components of understanding a programmer's manual: Verbatim, propositional, and situational representations. *Journal of Memory and Language*, **25**(3), 279-294.

Sheil, B. A. (1981). The psychological study of programming. *ACM Computing Surveys (CSUR)*, **13**(1), 101-120.

Shneiderman, B. (1975). Experimental testing in programming languages, stylistic considerations and design techniques. In *Proceedings of the May 19-22, 1975, national computer conference and exposition.* ACM, 653-656.

Shneiderman, B. (1977). Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, **9**(4), 465-478.

Shneiderman, B. (1980). Software psychology: Human factors in computer and information systems (Winthrop computer systems series). Winthrop Publishers.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behaviour: A model and experimental results. *International Journal of Computer & Information Sciences*, **8**(3), 219-238.

Shull, F., Singer, J., & Sjberg, D. I. (2008). *Guide to advanced empirical software engineering*. Springer.

Siddiqi, J. (1984). *An empirical investigation into problem decomposition strategies used in program design.* PhD thesis. The University of Aston.

Sime, M. E., Green, T. R. G., & Guest, D. J. (1999). Psychological evaluation of two conditional constructions used in computer languages. *International journal of human-computer studies,* **51**(2), 125-133.

Simon, H. A., & Newell, A. (1971). Human problem solving: The state of the theory in 1970. *American Psychologist*, **26**(2), 145-159.

Sjøberg, D. I., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanović, A., & Vokáč, M. (2003). Challenges and recommendations when increasing the realism of controlled software engineering experiments. In *Empirical methods and studies in software engineering.* Springer Berlin Heidelberg, 24-38.

Sjoberg, D. I., Anda, B., Arisholm, E., Dyba, T., Jorgensen, M., Karahasanovic, A. & Vokác, M. (2002). Conducting realistic experiments in software engineering. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*. IEEE, 17-26.

Sjøberg, D. I., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N. K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *Software Engineering, IEEE Transactions on*, **31**(9), 733-753.

Soloway, E., & Ehrlich, K. (1989). Empirical studies of programming knowledge. In: *Software Reusability,* ACM, 235-267.

Soloway, E., Ehrlich, K., & Bonar, J. (1982, March). Tapping into tacit programming knowledge. In *Proceedings of the 1982 conference on Human factors in computing systems.*ACM. 52-57.

Spinellis, D. (2003). *Code reading: the open source perspective.* Addison-Wesley Professional.

Sprent, P., & Smeeton, N. C. (2007). *Applied nonparametric statistical methods.* Chapman & Hall.

Stark, S., Torrance, H. (2005). Case Study, w: B. Somekh, C. Lewin (red.), Research Methods in the Social Sciences, Londyn, New Delhi: Thousands Oaks.

Stolin, Y., & Hazzan, O. (2007). Students' understanding of computer science soft ideas: the case of programming paradigm. *ACM SIGCSE Bulletin*, **39**(2), 65-69.

Storey, M. A. (2006). Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, **14(**3), 187-208.

Thomas, R. M. (2003). Blending qualitative and quantitative research methods in theses and dissertations. Corwin-volume discounts.

Thompson, E. (2008). *How do they understand? Practitioner perceptions of an object-oriented program.* PhD thesis, Massey University.

Van Dijk, T. A., Kintsch, W., & Van Dijk, T. A. (1983). *Strategies of discourse comprehension*. New York: Academic Press.

Von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, **28**(8), 44-55.

Walliman, N. (2005). Your research project: a step-by-step guide for the first-time researcher. Sage Publications Limited.

Weinberg, G. M. (1971). *The psychology of computer programming*. New York: Van Nostrand Reinhold, 932633420.

Weinberg, G. M. (1992). *Quality software management: systems thinking*. Dorset House Publishing Co., Inc, 1.

Weinberg, G. M. (2005). The Psychology of Computer Programming. *Phi Delta Kappan*.

Weissman, L. (1974). Psychological complexity of computer programs: an experimental methodology. *ACM Sigplan Notices*, **9**(6), 25-36.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*, **51**(1), 71-87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, **11**(3), 255-282.

William, W. (2009). *Research methods in education*. Pearson Education India.

Wing, J. M. (1988). A study of 12 specifications of the library problem. *Software, IEEE*, **5**(4), 66-76.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer.

Yeh, R. T. (1979). In Memory of Maurice H. Halstead. *IEEE Trans. Software Eng.*, **5**(2), 74.

Yin, R. K. (2009). Case study research: Design and methods (Vol. 5). Sage.

Zaidman, A. (2006, March). Scalability solutions for program comprehension through dynamic analysis. In *Software Maintenance and Reengineering, 2006. CSMR 2006*. Proceedings of the 10th European Conference on, IEEE, 4.

Zhu, H., & Zhou, M. (2003). Methodology first and language second: a way to teach object-oriented programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* ACM, 140-147.

Appendixes

Appendix A: Materials: Car Study

This appendix shows the experimental packet given to subjects in the Car experiment (described in Chapter 5), and includes:

1. The Car problem specification[1];
2. The experimental purpose and procedure[2]
3. The programming background questionnaire.
4. The experimental treatments (problem solutions) and the corresponding comprehension questions. The ranking question at the end

[1] The problem specification was not given to the subjects.
[2] As the experiment took place within a lab session for each group, the purposes and procedure for the experiment were verbally embedded within a description of that day's lab session.

The Car Problem's Specification

There is no definitive statement for the car problem's specification[3]. The following is the best description that I can provide.

Design a simulation program that maintains the speed of a car. This car serves as a generic description of any existing car; car could have, for example, brand name, speed, number of passengers, and engine. The program should first describe the car specifications (brand and engine), by outputting description messages to the user. Then, based on whether there are passengers on the car or not, the program maintains the car's speed. This also can be done by outputting different messages back to the user based on the value of the passengers and speed as following:

- if there are no passengers on the car, this means the car speed equal 0 mph and the car is stopping. Otherwise, move to the next two options:
  - .1 The speed is less than or equal to 50 mph. output message that the car is travelling within the speed limit.
  - .2 The speed is more than 50 mph. output message that the car is travelling over speed limit.

Experimental purposes and procedures

During this lab session, you will be asked to take part in a short experiment. This experiment is a part of a research aims to investigate program comprehension for Object Oriented concepts. Obviously, we could simply ask you this question directly, but as you can probably guess, doing so wouldn't necessarily give us the types of answers which might be useful: for example,

[3] The references had discribed different attributes of the car program that deals with passengers and speed (Ramalingam and Susan Wiedenbeck 1997; Wiedenbeck et al 1999; Wiedenbeck and Ramalingam 1999; Khazaei and Jackson 2002)

the answers might not be quantifiable in a consistent way and hence will be not suitable for any sort of statistical analysis. We hope to collect data which can be analysed, and which either support work which has already done or offer new and possibly conflicted evidence. If you are interested obtaining more information on this research, please let me know.

Before starting the experiment, you should be aware that:

- This is no way a test of your programming knowledge level will not be used in any form as part of your assessment on the all courses.
- You are not expected to get all of the answers right. What we are interested in here are the types of questions which people get wrong compared to those they get right.
- The data collected will be both anonymous and strictly confidential and will be only used in the purposes of this research.

Now, the experiment has three stages. At the first stage, you will be given a programming background questionnaire that contains two sections. The first section asking about personal details, please note that giving these details is optional. The second section asking you about you programming experience, please fill this in, and then give it to your experimenter.
In the second stage, you will be given a booklet of two pages to work through. The booklet contains a computer program and a number of questions you are asked to answer. Please do not start work with the program unless the experimenter tells you to do so. Once you told to start please spend some time reading the program before you go to the questions. If you finish or you decide to stop or withdraw from the experiment please tel the experimenter immediately.

The last stage you will have a question asking you to rank the program you had just gone through please answer this and give it back to the experimenter.

Note that it will not be possible for the experimenter to give you details of the experiment through the whole experimental period, as knowing the hypotheses may influence how you respond[4]. If you have any questions about these instructions, please ask the experimenter now.

Thank toy for taking part of this experiment.

[4] For the same reason, please do not discuss you ideas on the experimental program you given with people who have not taken part in this week's lab session.

Programming Background Questionnaire

Thank you for taking the time to complete the following questions, through the following questions you will be asked about your programming experience, please fill this in. The information you provide will be used in a study to investigate program comprehension for Object Oriented concepts and will be treated confidentially.

Personal Details:

Name (optional) ______________________ Age: (Optional) __________

Course/Module: ______________________ Gender: Male / Female

Programming Experience:

For the following programming languages, please indicate:

1. How you learnt the language (School, University, Work, Self taught).
2. Rate yourself according to your level of knowledge (1- Novice, 5-Expert).
3. Add any Programming language which is missed.

	School	University	Work	Self Taught	Rate of Knowledge (1 to 5)
Basic					
Fortran					
Logo					
Pascal					
C					
Visual Basic					
C++					
Java					
C#					

The Experimental Treatments

7.3.1.1 A1. Visual Basic Non-Object based Experimental Treatment

The Car Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Car program". It deals with body, engine, speed, and passengers.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:		VB *Non-Object* based			Subject code				
Date		Place:							
Starting Time			Ending Time						
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Program Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
Public Class Car_Program
    Private Sub Set_Car_Click(... ) Handles Set_Car.Click
        Dim Power As Integer
        Dim Type As String
        ' Assigne the Power value of the engine
        Power = Val(TextPower.Text)
        ' Assigne the type value of the body
        Type = TextType.Text
        ' Discribe Car's specification
        HessageBox. Show ("You have created car"sType£"Itsengme power="£Power)
        Car_status.Enabled = True
    End Sub
   Private Sub Car_status_Click(...)Handles Car_status.Click
        Dim Passengers, Speed As Integer
        ' Assigne the No.of.Passengers valus
        Passengers = Val(TextPassengers.Text)
        If Passengers = 0 Then
            MessageBox.Show("Car is Stopping")
        Else
            1 Assigne the Speed value of the car
            Speed = Val(TextSpeed.Text)
            If Speed > 50 Then
                MessageBox.Show("Over Speed")
            Else
                MessageBox.Show("Within Normal Speed")
            End If
        End If
    End Sub
End Class
```

<u>Please answer the following questions:</u>

- Does the user assign a value to variable "Brand"? (Yes/No/Do not Know)
- Is the variable "Passengers" initialized to zero? (Yes/No/Do not Know)
- Does the user assign a value to variable "Speed"? (Yes/No/Do not Know)
- In "Car_Status" method, does "Speed" value assigned in the case of "Passengers" =zero? (Yes/No/Do not Know)
- Is the value of variable "Speed" assigned before the value of variable "Passengers"? (Yes/No/Do not Know)
- In "Set_Car" method, does the program instantiate engine before body? (Yes/No/Do not Know)
- Does the value of "Passengers" affect the value of "Speed"? (Yes/No/Do not Know)
- Does the value of "Brand" affect the value of "Power"? (Yes/No/Do not Know)
- Does the program allow you to create new car with a certain brand and power? (Yes/No/Do not Know)
- Does the program allow you to change the car specifications (Brand / Power)? (Yes/No/Do not Know)
- Does the program compare number of passengers in two cars? (Yes/No/ Do not Know)
- When the "Car is Stopping" statement is reached, is the value of "Passengers" > zero?(Yes/No/Do not Know)
- When the "Over Speed" statement is reached, is the value of "Speed" = 50? (Yes/No/Do not Know).

Now, if you were asked to develop the same program based on the concept of chunking every relevant code together into number of entities, this concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and functions/methods. Fro the following questions please state which of the following entities would be useful?

- Entity called "***Body***", putting together all relevant attributes and functions, used to set and describe the body's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Wheels***", putting together all relevant attributes and functions, used to set and describe the wheels' specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "***Engine***", putting together all relevant attributes and functions, used to set and describe the engine's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Passengers***", putting together all relevant attributes and functions, used to set and describe the passengers' information. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Car***", putting together all relevant attributes and functions, used to set and describe the car's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "***Lorry***", putting together all relevant attributes and functions, used to set and describe the lorry's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.

Ranking Question:

- How well do you understand the code?
  - o not very well
  - o fairly to moderately well
  - o well to very well.

Thank you for your cooperation

7.3.1.2 A2. Visual Basic Object based Experimental Treatment

The Car Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Car program". It deals with body, engine, speed, and passengers.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:		VB *Object* based			Subject code				
Date		Place:							
Starting Time			Ending Time						
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Progra m Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
Class Engine ' Beginning of Engine class
    'Declare Engine class Attributes
    Private Power As Integer
    ' Class Hethods and behaviour
    Public Sub Set_Engine()
    Console.WnteLine("Enter the engine's power")
    ' Assigne the Power value of the engine
    Power = Val (Console. ReadLineO)
    End Sib
    Public Sub Engine_Describe{)
        Console. WnteLine ("Engine power is ="4Power)
    End Sub
End Class 1 End of class Engine
```

```
Class Body ' Beginning of Body class
    ' Declare Body class Attributes
    Private Brand As String
    ' Class Hethods and behaviour
    Public Sub Set_Body{)
    Console. Writeline ("Enter the Body's Brand")
    ' Assigne the Brand value of the engine
    Brand = Console. ReadLineO
    End Sub
    Public Sub BodyJDescribeO
      Console.Writeline("Car Brand is:  " 4 Brand)
    End Sub
End Class ' End of class Body
```

```
Class Car ' Beginning of Car class
    Private Passengers, Speed As Integer 'Declare Car class Attributes
    Private CEngine As New Engine ' Creates new instant of class Engine
    Private CBody As New Body ' Creates new instant of class Body
    'Class Methods and behaviour
    Public Sub Set_Car()
        CEngine.Set_Engine() 'Instantiate Engine object
        CBody. Set_Body() 'Instantiate Body object
    End Sib
    Public Sub Car_Descnbe()
        CEngine. Engme_Describe()
        CBody.Body Descnbe()
    End Sib
    Public Sub Car_Status()
        Console.WriteLine("Enter the No.of.Passengers")
        Passengers = Val (Console. ReadLme 0 )
        If Passengers - 0 Then
            Console.WriteLine("Car is Stopping")
        Else
            Console.WnteLine("Enter the Car Speed")
            Speed = Val (Console. ReadLineO)
            If Speed > 50 Then
                Console. Writelme ("Over Speed")
            Else
                Console.Writeline ("Within Normal Speed")
            End If
        End If
    End Sib
End Class
```

```
'the main program start here
Hodule Hodulel
    Sub HainO
        Dim CCarl As New Car 'Create new instance
        CCarl.Set_Car ()
        CCarl.Car_Describe()
        CCarl.Car_Status()
        Dim CCar2 As New Car 'Create new instance
        CCar2.Set_Car()
        CCar2.Car_Describe()
        Console. ReadLineO
    End Sib
End Hodule
```

Please answer the following questions:

- Does the user assign a value to variable **"Brand"**? (Yes/No/Do not Know)
- Is the variable **"Passengers"** initialized to zero? (Yes/No/Do not Know)
- Does the user assign a value to variable "**Speed"**? (Yes/No/Do not Know)
- In **"Car_Status"** method in class **"Car"**, does **"Speed"** value assigned in the case of **"Passengers"** =zero? (Yes/No/Do not Know)
- Is the value of variable "**Speed**" assigned before the value of variable "**Passengers"**? (Yes/No/Do not Know)
- In **"Set_Car"** method in class **"Car"**, does the program instantiate engine before body? (Yes/No/Do not Know)
- Does the value of "**Passengers"** affect the value of "**Speed"**? (Yes/No/Do not Know)
- Does the value of "**Brand"** in class **"Body"** affect the value of **"Power"** in class **"Engine"**? (Yes/No/Do not Know)
- Does the program allow you to create new car with a certain brand and power? (Yes/No/Do not Know)
- Does the program allow you to change the car specifications (**Brand / Power**)? (Yes/No/Do not Know)
- Does the program compare number of passengers in two cars? (Yes/No/Do not Know)
- When the **"Car is Stopping"** statement is reached, is the value of "**Passengers"** > zero?(Yes/No/Do not Know)
- When the **"Over Speed"** statement is reached, is the value of "**Speed"** = 50? (Yes/No/Do not Know)

The listed program has been developed based on the concept of chunking every relevant code together into number of entities. This concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and function/methods. Fro the following questions please state which of the following entities were used?

- Entity called "**Body**", putting together all relevant attributes and functions, used to set and describe the body's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "**Wheels**", putting together all relevant attributes and functions, used to set and describe the wheels' specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "**Engine**", putting together all relevant attributes and functions, used to set and describe the engine's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "**Passengers**", putting together all relevant attributes and functions, used to set and describe the passengers' information. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "**Car**", putting together all relevant attributes and functions, used to set and describe the car's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "**Lorry**", putting together all relevant attributes and functions, used to set and describe the lorry's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.

<u>Ranking Question:</u>

- How well do you understand the code?
    - not very well
    - fairly to moderately well
    - well to very well.

Thank you for your cooperation

7.3.1.3 A3. JAVA Non-Object based Experimental Treatment

The Car Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Car program". It deals with body, engine, speed, and passengers.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:	JAVA *Non-Object* based				Subject code				
Date		Place:							
Starting Time			Ending Time						
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Progra m Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
public class Hodule {

    private static boolean isStatusEnabled = false;

    private static void set_Car() ...{
        System.out.println("Enter the engine's power");
        // Assign the power value of the engine
        int Power = Integer.parselnt(in.readlinef));
        System.out.pnntin( "Enter the Body's Type");
        //Assign the brand value of the body
        String Brand = in.readline();
        // Discnbe Car's specification
        System out.println("You have created car "+Brand+"Its engine power +Power);
        isStatusEnabled = true; }

    private static void statusf) .... {
        System.out.pnntln("Enter the Ho. of Passengers");
        // Assign the No.of Passenger value
        int passengers = Integer parselnt(m.readLinef));
        if (passengers == 0) {
            System.out.println("Car is Stopping"); }
        else {
            System.out.pnntln("Enter the Car Speed");
            // Assign the speed value of the car
            int speed = Integer parselnt(in readLmef));
            if (speed > SO) {
                System.out.pnntlnfOver Speed"); }
            else {
                System.out.pnntln( "Uithm Normal Speed");
            }
        }
    }
    public static void mam .... {
        System.out.pnntln("Vhat would you like to do?");
        System.out.println("l) Set Car");
        if (isStatusEnabled) {
            System.out.pnntln("2) View Car Status"); }
        int option = Integer.parselnt(m.readline0);
        if (option == 1) {
            setCarf);}
        else if (option «  2 SA isStatusEnabled) {
            status)); }
        else {
            System.out.pnntln("Invalid Option"); }
        main(args); // loop back
```

Please answer the following questions:

- Does the user assign a value to variable **"Brand"**? (Yes/No/Do not Know)
- Is the variable **"Passengers"** initialized to zero? (Yes/No/Do not Know)
- Does the user assign a value to variable "**Speed**"? (Yes/No/Do not Know)
- In **"Car_Status"** method, does **"Speed"** value assigned in the case of **"Passengers"** =zero? (Yes/No/Do not Know)
- Is the value of variable "**Speed**" assigned before the value of variable "**Passengers**"? (Yes/No/Do not Know)
- In **"Set_Car"** method, does the program instantiate engine before body? (Yes/No/Do not Know)
- Does the value of "**Passengers**" affect the value of "S**peed**"? (Yes/No/Do not Know)
- Does the value of "**Brand**" affect the value of **"Power"**? (Yes/No/Do not Know)
- Does the program allow you to create new car with a certain brand and power? (Yes/No/Do not Know)
- Does the program allow you to change the car specifications (**Brand / Power**)? (Yes/No/Do not Know)
- Does the program compare number of passengers in two cars? (Yes/No/ Do not Know)
- When the **"Car is Stopping"** statement is reached, is the value of "**Passengers**" > zero?(Yes/No/Do not Know)
- When the **"Over Speed"** statement is reached, is the value of "**Speed**" = 50? (Yes/No/Do not Know)

Now, if you were asked to develop the same program based on the concept of chunking every relevant code together into number of entities, this concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and functions/methods. Fro the following questions please state which of the following entities would be useful?

- Entity called "***Body***", putting together all relevant attributes and functions, used to set and describe the body's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Wheels***", putting together all relevant attributes and functions, used to set and describe the wheels' specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "***Engine***", putting together all relevant attributes and functions, used to set and describe the engine's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Passengers***", putting together all relevant attributes and functions, used to set and describe the passengers' information. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Car***", putting together all relevant attributes and functions, used to set and describe the car's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "***Lorry***", putting together all relevant attributes and functions, used to set and describe the lorry's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.

Ranking Question:

- How well do you understand the code?
    - not very well
    - fairly to moderately well
    - well to very well.

Thank you for your cooperation

7.3.1.4 A4. Java Object based Experimental Material

The Car Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Car program". It deals with body, engine, speed, and passengers.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:	JAVA *Object* based			Subject code					
Date		Place:							
Starting Time			Ending Time						
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Progra m Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
ptblic class Engine (//Beginning of Engine class
        //Declare Bngine class Attributes
        private int power;
        //Class's Methods and behaviour
        piblie void Set Engine!)
        {
            // Assign the Power value of the engine
            Syst«».out.println("Ent«x th« engine's power");
            power = Integer.parseInt!in.readLine());
        i
        )
        public void Engine Describe!)
        {
        System, out. print In ("Bngine power is = " +
                            String. valueOf (power));
        )
) //End of class Engine
```

```
public class Body ( //Beginning of Body class
        //Declare Body class Attributes
        private String Brand;
        //Class's Methods and behaviour
        public void Set_Body()
        1
            //Assign the type value of the Body
            System, out. print In ("Enter the Body's Type");
            type = in.readLine!);
        }
        pxfclic void BodyJDescri.be 0
        1                                                              |
        System, out. print In ("Car Type is:  " + type);
        )
} //End of class Body
```

```
public class Car ' Beginning of Car class
(
        private int passengers, speed; //Declare Car class Attributes
        private Engine engine; //Declare new instant of class Engine
        private Body body; //Declare new instant of class Body
        //Class Methods and behaviour
        public void Set_Car()
        <
                engine = new Engine0 ;
                engine.setEngine0 ; //Instantiate Engine object
                body = new Body!);
                body.setBody();//Instantiate Body object
        }
        public void Csr Describe!)
        i
                •sgtat.!tj$x<iBg*erJo* ()t
                hedyJody 5*s«rib« 9 ;
        }
        public void Car Status()
        /
                System-out, printlu< "Sixnr the |ia_ of Passengers");
                $*&&&$*** * lat                    (in. raadtin*!));
                ii dpwfttjjar* *■ 0)
                    {                          *C« i*              )? }
                •|t*
                    { Sfattar. out. peLnfclttf                 Css

                      if fsghasS > SB)
                            !^*a,e^,S>5dssli5f*S^                )
                      «lsa
                                                          Sbml %#•**};   )

        )
}
```

```
'the main program start here
pdblic class Module
{
        piblie static void main! ..)
        (
                // create new instance
                Car carl - new Car ();
                carl.setCar ();
                carl, describe!);
                carl.status ();
                // create new instance
                Car car2 = new Car ();
                car2.setCar ();
                car2. describe!);
                sttr2.fct«*s» i)i
        |
}
```

<u>**Please answer the following questions:**</u>

- Does the user assign a value to variable **"Brand"**? (Yes/No/Do not Know)
- Is the variable **"Passengers"** initialized to zero? (Yes/No/Do not Know)
- Does the user assign a value to variable "**Speed**"? (Yes/No/Do not Know)
- In **"Car_Status"** method in class **"Car"**, does **"Speed"** value assigned in the case of **"Passengers"** =zero? (Yes/No/Do not Know)
- Is the value of variable "**Speed**" assigned before the value of variable "**Passengers**"? (Yes/No/Do not Know)
- In **"Set_Car"** method in class **"Car"**, does the program instantiate engine before body? (Yes/No/Do not Know)
- Does the value of "**Passengers**" affect the value of "Speed"? (Yes/No/Do not Know)
- Does the value of "**Brand"** in class **"Body"** affect the value of **"Power"** in class **"Engine"**? (Yes/No/Do not Know)
- Does the program allow you to create new car with a certain brand and power? (Yes/No/Do not Know)
- Does the program allow you to change the car specifications (**Brand / Power**)? (Yes/No/Do not Know)
- Does the program compare number of passengers in two cars? (Yes/No/Do not Know)
- When the **"Car is Stopping"** statement is reached, is the value of "**Passengers"** > zero?(Yes/No/Do not Know)
- When the **"Over Speed"** statement is reached, is the value of "**Speed**" = 50? (Yes/No/Do not Know)

The listed program has been developed based on the concept of chunking every relevant code together into number of entities. This concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and function/methods. Fro the following questions please state which of the following entities were used?

- Entity called "***Body***", putting together all relevant attributes and functions, used to set and describe the body's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Wheels***", putting together all relevant attributes and functions, used to set and describe the wheels' specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "***Engine***", putting together all relevant attributes and functions, used to set and describe the engine's specifications. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Passengers***", putting together all relevant attributes and functions, used to set and describe the passengers' information. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "***Car***", putting together all relevant attributes and functions, used to set and describe the car's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.
- Entity called "***Lorry***", putting together all relevant attributes and functions, used to set and describe the lorry's specifications by communicating with the appropriate entities in the program. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity.

Ranking Question:

- How well do you understand the code?
    - not very well
    - fairly to moderately well
    - well to very well.

Thank you for your cooperation

Appendix B: Materials: Line-Edit Study

This appendix shows the experimental packet given to subjects in the Line-Edit experiment (described in Chapter 5), and includes:

5. The Line-Edit problem specification;
6. The experimental purpose and procedure
7. The programming background questionnaire.
8. The experimental treatments (problem solutions) and the corresponding comprehension questions. The ranking question at the end

The Line-Edit Problem's Specification

A piece of text consisting of words separated by one or more space character is terminated by an *.

It is required to convert it to line by line form in accordance with the following rules:

a) Redundant spaces between words are to be removed;
b) No line will contain more than m characters and each line is filled as far as possible;
c) Line-breaks must not occur in the middle of a word.

(You may ignore the presence of line-feed character and the possibility of a word being greater than m character).

Design a program to read the text and output it in accordance with the above rules.

Experimental purposes and procedures

During this lab session, you will be asked to take part in a short experiment. This experiment is a part of a research aims to investigate program comprehension for Object Oriented concepts. Obviously, we could simply ask you this question directly, but as you can probably guess, doing so wouldn't necessarily give us the types of answers which might be useful: for example, the answers might not be quantifiable in a consistent way and hence will be not suitable for any sort of statistical analysis. We hope to collect data which can be analysed, and which either support work which has already done or offer new and possibly conflicted evidence. If you are interested obtaining more information on this research, please let me know.

Before starting the experiment, you should be aware that:

- This is no way a test of your programming knowledge level will not be used in any form as part of your assessment on the all courses.
- You are not expected to get all of the answers right. What we are interested in here are the types of questions which people get wrong compared to those they get right.
- The data collected will be both anonymous and strictly confidential and will be only used in the purposes of this research.

Now, the experiment has three stages. At the first stage, you will be given a programming background questionnaire that contains two sections. The first section asking about personal details, please note that giving these details is optional. The second section asking you about you programming experience, please fill this in, and then give it to your experimenter.

In the second stage, you will be given a booklet of two pages to work through. The booklet contains a computer program and a number of questions you are asked to answer. Please do not start work with the program unless the experimenter tells you to do so. Once you told to start please spend some time reading the program before you go to the questions. If you finish or you decide to stop or withdraw from the experiment please tel the experimenter immediately.

The last stage you will have a question asking you to rank the program you had just gone through please answer this and give it back to the experimenter.

Note that it will not be possible for the experimenter to give you details of the experiment through the whole experimental period, as knowing the hypotheses may influence how you respond[5]. If you have any questions about these instructions, please ask the experimenter now.

Thank toy for taking part of this experiment.

[5] For the same reason, please do not discuss you ideas on the experimental program you given with people who have not taken part in this week's lab session.

Programming Background Questionnaire

Thank you for taking the time to complete the following questions, through the following questions you will be asked about your programming experience, please fill this in. The information you provide will be used in a study to investigate program comprehension for Object Oriented concepts and will be treated confidentially.

Personal Details:

Name (optional) ______________________ Age: (Optional)

Course/Module: ______________________ Gender: Male / Female ________

Programming Experience:

F or the following programming languages, please indicate:

4. How you learnt the language (School, University, Work, Self taught).
5. Rate yourself according to your level of knowledge (1- Novice, 5-Expert).
6. Add any Programming language which is missed.

	School	University	Work	Self Taught	Rate of Knowledge (1 to 5)
Basic					
Fortran					
Logo					
Pascal					
C					
Visual Basic					
C++					
Java					
C#					

The Experimental treatments

7.3.1.5 B1. Visual Basic Non-Object based Experimental Treatment

The Line-Edit Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Line Edit". It deals with character, word, and line.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:			VB *Non-Object* based			Subject code			
Date		Place:							
Starting Time				Ending Time					
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Progra m Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
Module _ineEdit
    Sub MainQ
        Dirr Textlndex, MaxLineLength, WordLength, LineLength As Integer
        Dins NewCharacter As Char
        Dirr TextBody, NewWord As String
        Textlndex = 8
        MaxLineLength = WordLength = LineLength = 8
        NewWord = ""
        Console.WriteLine("enter the original text")
         'insert the original text into string TextBody
        TextBody = Console.ReadLine()
        Console.Write.ine("Enter the TaxiruT line length")   ^--------------   1
         'the rraxirurr line length given by user
        MaxLineLength = Val(Console.Readline())
        NewCharacter = TextBody.Chars(TextIndex)
        While (NewCharacter <> "'")
            If (NewCharacter = " ") Then
                If 'WordLength <>8 Then
                     'output a word on current or new line
                    If LineLength + Word.ength <= Max.ine.ength Then
                        LineLength = LineLength + 1
                    Else 'strats a new line and reset the LineLength to zero
                        Console.Writeline()   <----------------------------   4
                                                                               v
                        LineLength = 8
                    End If
                     'print out a built word followed by space and reset the
                     'WordLength to start building a new word
I                   Console.'Write(NewWord.Substring(8, 'Word.ength) + NewCharacter)
                    LineLength = LineLength + 'WordLength *----------------   2   !
                    NewWord =
                    Word.ength = 8
                End If
            Else
                 'build a word by adding up characters to the NewWrod string
                NewWord = NewWord.Insert(WordLength, NewCharacter)
                Word.ength = Word.ength + 1
            End If
             'pull out a new character
            Textlndex = Textlndex + 1
            NewCharacter = TextBody.Chars(TextIndex\   ^                        3
        End While
        Console.Read.ine()
    End Sub
End Mule
```

Please answer the following questions:

- Does the program contain the code fragment: "*NewCharacter = TextBody.Chars(TextIndex)"*? (Yes/No/Don't Know)
- Does the program contain the code fragment: *"LineLength = WordLength + MaxLineLength"?* (Yes/No/Don't Know)
- Does the program check the "*LineLength*" value before starting output the new built word? (Yes/No/Don't Know)
- Does the program start building a word before check "*WordLength*" value? (Yes/No/Don't Know)
- Does the value of "*WordLength"* affect the value of "*LineLength*"? (Yes/No/Don't Know)
- Does the value of "*LineLength"* affect the value of "*MaxLineLength*"? (Yes/No/Don't Know)
- Does the program remove any spaces within the input text? (Yes/No/Don't Know)
- Does the program output the original text in upper case format? (Yes/No/Don't Know)
- When the statement labelled with number ① is reached, is the original text entered? (Yes/No/Don't Know)
- When the statement labelled with number ② is reached, is the value of "*WordLength*" > 0? (Yes/No/Don't Know)
- When the statement labelled with number ③ is reached, is the value of "*TextIndex*" = 0? (Yes/No/Don't Know)
- When the statement labelled with number ④ is reached, is the value of "*LineLength*" = 0? (Yes/No/Don't Know)

Now, if you were asked to develop the same program based on the concept of chunking every relevant code together into number of entities, this concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and functions/methods. Fro the following 4 questions please state which of the following entities would be useful?

- Entity called "*Word*", putting together all relevant attributes and functions, used to build a word from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Building Word*", putting together all relevant attributes and functions, used to edit a given piece of text by communicating with "*Word*" entity. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Read Character*", putting together all relevant attributes and functions, used to read character from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Print Word*", putting together all relevant attributes and functions, used to print the built word. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity

- **<u>Ranking Question:</u>**
- How well do you understand the code?
    - not very well
    - fairly to moderately well
    - well to very well.

Thank you for your cooperation

7.3.1.6 B2. Visual Basic Object based Experimental Treatment

The Line-Edit Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Line Edit". It deals with character, word, and line.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:		VB *Object* based			Subject code				
Date		Place:							
Starting Time			Ending Time						
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Progra m Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
        Class Word
            Public Textlndex As Integer = 8
            Public Word_ength As Integer = 8
            Public NewCharacter As Char
            Public NewWord As String = ""
            Public Sub Build'Word(ByVal Text As String)
                'build a word by adding up characters to the NewWrod string
                NewWord = NewWord .Insert(WordLength, NewCharacter )
                'pull out a new character
                Textlndex = Textlndex + 1
                NewCharacter = Text.Chars(Textlndex) ^_________________  3
                WordLength = WordLength + 1
            End Sub
            Function OutputWord(ByVal Text As String) As String
                NewWord.Substring(8, Word_ength)
                Return NewWord
            End Function
        End Class

  Class BuildingWords
      Private LineLength As Integer = 8
      Private word As New Word 'creates new instance of class Word
      Public Sub TextEdit(ByVal Text As String, ByVal MaxLineLength As Integer )
          word.NewCharacter = Text.Chars(word.Textlndex)
          While word.NewCharacter <>
              While word.NewCharacter = " "
                  'pull out and read the next character
                  word.Textlndex = word.Textlndex + 1
                  word.NewCharacter = Text.Chars(word.Textlndex)
              End While
              word.NewWord =
              word.Word_ength = 8
              While ((word.NewCharacter <> " ") And (word.NewCharacter <> "*"))   :
                  word.BuildWord(Text)
              End While
              'output a word on current or new line
              If -ine^ength + word,Kord_ength <= MaxLineLength Then
                  LineLength = LineLength + 1
              Else 'strats a new line and reset the LineLength to zero
                  Console .WriteLine( )     4-------------------------------------  4
                  LineLength = 8
              End If
              'print out a built word followed by space and reset the
              'WordLength to start building a new word
              Console.Write(word.OutputWord(Text) + " ")
              LineLength = Line.ength + word .WordLength-4-----------------------  2
          End While
      End Sub
  End Class


      Module Modulel
          Sub Main()
              Dim TextBody As String
              Dim MLL As Integer = 8
              Dim B'w'ords As New Building,Lords ()
              Console.WriteLine("enter the original text")
               'insert the original text into string TextBody
              TextBody = Console.ReadLine()
              Console.WriteLine("enter the Maximum line length") 4---_   1
               'the maximum line length given by user
              MLL = Val(Console.ReadLine())
               'creates new instance of class BuildingWord
              BWords.TextEdit(TextBody, MLL)
              Console.Readtine()
          End Sub
      End Module
```

Please answer the following questions:

- Does the program contain the code fragment: **"word.*NewCharacte =Text.Chars(word.TextIndex)*"**? (Yes/No/Don't Know)
- Does the program contain the code fragment: ***"LineLength = WordLength + MaxLineLength"***? (Yes/No/Don't Know)
- Does the program check the "***LineLength***" value before starting output the new built word? (Yes/No/Don't Know)
- Does the program start building a word before check "***word.WordLength***" value? (Yes/No/Don't Know)
- Does the value of "*word.WordLength*" affects the value of "***LineLength***"? (Yes/No/Don't Know)
- Does the value of "***LineLength***" affect the value of "***MaxLineLength***"? (Yes/No/Don't Know)
- Does the program remove any spaces within the input text? (Yes/No/Don't Know)
- Does the program output the original text in upper case format? (Yes/No/Don't Know)
- When the statement labelled with number ① is reached, is the original text entered (Yes/No/Don't Know)
- When the statement labelled with number ② is reached, is the value of "*word.WordLength*" > 0? (Yes/No/Don't Know)
- When the statement labelled with number ③ is reached, is the value of "*TextIndex*" = 0? (Yes/No/Don't Know)
- When the statement labelled with number ④ is reached, is the value of "*LineLength*" = 0? (Yes/No/Don't Know)

The listed program has been developed based on the concept of chunking every relevant code together into number of entities. This concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and function/methods. Fro the following 4 questions please state which of the following entities were used?

- Entity called "*Word*", putting together all relevant attributes and functions, used to build a word from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Building Word*", putting together all relevant attributes and functions, used to edit a given piece of text by communicating with "*Word*" entity. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Read Character*", putting together all relevant attributes and functions, used to read character from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Print Word*", putting together all relevant attributes and functions, used to print the built word. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity

Ranking Question:

- How well do you understand the code?
    - not very well
    - fairly to moderately well
    - well to very well.

Thank you for your cooperation

7.3.1.7 B3. Java Non-Object based Experimental Treatment

The Line-Edit Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Line Edit". It deals with character, word, and line.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:		JAVA *Non-Object* based			Subject code				
Date		Place:							
Starting Time			Ending Time						
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Progra m Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
public class lineEditCharacterBased{
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreaiReaderfSysten.in));
        int Textlndex, MaxLineLength, WordLength, LineLength;
        char NewCharacter;
        String NewWord = new Stringf);
        Textlndex = MaxLineLength = WordLength = LineLength =0;
        System.out.println("enter the original text");
        //insert the original text into string TextBody
        String Textbody = in.readLine!);
        System.out.printlnf"enter the Maximum line length11);            <                  1
        //the maximum line length given by user
        MaxLineLength = Integer.parselnt(in.readLinef));
        NewCharacter = Textbody.charAt(TextIndex);
        while (NewCharacter != '*') {
            if (NewCharacter == ' '){
                if (WordLength != 0){
                    //output a word on current or new line
                    if (LineLength + WordLength <= MaxLineLength) {
                        LineLength ++;
                    } else {
                        //strats a new line and reset the LineLength to zero
                        System.out.printlnf);              M_ _ _ _ _ _ _ _ _ _ _ _     4
                        LineLength = 0;

                    }
                    //print out a built word followed by space and
                    //reset the WordLength to start build a new word
                    System.out.print(NewWord,substring)0,WordLength)+ NewCharacter);
                    LineLength = LineLength + WordLength;        <- - - - - - - - -    2
                    NewWord = "";
                    WordLength = 0;

                }
            } else {
                //build a word by adding up characters to the NewWord string
                NewWord = new StnngBufferfNewWord(.insert(WordLength, NewCharacter). toString0;
                WordLength ++;

            }
            //pull out a new character
            Textlndex ++;
            NewCharacter = Textbody.charAt(TextIndex);     <- - - - - - - - - - - - - - - -  3

        }
        System.out.printlnf);

    }
}
```

Please answer the following questions:

- Does the program contain the code fragment: "*NewCharacter = Textbody.charAt(TextIndex);"*? (Yes/No/Don't Know)
- Does the program contain the code fragment: *" LineLength = WordLength + MaxLineLength;"*? (Yes/No/Don't Know)
- Does the program check the "*LineLength*" value before starting output new word? (Yes/No/Don't Know)
- Does the program start building a word before check the value of "*WordLength*"? (Yes/No/Don't Know)
- Does the value of "*WordLength*" affect the value of "*LineLength*"? (Yes/No/Don't Know)
- Does the value of "*LineLength*" affect the value of "*MaxLineLength*"? (Yes/No/Don't Know)
- Does the program remove any spaces within the input text? (Yes/No/Don't Know)
- Does the program output the original text in upper case format? (Yes/No/Don't Know)
- When the statement labelled with number ① is reached, is the original text entered? (Yes/No/Don't Know)
- When the statement labelled with number ② is reached, is the value of "*WordLength*" > 0? (Yes/No/Don't Know)
- When the statement labelled with number ③ is reached, is the value of "*TextIndex*" = 0? (Yes/No/Don't Know)
- When the statement labelled with number ④ is reached, is the value of "*LineLength*" = 0? (Yes/No/Don't Know)

Now, if you were asked to develop the same program based on the concept of chunking every relevant code together into number of entities, this concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and functions/methods. Fro the following 4 questions please state which of the following entities would be useful?

- Entity called "*Word*", putting together all relevant attributes and functions, used to build a word from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Building Word*", putting together all relevant attributes and functions, used to edit a given piece of text by communicating with "*Word*" entity. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Read Character*", putting together all relevant attributes and functions, used to read character from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Print Word*", putting together all relevant attributes and functions, used to print the built word. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity

Ranking Question:

- How well do you understand the code?
  - not very well
  - fairly to moderately well
  - well to very well.

Thank you for your cooperation

7.3.1.8 B4. Java Object based Experimental Treatment

The Line-Edit Study Booklet

Please do not turnover the page until you asked to do.

The name of the program listed over leaf is "Line Edit". It deals with character, word, and line.

Once you being asked to turn over the page, please give your self time to read all the given code before starting on the questions

for researcher use									
Program Version:		JAVA *Object* based			Subject code				
Date		Place:							
Starting Time				Ending Time					
Code	**Elementary Operation**	**Control Flow**	**Data Flow**	**Progra m Goals**	**State**	**Problem Classes**	**Total Performance**	**Time**	**Rank Response**
score									

Please read the code below:

```
public class Word{
    public int TextIndex= 0;
    public int WordLength = 0;
    public char NewCharacter;
    String NewWord = new String();
    public void BuildWord(String Text){
        //build a word by adding up characters to the NewWrod string
        NewWord = new StringBuffer(NewWord).insert(WordLength, NewCharacter).toString();
        //pull out a new character
        TextIndex++;
        NewCharacter = Text.charAt(TextIndex);  ←———————————— 3
        WordLength ++;
    }
    public String PrintWord(String Text){
        NewWord = NewWord.substring(0,WordLength);
        return NewWord;
    }
}
```

```
public class BuildingWords{
    private int LineLength = 0;
    Word word = new Word(); //creates new instance of class Word
    public void TextEdit(String Text, int MaxLineLength) {
        word.NewCharacter = Text.charAt(word.TextIndex);
        while (word.NewCharacter != '*') {
            while (word.NewCharacter ==' ') {
                //pull out and read the next character
                word.TextIndex ++;
                word.NewCharacter = Text.charAt(word.TextIndex);
                }
                word.NewWord = " "
                word.WordLength = 0;
                while ((word.NewCharacter != ' ') && (word.NewCharacter != '*')) {
                        word.BuildWord(Text);
                }
                //output a word on current or new line
                if (LineLength + word.WordLength <= MaxLineLength){
                                LineLength ++;
                                } else {
                                    //strats a new line and reset the LineLength to zero
                                    System.out.println();←———————————— 4
                                    LineLength = 0;
                                    }
                        //print out a built word followed by space and
                        //reset the WordLength to start building a new word
                        System.out.print(word.PrintWord(Text) + " ");
                        LineLength = LineLength + word.WordLength;←———— 2
        }
    }
}
```

```
public class LineEditWordBase{
    public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String Textbody = new String();
        int MLL = 0;
        BuildingWords BWords = new BuildingWords();
        System.out.println("enter the original text");
        //insert the original text into string TextBody
        Textbody = in.readLine();
        System.out.println("enter the Maximum line length"); ←—— 1
        //the maximum line length given by user
        MLL = Integer.parseInt(in.readLine());
        //creates new instance of class BuildingWord
        BWords.TextEdit(Textbody, MLL);
        System.out.println();
    }
}
```

Please answer the following questions:

- Does the program contain the code fragment: ***"word.NewCharacter = Text.charAt(word.TextIndex);"***? (Yes/No/Don't Know)
- Does the program contain the code fragment: ***"LineLength = WordLength + MaxLineLength;"***? (Yes/No/Don't Know)
- Does the program check the "***LineLength***" value before starting output new word? (Yes/No/Don't Know)
- Does the program start building a word before check the value of "***word.WordLength***"? (Yes/No/Don't Know)
- Does the value of "***word.WordLength***" affects the value of "***LineLength***"? (Yes/No/Don't Know)
- Does the value of "***LineLength***" affect the value of "***MaxLineLength***"? (Yes/No/Don't Know)
- Does the program remove any spaces within the input text? (Yes/No/Don't Know)
- Does the program output the original text in upper case format? (Yes/No/Don't Know)
- When the statement labelled with number ① is reached, is the original text entered (Yes/No/Don't Know)
- When the statement labelled with number ② is reached, is the value of "***word.WordLength***" > 0? (Yes/No/Don't Know)
- When the statement labelled with number ③ is reached, is the value of "***TextIndex***" = 0? (Yes/No/Don't Know)
- When the statement labelled with number ④ is reached, is the value of "***LineLength***" = 0? (Yes/No/Don't Know)

The listed program has been developed based on the concept of chunking every relevant code together into number of entities. This concept is called a "CLASS" in Object-Oriented programming. Each entity/Class has its own attributes and function/methods. Fro the following 4 questions please state which of the following entities were used?

- Entity called "*Word*", putting together all relevant attributes and functions, used to build a word from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Building Word*", putting together all relevant attributes and functions, used to edit a given piece of text by communicating with "*Word*" entity. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Read Character*", putting together all relevant attributes and functions, used to read character from the given piece of text. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity
- Entity called "*Print Word*", putting together all relevant attributes and functions, used to print the built word. (Yes/No/Don't Know). If yes, please write names of attributes and functions of this entity

Ranking Question:

- How well do you understand the code?
  - not very well
  - fairly to moderately well
  - well to very well.

Thank you for your cooperation

Appendix C: Ethical Approval

The following research ethical approval form was used at Sheffield Hallam University

Faculty of ACES Research Ethics Checklist

This form is designed to help students and staff to complete an ethical scrutiny of their proposed research. It also enables the University and Faculty to keep a record of research conducted that has been subjected to ethical scrutiny.

Answering the questions below will help decide whether your research proposal requires ethical review by the Faculty Research Ethics Committee (FREC). In cases of uncertainty members of the FREC can be approached for advice, or alternatively students and staff can refer to the SHU Research Ethics Policy. The large majority of research proposals will not need further scrutiny after completion of this form.

For staff research the form should be completed by the principal investigator. For student projects it may be completed by the student or the supervisor. In all cases it should be counter signed by the supervisor and kept as a record that ethical scrutiny has occurred. The final responsibility for ensuring that ethical research practices are followed rests with the supervisor for student research and the principal investigator for staff research projects.

Please note it may still be necessary to conduct a *risk assessment* for the proposal – for information contact the Faculty Safety Co-ordinator.

Name of student or principal investigator	
Name of supervisor (if applicable)	
Title of research proposal	
Outline of investigation	

	Question	Yes/No
1.	Does the research involve human participants? This includes surveys, questionnaires, observing behaviour etc.	

If NO please go to question No. 6.

If YES, then please answer the following questions No. 2 - 5:

2.	Will any of the participants be vulnerable? (E.g. Young people under 18, people with learning disabilities, people who may be limited by age or sickness or disability from understanding the research, etc.)	
3.	Is there any reasonable and foreseeable risk of physical or emotional harm to any of the participants? (E.g. Distressing or intrusive interview questions, uncomfortable procedures involving the participant, invasion of privacy, topics relating to	

	highly personal information, topics relating to illegal activity)	
4.	Will anyone be taking part without giving their informed consent? (E.g. Research involving covert study, coercion of subjects, or where subjects have not fully understood the research etc.)	
5.	Will the research output allow identification of any individual who has not given their express consent to be identified?	

If the answer to any of the questions 2 - 5 is YES then the research proposal must be submitted to the FREC for approval *unless* it falls into a category/programme of research that has already received **category approval.**

6.	Does the research involve the use of live animals?	

If the answer to questions 6 is YES then the research proposal must be submitted to the FREC for approval *unless* it falls into a category/programme of research that has already received **category approval.**

7.	Does the research require approval from any external ethics com-mittee, e.g. the NHS? For NHS research, this includes any service evaluation work, work concerning NHS Patients (tissues, organs, personal information or data), NHS staff, volunteers, carers, NHS premises or facilities.	

If the answer to question 7 is YES then the research proposal must be submitted to the relevant external body. For NHS

Research Ethics Committees please refer to http://www.corec.org.uk

If the research proposal does not require submission to either the FREC or an NHS or other external REC then **standard approval** applies.

If the research proposal requires submission to the FREC please contact a member of the committee for more information.

Approval awaited applies until the proposal has been considered by the FREC.

ETHICAL APPROVAL (please tick):

□ (*Standard approval)* This project does not require specific ethical approval.

□ *(Category approval)* In my opinion this work falls within the category of .... projects which has been previously approved by the FREC and it does not therefore need individual approval.

□ *(Approval awaited)* This project must be referred to the FREC for individual consideration – the work must not proceed unless and until the FREC gives approval.

I can confirm that I have read the Sheffield Hallam University Research Ethics Policy and Procedures document and agree to abide by its principles (please tick). □

Signed .... Name..............................
Date ....
Student / Researcher/ Principal Investigator (as applicable)

Signed .... Name..............................

Date ....
Supervisor or other person giving ethical sign-off
Note: University Research Ethics policy available from the following web link: http://www.shu.ac.uk/research/researchhallam.html

Students - If standard approval applies, please return this form to your supervisor before starting your research, and retain a copy for inclusion in your research report.

Staff - If standard approval applies, please keep this form for your own records.

Appendix D: Normality tests and Experimental Data

D1. Normality Test of the Car Study

Due to the relatively large number of subjects, timing data in minutes and the subjects' responses are presented in Appendix D. At the first stage, normality tests were run. These tests were used to check whether the collected data are normally distributed or not. Upon the tests results, the decision would be made to follow either parametric statistical tests or equivalent non-parametric statistical tests. Firstly, formal skewness and kurtosis tests were performed. Skewness involves the symmetry of the distribution and kurtosis involves the peakedness of the distribution. Both skewness and kurtosis are 0 in a normal distribution, so the farther away from 0, the more non-normal the distribution. Table 5.3 shows the descriptive statistics results about the dependent variables, including the value of skewness and kurtosis, with accompanying standard error for each.

Table D.1: Descriptive statistics, skewness and kurtosis tests results for normality for Car study

descriptive						
	mean	SD	skewness		kurtosis	
	statistic	statistic	statistic	std. error	statistic	std. error
time	12.59	2.49	0.08	0.13	-0.56	0.26
performance	59.19	18.49	-0.49	0.13	0.05	0.26

From table D.1 the mean for *time* was 12 minutes (SD = 2.49). The *time* was non-normally distributed, with skewness of 0.08 (SE = 0.13) and kurtosis of 0.56 (SE = 0.26). The *time* distribution clustered to the left, the tail extending to the right with flat distribution. The mean of *performance* was 59% (SD = 18.49), *performance* was non-normal with skewness of -0.49 (SE = 0.13) and kurtosis of 0.05 (SE = 0.26). The *performance* distribution clustered to the right, the tail extending to the left with non-flat distribution. However, how much skewness or kurtosis render the data non-normal could be an arbitrary determination, is sometimes difficult to interpret using the values of skewness and kurtosis. Luckily, there are more objective tests of normality.

The descriptive statistics for skewness and kurtosis are not as informative as established tests for normality that take skewness and kurtosis into account simultaneously. The kolmogorov-smirnov test (k-s) and shapiro-wilk (s-w) test are designed to test normality by comparing data to a normal distribution with the same mean and standard deviation of the sample (De Sá, 2007).

Since the sample size was greater than 50, the kolmogorov-Smirnov (k-s) test was carried out to check the normality of the data. The value of the significant (sig) column is the most important value that needs to be checked in a test of normality. In general, a sig. value less than or equal to 0.05 is considered good evidence that the data set is not normally distributed. Table D.2 illustrates the results of the kolmogorov-smirnov (k-s) normality test.

Table D.2: The kolmogorov-smirnov test results for normality for Car study

tests of normality			
	kolmogorov-smirnov (k-s)		
	statistic	df	sig.
time	.098	353	.000
performance	.118	353	.000

Table D.2 shows that all variables have sig values < 0.05, therefore *time* and *performance* can be assumed to be non-normally distributed (*time*: p = 0.000 and *performance*: p = 0.000).

D2 Normality Tests for the Line-Edit Study

Timing data in minutes and the subjects' responses are presented in Appendix D. Normality tests were applied, and formal skewness and kurtosis tests were performed. Table D.3 shows the descriptive statistics results about the dependent variables, including the value of skewness and kurtosis, with accompanying standard error for each.

Table D.3: Descriptive statistics, skewness, and kurtosis tests results of normality in the Line-Edit study

descriptive						
	mean	SD	skewness		kurtosis	
	Statistic	Statistic	Statistic	Std. Error	Statistic	Std. Error
time	16.71	3.51	0.01	0.31	-0.64	0.63
performance	60.26	16.44	-0.21	0.31	.000	0.63

From table D.3 the mean of *time* was 16 minutes (SD = 3.51), *time* was non-normally distributed, with skewness of 0.01 (SE = 0.31) and kurtosis of -0.64 (SE = 0.63). The *time* distribution clustered to the left with the tail extending to the right with flat distribution. The mean of *performance* was 60% (SD = 16.44), *performance* was non-normal with skewness of -0.21 (SE = 0.31) and kurtosis of 0.00 (SE = 0.63). The *performance* distribution clustered to the right with the tail extending to the left with non-flat distribution.

As for the Car study, the kolmogorov-smirnov test (k-s) test was applied to test normality by comparing data to a normal distribution with the same mean and standard deviation of the sample. Table D.4 illustrates the results of the kolmogorov-smirnov (K-S) normality test.

Table 0.4: The kolmogorov-smirnov test results of normality in the Line-Edit study

Tests of Normality			
	kolmogorov-smirnov (k-s)		
	Statistic	df	Sig.
time	.125	56	.029
performance	.233	56	.000

Table D.4 shows that all variables have sig values < 0.05, therefore *time* and *performance* can be assumed to be non-normally distributed (*time*: p = 0.029 and *performance*: p = 0.000). Consequently, as data distribution was shown to be non-normal, corresponding non-parametric Mann-Whitney U and Kruskal-Wallis tests were calculated where appropriate.

D3. Experimental Database

This appendix presents the database for each study run, i.e., one data based for the Car study and one database for the Line-Edit study. The logical variables from these two studies are presented.

The logical groupings of data for variables in table D.5 and table D.6 are now explained. Variable (1) represents the subject's **Age**. Variable (2) **Gender** is graded as: 1≡ male and 2 ≡ female. Variables from (3) to (8) represent the response for **Elementary Operations, Control Flow, Data Flow, Program Goals, State, and Problem Classes** knowledge categories respectively. Variable (9) represents the ***performance***, which is the sum of all knowledge categories. Variable (10) ***time*** is the time taken to accomplish the comprehension task, in minutes, by each subject individually. Finally, Variable (11) ***rank*** is graded as: 1≡not very well, 2 ≡ fairly to moderated well, and 3 ≡

well to very well. For table D.5, the first 176 row represent the data of subjects who were given the *Non-Object* based program version, and the rest of the table represent the data of subjects who were given the *Object* based program version. For D.6 table, the first 28 rows represent the data of subjects who were given the *Non-Object* based program version; the other 28 rows represent the data of subjects who were given the *Object* based program version.

Table D.5: Row data for the run of the Car study

	Experimental Variables										
	1	2	3	4	5	6	7	8	9	10	11
1	21	1	66.67	66.67	100	66.67	100	83.33	78.95	11	1
2	18	1	33.33	33.33	100	33.33	100	0	36.84	13	1
3	19	1	66.67	33.33	100	66.67	100	50	63.16	12	1
4	21	1	33.33	66.67	100	66.67	100	50	63.16	13	1
5	19	1	66.67	100	50	66.67	0	66.67	63.16	12	1
6	19	1	66.67	66.67	0	0	0	33.33	31.58	12	1
7	18	1	33.33	66.67	0	33.33	0	83.33	47.37	11	1
8	20	2	66.67	66.67	100	100	100	83.33	84.21	11	1
9	19	2	33.33	33.33	50	66.67	100	50	52.63	12	1
10	20	2	100	66.67	0	66.67	0	66.67	57.89	12	1
11	20	1	66.67	66.67	100	33.33	100	66.67	68.42	10	1
12	20	1	33.33	33.33	50	66.67	100	66.67	57.89	16	1
13	20	1	33.33	100	100	33.33	100	66.67	68.42	17	1
14	21	1	33.33	66.67	100	33.33	100	16.67	47.37	18	1
15	19	1	33.33	0	100	33.33	50	66.67	47.37	7	1
16	19	1	66.67	33.33	100	66.67	50	66.67	63.16	12	1
17	19	1	66.67	100	50	66.67	100	83.33	78.95	12	1
18	18	1	33.33	0	0	0	100	33.33	26.32	8	1
19	20	1	33.33	33.33	50	100	50	50	52.63	13	1
20	19	1	33.33	33.33	0	66.67	0	66.67	42.11	6	1
21	19	1	100	33.33	50	0	100	0	36.84	17	1
22	18	1	100	66.67	50	66.67	100	66.67	73.68	10	1
23	21	1	66.67	33.33	100	33.33	100	50	57.89	12	1
24	20	1	33.33	66.67	100	0	50	0	31.58	10	1
25	19	2	33.33	33.33	100	33.33	100	66.67	57.89	10	1
26	19	2	33.33	100	100	33.33	100	66.67	68.42	10	1
27	20	1	100	66.67	100	0	100	50	63.16	10	1
28	22	1	33.33	33.33	100	100	100	66.67	68.42	14	1
29	23	2	66.67	66.67	50	66.67	100	50	63.16	11	1
30	20	2	66.67	66.67	100	100	100	16.67	63.16	14	1
31	20	2	33.33	66.67	100	66.67	50	50	57.89	10	1
32	21	1	66.67	33.33	0	33.33	100	33.33	42.11	12	1
33	19	2	66.67	100	0	33.33	100	66.67	63.16	10	1
34	21	2	33.33	100	100	66.67	100	50	68.42	14	1
35	20	2	66.67	100	100	66.67	100	50	73.68	12	1
36	24	1	100	66.67	100	0	100	50	63.16	17	1

37	19	1	66.67	33.33	50	66.67	50	33.33	47.37	10	1
38	20	1	0	100	100	66.67	0	33.33	47.37	10	2
39	20	1	33.33	33.33	50	33.33	100	0	31.58	10	2
40	20	2	33.33	33.33	50	66.67	50	33.33	42.11	16	2
41	20	2	66.67	66.67	100	66.67	100	33.33	63.16	12	2
42	20	2	66.67	0	50	33.33	0	0	21.05	12	2
43	20	2	100	100	50	100	100	33.33	73.68	11	2
44	20	2	33.33	100	50	33.33	100	50	57.89	12	2
45	20	2	66.67	66.67	0	0	50	33.33	36.84	10	2
46	21	2	33.33	0	0	33.33	50	0	15.79	13	2
47	21	2	100	66.67	50	66.67	50	66.67	68.42	14	2
48	21	2	66.67	66.67	100	33.33	100	66.67	68.42	14	2
49	21	2	66.67	33.33	50	66.67	100	0	42.11	15	2
50	21	2	33.33	33.33	50	33.33	50	0	26.32	17	2
51	21	2	66.67	66.67	50	66.67	100	50	63.16	11	2
52	22	1	66.67	33.33	50	33.33	0	33.33	36.84	16	2
53	22	1	66.67	66.67	0	33.33	100	16.67	42.11	13	2
54	22	2	33.33	66.67	100	33.33	100	50	57.89	10	2
55	23	2	33.33	0	0	0	50	33.33	21.05	15	2
56	23	2	33.33	0	50	33.33	100	16.67	31.58	14	2
57	23	2	66.67	33.33	50	33.33	100	50	52.63	14	2
58	23	2	66.67	66.67	100	33.33	50	66.67	63.16	8	2
59	23	2	100	100	50	66.67	100	33.33	68.42	9	2
60	24	2	33.33	0	100	66.67	50	0	31.58	15	2
61	24	1	66.67	33.33	0	33.33	50	0	26.32	11	2
62	25	2	66.67	33.33	0	0	50	16.67	26.32	8	2
63	22	2	0	33.33	0	0	100	0	15.79	15	2
64	22	2	66.67	66.67	0	33.33	100	16.67	42.11	13	2
65	22	2	100	66.67	50	33.33	100	33.33	57.89	16	2
66	23	2	66.67	100	50	66.67	100	33.33	63.16	16	2
67	22	2	66.67	100	50	66.67	100	16.67	57.89	13	2
68	22	2	66.67	100	50	66.67	100	16.67	57.89	15	2
69	21	2	100	33.33	50	66.67	100	66.67	68.42	15	2
70	24	2	33.33	33.33	50	66.67	100	33.33	47.37	14	2
71	21	1	100	100	100	66.67	100	66.67	84.21	11	2
72	20	2	100	100	100	100	100	16.67	73.68	11	2
73	20	2	66.67	100	50	66.67	100	50	68.42	16	2
74	21	2	100	66.67	0	66.67	100	50	63.16	16	2
75	20	1	100	66.67	100	100	100	16.67	68.42	17	2
76	28	1	66.67	100	100	66.67	100	33.33	68.42	13	3

77	21	2	66.67	66.67	50	100	100	33.33	63.16	17	3
78	22	1	66.67	33.33	0	100	0	16.67	36.84	14	3
79	21	1	100	66.67	100	100	100	50	78.95	10	3
80	20	2	0	0	0	66.67	100	16.67	26.32	14	3
81	18	1	66.67	66.67	100	100	100	50	73.68	14	3
82	19	2	33.33	66.67	0	0	50	50	36.84	14	3
83	21	2	100	0	0	0	50	0	21.05	14	3
84	19	2	66.67	66.67	50	66.67	50	33.33	52.63	13	3
85	20	2	66.67	33.33	50	66.67	50	0	36.84	13	3
86	22	1	66.67	100	50	66.67	100	16.67	57.89	14	3
87	20	1	0	33.33	100	0	50	50	36.84	13	3
88	19	1	0	0	0	33.33	0	0	5.26	11	3
89	18	1	66.67	33.33	0	66.67	100	0	36.84	11	3
90	19	2	33.33	33.33	50	33.33	0	16.67	26.32	11	3
91	20	1	33.33	66.67	100	66.67	100	50	63.16	10	3
92	21	1	33.33	66.67	0	66.67	100	66.67	57.89	10	3
93	19	1	0	0	0	0	0	0	0	11	3
94	22	2	66.67	33.33	0	33.33	100	33.33	42.11	14	3
95	19	2	100	100	100	66.67	100	83.33	89.47	11	3
96	19	2	100	66.67	0	33.33	100	66.67	63.16	14	3
97	20	2	33.33	66.67	100	33.33	50	16.67	42.11	17	3
98	20	2	33.33	66.67	50	33.33	100	33.33	47.37	15	1
99	23	2	66.67	100	100	66.67	50	16.67	57.89	13	1
100	21	2	100	100	50	66.67	50	50	68.42	9	1
101	22	1	66.67	33.33	50	33.33	100	66.67	57.89	9	1
102	20	1	66.67	66.67	100	66.67	50	83.33	73.68	11	1
103	21	2	100	33.33	0	66.67	0	16.67	36.84	9	1
104	22	2	66.67	0	0	33.33	100	83.33	52.63	10	1
105	23	2	33.33	33.33	50	66.67	0	83.33	52.63	11	1
106	21	2	33.33	0	50	33.33	100	0	26.32	12	1
107	19	1	0	66.67	100	66.67	100	66.67	63.16	10	2
108	19	1	100	100	100	100	100	66.67	89.47	17	2
109	21	1	0	66.67	50	0	100	100	57.89	10	2
110	23	1	66.67	66.67	50	66.67	100	0	47.37	11	2
111	19	2	33.33	66.67	100	66.67	100	0	47.37	15	2
112	20	2	66.67	66.67	100	66.67	100	50	68.42	15	2
113	22	2	66.67	66.67	100	66.67	100	50	68.42	12	2
114	21	2	66.67	100	0	100	100	16.67	57.89	11	2
115	19	2	33.33	66.67	50	33.33	100	33.33	47.37	12	2
116	19	2	100	66.67	50	66.67	100	50	68.42	14	2

117	20	1	33.33	0	50	66.67	100	100	63.16	11	2
118	23	2	100	66.67	100	33.33	100	83.33	78.95	16	2
119	19	1	100	100	100	33.33	100	50	73.68	12	2
120	28	1	100	100	100	33.33	100	16.67	63.16	11	2
121	19	2	66.67	66.67	50	66.67	50	50	57.89	12	2
122	20	1	66.67	66.67	50	66.67	100	33.33	57.89	16	2
123	20	1	66.67	66.67	100	66.67	100	66.67	73.68	14	2
124	19	2	66.67	33.33	0	33.33	100	50	47.37	14	2
125	19	1	66.67	0	100	100	100	50	63.16	17	2
126	20	2	0	66.67	50	66.67	0	33.33	36.84	12	2
127	19	2	0	66.67	50	33.33	100	50	47.37	10	2
128	18	2	66.67	66.67	50	66.67	100	50	63.16	10	2
129	19	2	66.67	0	50	0	100	33.33	36.84	13	2
130	19	2	33.33	66.67	100	0	0	0	26.32	14	2
131	18	2	100	66.67	100	100	100	50	78.95	15	2
132	19	2	33.33	100	100	33.33	100	50	63.16	18	2
133	18	2	33.33	33.33	0	33.33	100	16.67	31.58	18	2
134	19	1	33.33	0	50	33.33	50	0	21.05	10	2
135	18	1	66.67	100	100	66.67	100	0	57.89	11	2
136	18	2	66.67	66.67	100	33.33	50	66.67	63.16	12	2
137	20	2	66.67	100	100	33.33	100	66.67	73.68	13	2
138	21	1	66.67	100	100	66.67	100	33.33	68.42	15	2
139	18	2	33.33	66.67	100	66.67	100	50	63.16	16	2
140	18	1	100	100	50	33.33	100	66.67	73.68	17	2
141	22	1	66.67	66.67	50	66.67	50	16.67	47.37	16	2
142	18	2	0	66.67	50	66.67	50	16.67	36.84	17	2
143	18	2	66.67	100	0	66.67	50	66.67	63.16	15	2
144	20	2	33.33	66.67	0	66.67	100	33.33	47.37	12	2
145	19	2	66.67	33.33	50	66.67	50	33.33	47.37	13	2
146	19	1	33.33	33.33	100	66.67	100	50	57.89	10	2
147	19	2	33.33	66.67	50	33.33	100	66.67	57.89	12	2
148	18	1	66.67	33.33	50	33.33	100	50	52.63	14	2
149	22	2	33.33	100	100	66.67	100	50	68.42	13	2
150	20	2	33.33	66.67	100	66.67	50	0	42.11	15	3
151	23	2	100	100	50	100	100	66.67	84.21	16	3
152	20	2	100	100	50	66.67	100	33.33	68.42	13	3
153	18	1	66.67	100	0	66.67	100	66.67	68.42	12	3
154	19	1	100	66.67	50	66.67	100	16.67	57.89	16	3
155	20	2	33.33	66.67	100	33.33	100	33.33	52.63	14	3
156	20	2	66.67	33.33	50	66.67	100	83.33	68.42	14	3

157	20	2	0	33.33	100	0	100	50	42.11	13	3
158	21	1	100	100	100	100	100	83.33	94.74	13	3
159	20	2	100	66.67	50	33.33	100	33.33	57.89	12	3
160	19	2	66.67	66.67	50	66.67	100	66.67	68.42	12	3
161	21	2	0	66.67	50	66.67	50	50	47.37	14	3
162	23	2	66.67	66.67	0	33.33	100	16.67	42.11	11	3
163	21	2	33.33	33.33	0	33.33	0	83.33	42.11	9	3
164	25	2	100	66.67	0	33.33	100	66.67	63.16	10	3
165	20	1	100	66.67	100	33.33	100	0	52.63	10	3
166	24	1	33.33	33.33	0	0	100	0	21.05	11	3
167	21	1	100	100	100	66.67	100	83.33	89.47	12	3
168	20	2	66.67	33.33	50	0	0	50	36.84	8	3
169	20	1	100	100	100	66.67	100	66.67	84.21	10	3
170	20	1	100	66.67	100	33.33	50	66.67	68.42	15	3
171	19	2	33.33	66.67	0	33.33	100	50	47.37	11	3
172	18	2	0	66.67	50	33.33	50	33.33	36.84	9	3
173	19	1	33.33	100	100	100	100	66.67	78.95	11	3
174	19	2	100	66.67	50	0	50	83.33	63.16	9	3
175	22	2	33.33	100	50	66.67	100	66.67	68.42	9	3
176	20	1	100	66.67	50	66.67	100	50	68.42	13	3
177	21	1	66.67	66.67	100	66.67	100	83.33	78.95	11	1
178	22	1	66.67	66.67	0	100	100	100	78.95	14	1
179	20	1	0	100	0	66.67	100	83.33	63.16	13	1
180	19	1	33.33	66.67	50	100	100	83.33	73.68	12	1
181	19	1	33.33	66.67	100	100	50	83.33	73.68	12	1
182	18	1	100	66.67	100	100	0	100	84.21	11	1
183	18	1	66.67	33.33	100	66.67	100	83.33	73.68	11	1
184	19	1	100	33.33	100	66.67	100	100	84.21	12	1
185	19	1	100	66.67	50	66.67	50	83.33	73.68	13	1
186	18	1	0	66.67	50	33.33	50	100	57.89	13	1
187	19	1	66.67	100	50	33.33	100	83.33	73.68	12	1
188	18	1	66.67	66.67	50	66.67	100	83.33	73.68	13	1
189	18	1	66.67	100	100	100	50	100	89.47	12	1
190	18	1	33.33	100	100	66.67	50	100	78.95	13	1
191	21	1	33.33	33.33	0	66.67	0	66.67	42.11	7	1
192	21	1	66.67	66.67	100	66.67	0	33.33	52.63	18	1
193	19	2	66.67	66.67	0	33.33	0	66.67	47.37	17	1
194	20	1	66.67	33.33	50	33.33	0	83.33	52.63	14	1
195	20	1	100	66.67	50	33.33	50	83.33	68.42	13	1
196	18	1	33.33	100	50	100	100	100	84.21	9	1

197	20	2	66.67	33.33	0	66.67	50	66.67	52.63	11	1
198	20	1	33.33	66.67	0	33.33	0	83.33	47.37	18	1
199	20	1	66.67	100	0	66.67	50	50	57.89	10	1
200	19	1	33.33	66.67	100	0	50	83.33	57.89	11	1
201	18	1	66.67	0	50	100	0	83.33	57.89	10	1
202	20	1	66.67	66.67	100	33.33	0	83.33	63.16	7	1
203	20	2	100	66.67	0	0	100	50	52.63	14	1
204	21	1	33.33	66.67	50	66.67	100	50	57.89	9	1
205	21	2	66.67	66.67	0	33.33	100	50	52.63	11	1
206	20	2	66.67	100	100	66.67	100	100	89.47	13	1
207	19	2	66.67	66.67	100	66.67	50	50	63.16	12	1
208	20	2	33.33	33.33	0	66.67	100	66.67	52.63	13	1
209	24	1	0	100	100	66.67	100	50	63.16	13	1
210	20	2	100	0	100	33.33	50	16.67	42.11	16	1
211	23	2	33.33	100	0	33.33	100	100	68.42	10	1
212	21	2	66.67	66.67	100	66.67	100	50	68.42	12	1
213	20	2	33.33	0	0	66.67	100	83.33	52.63	10	1
214	19	2	100	100	100	100	100	100	100	10	1
215	20	2	33.33	100	0	66.67	100	83.33	68.42	17	1
216	22	1	100	100	100	66.67	100	33.33	73.68	15	1
217	23	2	66.67	66.67	50	66.67	0	83.33	63.16	15	1
218	22	2	66.67	33.33	50	33.33	100	33.33	47.37	15	1
219	19	2	33.33	33.33	50	33.33	0	66.67	42.11	12	2
220	20	2	66.67	33.33	0	0	100	50	42.11	9	2
221	20	2	33.33	100	50	100	100	66.67	73.68	17	2
222	20	2	33.33	33.33	100	66.67	50	33.33	47.37	8	2
223	20	2	100	66.67	0	66.67	50	83.33	68.42	11	2
224	20	2	100	100	50	66.67	100	83.33	84.21	12	2
225	21	2	66.67	0	50	66.67	50	16.67	36.84	15	2
226	21	2	33.33	33.33	50	0	0	33.33	26.32	16	2
227	21	1	33.33	33.33	0	66.67	50	50	42.11	15	2
228	21	2	66.67	33.33	0	0	100	50	42.11	8	2
229	21	2	66.67	0	0	33.33	100	100	57.89	11	2
230	21	2	33.33	66.67	0	33.33	0	66.67	42.11	11	2
231	21	2	33.33	66.67	50	100	50	83.33	68.42	13	2
232	22	1	66.67	66.67	100	100	100	83.33	84.21	16	2
233	22	1	66.67	66.67	50	33.33	50	33.33	47.37	10	2
234	22	1	0	33.33	50	33.33	0	0	15.79	10	2
235	22	2	33.33	33.33	0	66.67	0	33.33	31.58	8	2
236	22	2	100	66.67	50	33.33	50	100	73.68	10	2

237	22	2	66.67	0	50	33.33	100	83.33	57.89	11	2
238	23	2	33.33	33.33	0	0	50	83.33	42.11	7	2
239	23	2	66.67	33.33	0	66.67	0	83.33	52.63	8	2
240	23	2	66.67	33.33	0	0	0	0	15.79	12	2
241	23	2	66.67	100	50	66.67	100	83.33	78.95	10	2
242	24	2	100	33.33	0	0	50	50	42.11	14	2
243	25	2	33.33	33.33	0	0	0	0	10.53	9	2
244	23	2	66.67	0	100	66.67	100	100	73.68	13	2
245	21	2	66.67	66.67	50	66.67	100	50	63.16	12	2
246	23	2	100	100	100	66.67	100	50	78.95	14	2
247	21	2	0	33.33	50	66.67	100	50	47.37	14	2
248	21	2	66.67	33.33	50	66.67	100	50	57.89	11	2
249	21	2	66.67	66.67	0	33.33	100	100	68.42	10	2
250	20	2	33.33	66.67	100	66.67	100	50	63.16	10	2
251	19	2	66.67	66.67	50	33.33	100	100	73.68	16	3
252	20	2	66.67	100	100	100	100	50	78.95	15	3
253	19	2	33.33	100	0	66.67	100	100	73.68	15	3
254	22	2	0	33.33	0	66.67	50	83.33	47.37	15	3
255	21	2	66.67	100	0	100	100	50	68.42	16	3
256	22	1	33.33	100	50	100	100	83.33	78.95	12	3
257	19	2	100	100	50	100	50	66.67	78.95	10	1
258	18	1	100	100	100	100	100	83.33	94.74	14	1
259	20	2	33.33	66.67	0	66.67	100	16.67	42.11	14	1
260	23	1	33.33	33.33	50	33.33	50	100	57.89	15	1
261	20	2	66.67	33.33	0	33.33	0	0	21.05	15	1
262	20	1	100	66.67	100	66.67	50	83.33	78.95	13	1
263	18	2	33.33	100	100	100	100	83.33	84.21	15	1
264	20	1	66.67	0	0	0	0	0	10.53	13	1
265	21	2	66.67	33.33	100	33.33	100	100	73.68	13	1
266	17	1	100	66.67	50	33.33	100	83.33	73.68	13	1
267	21	2	100	33.33	0	100	100	100	78.95	12	1
268	20	1	33.33	33.33	100	33.33	50	16.67	36.84	12	1
269	18	1	33.33	0	0	0	0	0	5.26	12	1
270	22	1	33.33	66.67	0	66.67	100	83.33	63.16	14	1
271	22	1	66.67	33.33	100	66.67	100	50	63.16	13	1
272	19	1	100	100	100	100	100	100	100	17	1
273	19	2	66.67	0	50	66.67	100	83.33	63.16	14	1
274	19	2	66.67	100	50	66.67	0	33.33	52.63	14	1
275	21	2	100	66.67	50	33.33	100	50	63.16	14	1
276	21	1	33.33	0	50	33.33	100	50	42.11	9	1

277	23	2	100	66.67	100	33.33	100	83.33	78.95	9	1
278	20	2	33.33	0	50	66.67	100	83.33	57.89	11	1
279	21	2	100	100	50	33.33	100	100	84.21	13	1
280	26	2	33.33	66.67	50	66.67	100	83.33	68.42	17	1
281	21	2	100	100	100	0	100	33.33	63.16	14	1
282	20	2	100	100	50	33.33	100	50	68.42	15	1
283	20	1	66.67	100	50	66.67	100	83.33	78.95	8	1
284	23	2	100	33.33	0	66.67	50	100	68.42	17	1
285	20	2	66.67	33.33	50	33.33	100	100	68.42	12	1
286	22	1	66.67	100	50	66.67	100	50	68.42	11	1
287	22	2	66.67	100	0	0	100	83.33	63.16	11	2
288	20	2	66.67	66.67	100	33.33	100	83.33	73.68	10	2
289	25	1	33.33	0	50	66.67	100	100	63.16	10	2
290	18	2	100	100	100	100	100	100	100	12	2
291	20	1	100	66.67	100	33.33	100	100	84.21	12	2
292	18	2	100	66.67	100	33.33	100	83.33	78.95	11	2
293	19	1	100	66.67	50	66.67	50	100	78.95	9	2
294	21	2	33.33	33.33	50	66.67	100	83.33	63.16	13	2
295	18	2	66.67	100	0	66.67	100	83.33	73.68	13	2
296	20	2	66.67	100	50	33.33	100	83.33	73.68	16	2
297	23	2	33.33	66.67	50	0	100	66.67	52.63	16	2
298	22	2	33.33	66.67	50	33.33	100	33.33	47.37	15	2
299	20	2	66.67	100	100	33.33	100	100	84.21	12	2
300	19	2	33.33	66.67	0	100	50	83.33	63.16	11	2
301	20	2	100	33.33	100	66.67	100	83.33	78.95	11	2
302	19	2	66.67	33.33	0	66.67	50	50	47.37	17	2
303	20	2	100	100	50	66.67	100	100	89.47	17	2
304	19	2	33.33	66.67	100	66.67	100	100	78.95	13	2
305	20	1	100	100	100	66.67	100	83.33	89.47	11	2
306	22	2	66.67	66.67	50	33.33	100	83.33	68.42	18	2
307	18	2	33.33	66.67	100	33.33	100	100	73.68	16	2
308	19	2	100	66.67	100	33.33	50	100	78.95	13	2
309	20	2	33.33	66.67	50	66.67	100	16.67	47.37	11	2
310	19	2	33.33	33.33	0	66.67	0	83.33	47.37	15	2
311	19	2	33.33	33.33	50	33.33	50	16.67	31.58	16	2
312	18	2	33.33	100	100	66.67	100	100	84.21	18	2
313	21	1	66.67	0	0	33.33	50	100	52.63	15	2
314	19	2	66.67	66.67	0	66.67	100	83.33	68.42	14	2
315	19	2	66.67	33.33	50	33.33	50	16.67	36.84	10	2
316	18	2	0	66.67	0	66.67	100	50	47.37	12	2

317	18	2	66.67	66.67	100	33.33	100	100	78.95	12	2
318	18	1	66.67	66.67	50	0	100	83.33	63.16	12	2
319	23	1	33.33	0	50	66.67	0	50	36.84	8	2
320	22	1	66.67	0	0	66.67	0	33.33	31.58	13	2
321	18	1	0	33.33	50	100	50	66.67	52.63	15	2
322	20	2	33.33	100	0	33.33	50	50	47.37	15	2
323	20	2	66.67	66.67	50	33.33	50	100	68.42	11	2
324	18	2	100	33.33	100	100	100	83.33	84.21	11	2
325	19	2	33.33	66.67	0	33.33	100	100	63.16	12	2
326	22	2	0	0	50	66.67	0	66.67	36.84	15	3
327	19	2	66.67	66.67	100	66.67	100	83.33	78.95	15	3
328	19	2	33.33	33.33	0	66.67	100	50	47.37	14	3
329	20	1	100	100	50	66.67	100	100	89.47	11	3
330	19	2	66.67	66.67	50	33.33	50	100	68.42	12	3
331	22	1	100	100	50	100	100	66.67	84.21	14	3
332	23	1	100	66.67	0	66.67	100	100	78.95	14	3
333	25	2	66.67	100	0	33.33	100	83.33	68.42	14	3
334	20	2	100	66.67	50	66.67	100	83.33	78.95	15	3
335	23	2	0	66.67	50	33.33	50	66.67	47.37	15	3
336	20	2	100	100	50	100	50	66.67	78.95	13	3
337	22	2	100	100	100	66.67	100	100	94.74	12	3
338	24	2	100	100	0	66.67	100	83.33	78.95	12	3
339	20	2	66.67	66.67	0	33.33	100	66.67	57.89	14	3
340	22	1	100	66.67	50	66.67	100	83.33	78.95	13	3
341	20	2	66.67	66.67	100	66.67	100	83.33	78.95	13	3
342	22	2	66.67	66.67	50	66.67	100	83.33	73.68	13	3
343	21	2	33.33	33.33	100	0	100	83.33	57.89	11	3
344	22	2	66.67	66.67	50	66.67	100	83.33	73.68	9	3
345	21	2	100	33.33	100	33.33	100	83.33	73.68	9	3
346	20	1	66.67	33.33	100	66.67	100	66.67	68.42	9	3
347	21	1	100	66.67	50	100	100	83.33	84.21	15	3
348	22	1	66.67	66.67	50	66.67	100	100	78.95	10	3
349	19	2	100	100	100	100	0	100	89.47	13	3
350	20	2	100	66.67	0	0	100	83.33	63.16	15	3
351	19	2	66.67	0	100	33.33	50	100	63.16	10	3
352	19	2	66.67	33.33	50	66.67	50	100	68.42	10	3
353	20	2	66.67	33.33	0	66.67	50	83.33	57.89	10	3

Table D.6: Row data for the run of the Line-Edit study

	Experimental Variables										
	1	2	3	4	5	6	7	8	9	10	11
1	19	1	50	0	100	0	50	0	31.25	10	3
2	19	1	0	50	50	50	25	25	31.25	14	2
3	20	1	100	0	100	50	75	50	62.5	14	2
4	19	1	50	50	50	50	75	25	50	14	2
5	22	1	50	100	0	50	0	50	37.5	15	3
6	20	1	50	50	0	50	0	0	18.75	11	3
7	18	1	100	50	100	50	25	50	56.25	15	3
8	20	2	50	50	100	50	100	50	68.75	19	2
9	19	1	50	0	100	100	0	0	31.25	13	3
10	20	2	50	50	50	50	100	50	62.5	13	2
11	19	1	50	50	100	100	75	25	62.5	13	2
12	19	1	100	50	100	100	75	25	68.75	13	3
13	19	1	50	100	100	50	50	50	62.5	17	2
14	19	1	50	50	0	100	50	0	37.5	16	1
15	23	1	50	0	100	0	50	0	31.25	18	2
16	20	1	100	50	50	50	25	25	43.75	19	2
17	20	1	100	0	100	50	75	50	62.5	21	2
18	25	1	100	50	50	50	75	50	62.5	21	1
19	19	1	50	50	0	50	25	50	37.5	18	2
20	20	1	50	50	0	50	25	0	25	18	1
21	20	1	100	50	100	100	25	50	62.5	18	2
22	19	1	50	50	100	50	100	100	81.25	19	1
23	20	1	50	50	100	100	25	0	43.75	15	2
24	19	1	50	50	50	50	100	25	56.25	20	2
25	21	1	50	50	100	100	75	50	68.75	20	2
26	19	1	100	50	100	100	75	25	68.75	18	3
27	19	1	0	50	100	50	50	50	50	20	1
28	19	1	50	100	0	100	50	0	43.75	15	2
29	18	1	50	50	50	100	50	100	68.75	14	2
30	19	1	50	100	50	50	75	75	68.75	10	2
31	20	1	100	100	100	50	75	75	81.25	14	3
32	20	1	50	50	100	100	50	75	68.75	20	2
33	19	1	50	50	100	50	25	50	50	14	2
34	19	1	50	100	0	100	50	75	62.5	14	2
35	20	1	100	50	100	50	75	100	81.25	16	1
36	18	1	50	100	100	50	25	100	68.75	16	2

37	18	1	50	50	0	50	100	75	62.5	13	1
38	19	1	50	100	50	50	25	75	56.25	13	2
39	20	2	50	50	50	50	100	75	68.75	18	2
40	19	1	100	100	50	100	25	50	62.5	16	3
41	20	1	50	50	100	100	100	100	87.5	16	1
42	21	1	100	50	50	50	75	50	62.5	13	1
43	21	1	50	50	50	100	100	100	81.25	22	2
44	19	2	50	50	50	50	100	75	68.75	20	2
45	19	2	100	100	100	50	75	75	81.25	21	2
46	18	1	50	50	100	50	50	75	62.5	18	2
47	21	1	100	100	100	100	50	50	75	18	3
48	21	1	50	50	0	100	75	75	62.5	22	2
49	19	1	100	50	100	50	50	100	75	12	1
50	24	1	50	100	100	100	100	100	93.75	19	2
51	24	1	50	50	0	50	100	75	62.5	20	3
52	21	1	50	50	50	100	50	75	62.5	20	3
53	22	1	50	100	50	50	100	75	75	24	2
54	19	1	100	50	50	50	100	50	68.75	25	1
55	20	2	50	50	0	100	100	100	75	12	2