SCOOT : An Object-oriented Text Based Computer Programming Teaching Tool for Novices, with an Emphasis on Ease-of-use

By

Eugene McArdle

This thesis is submitted in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

School of Business

Discipline of IT James Cook University

Australia

31 October 2017

Statement of Access to Thesis

I, the undersigned, the author of this thesis, understand the James Cook University will make this thesis available for use within the University Library and, via the Australian Digital Theses network, for use elsewhere.

I understand that, as an unpublished work, a thesis has significant protection under the Copyright Act and;

I do not wish to place any restriction on access to this work.

.....

31 Oct 2017

Declaration On Ethics

The research presented and reported in this thesis was conducted in accordance with the National Health and Medical Research Council (NHMRC) National Statement on Ethical Conduct in Human Research, 2007. The proposed research study received human research ethics approval from the JCU Human Research Ethics Committee Approval Number H2911.

31 Oct 2017

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Eugene McArdle

31 October 2017

Contributions of Others

Nature of Assistance	Contribution	Names, Titles and Affiliations of Co-Contributors
Intellectual Support	Proposal Proofreading	Dr Jason Holdsworth, School of Business (IT), JCU Cairns
		Dr Phillip Musumeci, School of Engineering, JCU Cairns
	Advice with Data Analysis	Dr Siu-Man Carrie Lui, School of Business (IT), JCU Cairns
	Assistance with Survey Structure	Dr Katrina Lines, School of Psychology, JCU Cairns
Financial Support	Scholarship	APA Scholarship
	Grant	GRS Grant
Data Collection	Assisted during 2 test sessions	Dr Jason Holdsworth, School of Business (IT), JCU Cairns

Publications

Published

(1) McArdle, E., Holdsworth, J. & Lui, S.M. (2009). Usability Evaluation of SOLA: An Object-oriented Programming Environment for Children. In T. Bastiaens et al. (Eds.), Proceedings of World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2009 (pp. 2922-2929). Chesapeake, VA: AACE.

(2) McArdle, Eugene, Holdsworth, Jason, and Lee, Ickjai (2013) Assessing the Usability of Students Object-oriented Language with First-year IT Students: A Case Study. Proceedings of the Australasian Computer-Human Interaction Conference Australasian Computer-Human Interaction Conference. , 25-29, November 2013, Adelaide, Australia.

Acknowledgements

This dissertation would not have been possible without the help and support of more people than I can possibly remember here.

First thanks must go to my primary advisor, Dr Jason Holdsworth, for his consistent support and encouragement during the preparation of this dissertation, particularly when it seemed impossible to finish. Thanks also go to my other advisors, Dr Carrie Siu-Man Lui and Associate Professor Ickjai Lee. Carrie's help with the analysis and reporting of the statistics was essential, and Ickjai's help with getting the dissertation to a finalised state was invaluable.

I also wish to thank Dr Katrina Lines for her help in preparing the surveys used in this research.

Finally, this work would not have been completed with out the unending support and encouragement of my wife, Helen. Thank you for your incredible love and patience. It's over now :).

Abstract

IT educators have developed many tools and techniques to help novices learn to program computers, and yet learning to program is still hard. Some solutions attempt to remove the initial language barrier by replace code entry with another system, such as manipulating a graphical interface. There have been several attempts to create a visual language which allows 'novice' programmers to quickly learn programming skills including Tynker, Scratch and Alice. Case studies with Alice and other visual languages have shown a short learning curve, such that users are often able to create running programs, of varying degrees of complexity, within the first hour of exposure. Visual languages overcome initial barrier of code entry for novice programmers, and when combined with a good teaching environment and community they are a powerful tool for novices to learn - a fact which is supported by numerous case studies. Initially these visual languages were aimed at children, who would be more easily engaged by the visual interface, however in recent years visual languages (particularly Alice) have been implemented as part of introductory tertiary courses on programming. This lead to several case studies using Alice and other languages at a University level, where the focus is particularly on the transferability of the rapidly developed programming skills. Some studies indicate that many students are overwhelmed, confused and discouraged upon seeing their first 'real' programming language, whilst other studies indicate more successful transferability. The transferability of programming skills from a children's language has not always been seen as essential, but in light of the Queensland government's "Coding counts" initiative, and the inclusion of these languages in tertiary courses, their usefulness must be examined. This has coincided with a shift in how programming is taught at university, with object-oriented (OO) concepts now being introduced far earlier than in previous years. This dissertation argues that a text-based teaching tool designed to teach OO computer programming along with some foundational

logical thinking skills is needed. The Student-Centric Object-Oriented Teaching Tool (SCOOT) was designed as a stepping stone to full 'commercial' programming languages, with a focus on transferability of skills to those languages. SCOOT was developed by selecting appropriate educational principles and examining existing novice languages. Papert developed three principles for teaching, the "Continuity principle", the "Power principle" and the "Principle of Cultural Resonance". These design principles influence many areas of education, including programming education. Research into the design of novice programming languages also identified common mistakes made when designing a programming language. Several design principles to avoid these pitfalls have also been researched. Based on modern coding principles and Papert's educational principles, the design principles for SCOOT were developed. They are: Predictability (the interface must behave in logical ways), Familiarity (new concepts must build on old concepts) and Simplicity (the interface and commands must be as clear and simple as possible). SCOOT's interface was designed to be as minimalist as possible, in order to minimise extraneous cognitive load. SCOOT incorporates basic programming skills, including conditional and iterative statements, all within an OO framework, where all variables and methods are part of an object. Simple inheritance is also provided, as are dynamic object editing capabilities. The effectiveness of SCOOT was tested in a series of half-day case studies involving university students, where participants learned some OO programming theory and subsequently completed a series of programming tasks using the SCOOT environment. All participants were novices to computer programming either half way through their first semester, or with no experience at all. The tasks completed by the participants included code writing, reading and debugging components. The participants experiences with the SCOOT environment were examined, using a series of constructs measuring perceived ease-of-use, usefulness and enjoyment (based on the Technology Acceptance Model, TAM). A similar series of sessions were conducted using C++, the computer programming language used in the introductory programming course

at the University where this project was conducted. Steps were taken to ensure that the interface for both SCOOT and C++ were similar in visual style and complexity. The results showed significant reductions in the time to complete almost all tasks for SCOOT users, as well as significant increases in their performance in those tasks. SCOOT users also reported that they found SCOOT significantly easier to use than the users of C++. A correlation was also found between perceived ease-of-use and performance in most tasks. The simplicity of the SCOOT GUI did not introduce significant learning barriers in addition to the material. The degree to which users found the teaching tool easy to use is related to their performance using the principles being taught. A simplified teaching tool can teach basic programming principles without the constraints and overhead complexity found in typical introductory languages. A textual teaching tool overcomes the issue of transferability of skills.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Chapter Synopsis

Learning computer programming is not an easy thing to do [Powers et al., 2007, Chao, 2016]. Papert, an influential researcher in education and computing, identified several areas of expertise that would be improved and enhanced by learning programming [Feurzeig et al., 1970, Papert, 1980]. These include mathematics, logical analysis and thinking skills, problem solving skills, and design principles. Learning to program computers also demystifies the computer for end users. As computers become more prevalent in society it is necessary for people to have an elementary understanding of computers and computing. The majority of the skills learned in programming are not domain specific, and can be applied to many other areas of learning.

Part of the difficulty in learning computer programming stems from the fact that it is not a single skill. Rather it is the combination of several different skills including problem solving, algorithmic design, program design and program implementation [Garner, 2004]. Program implementation itself is a combination of knowledge of the programming language being used, and knowledge of the development software (referred to as an Integrated Development Environment, or IDE) being used. This dissertation focuses on the program implementation

and IDE aspects of programming. Issues related to learning problem solving, and algorithmic design skills can be more effectively addressed through teaching curricula.

Many universities currently use a commercial language, such as Java, C++ or C#, at an introductory level. Many educators, however, are unsatisfied with the use of a commercial language as a teaching tool [Ivanović and Pitnerl, 2011]. There is also some consensus on the need for a new introductory programming language [Black et al., 2010]. Other universities uses languages designed explicitly for teaching novices, such as Python, while others are using visual languages such as Alice.

When examining the difficulties faced when learning a programming language there are many potential hurdles which can impact a student's learning experience. In a comprehensive taxonomy of approaches to teaching programming [Kelleher and Pausch, 2005] identify several core problems which different languages attempt to solve. Briefly these problems include (taken from [Kelleher and Pausch, 2005]): Expressing Programs, Structuring Programs, and Understanding Program Execution. This dissertation focusses on the issue of expressing code, and several solutions are noted in their paper. Some languages attempt to simplify the entering of code, either by simplifying the language (Turing, GRAIL) or enforcing strict coding standards to avoid errors (GNOME, Cornell). Other languages find alternatives to typing programs, sometimes through graphical or physical interfaces (Alice 2). This work introduces a new object-oriented textual teaching tool designed for novices called SCOOT - Student-Centric Object-Oriented Teaching-tool. SCOOT is proposed as a bridge to more commercial programming languages. GRAIL, a procedural novice language developed in 1999, was created explicitly to be a stepping stone into other programming languages. The need for an OO language with the same design approach has recently been echoed in [Black et al., 2010]. This language needs to be text-based, in order to provide transferable skills, object-oriented, and designed as a gateway into other, more commercial programming

languages. It is proposed that SCOOT could meet this need by serving as a stepping-stone to a commercial grade OO programming language such as Java. While there is debate concerning the introduction of object-oriented (OO) programming principles [Braught and Wahls, 2008, Gal-Ezer et al., 2009, Goldwasser and Letscher, 2008, Lister et al., 2006a, Reges, 2006], this dissertation takes the view that introducing objects first enables students to fit procedural programming techniques into an existing paradigm, rather than needing to relearn core concepts. The majority of modern OO novice languages are designed for children, and as such may suffer from a lack of transferable skills due to over-visualisation.

1.2 Background and Motivation

Before a solution can be evaluated, the goals of the solution must be identified. This dissertation assumes that programming students need to learn skills which can be transferred to commercial programming languages. The transferability of skills learned through highly engaging visual environments is under question [Powers et al., 2007, Brown, 2008, Lorenzen and Sattar, 2008, Rader et al., 1997] and so these solutions may not solve the specific problem of turning novice programmers into skilled programmers.

Many languages have been developed either explicitly for novices, or with a focus on being easy to learn. Some common examples include Python, Turing, Scheme and (very early on) Pascal. Introductory programming courses at university typically use one of the more recent languages, or a commercial language such as C++ or Java. [Ivanović and Pitnerl, 2011] suggests that there is growing discontent within the education community with using commercial grade languages for this purpose. Educators are also unhappy using languages designed for novices which are inconsistent with common programming practices.

A panel discussion [Black et al., 2010] recently stressed the

need for a new programming language to be developed to teach object-oriented (OO) programming principles to novices. It should be noted that there is currently a discussion amongst IT educators regarding when OO principles should be introduced. Some educators [McIver and Conway, 1999, Ivanović and Pitnerl, 2011] argue that OO principles are built on more traditional procedural programming principles, and should therefore be introduced later in a programming course. Other educators [Braught and Wahls, 2008, Goldwasser and Letscher, 2008, Lorenzen and Sattar, 2008, Bloch, 2000, Bloch, 2003] argue that this simply establishes one programming paradigm (procedural), only to replace it with another (OO). This replacement causes the student to "relearn" much of what they have already learned, putting it into a new and broader context. There are several issues to consider here, but this dissertation agrees with the educators who are convinced that introducing objects first will ease the process of learning programming for novices at a university level.

GRAIL is an example of a language designed for novices, with similar goals to those identified in [Black et al., 2010], though from a procedural paradigm. The developers of GRAIL noted that the majority of novice languages either focus on engaging their users, and masking the complexities of programming, or eventually become too powerful for novice users to truly benefit from them [McIver and Conway, 1996]. To solve this problem McIver and Conway proposed GRAIL as a "zeroth" programming language [McIver and Conway, 1999]. GRAIL was designed explicitly for novice programmers, focusing on usability and transferability of skills. More importantly, it was not designed to be a full programming language in its own right - it would only exist as a stepping stone to other programming languages.

There are many languages designed to introduce programming concepts, with a wide range of interfaces and designs. Some of these languages are examined in Chapter 2.

1.3 Introducing SCOOT

SCOOT is a newly developed Object-oriented teaching tool, designed to enable novice programmers to learn programming concepts without the overheads of a commercial programming language, and without the potential for non-transferable skills taught by visual environments. When designing SCOOT the field of education was examined for inspiration, and in particular the work of Seymour Papert, developer of one of the first children's programming languages, LOGO. In his seminal work, Mindstorms [Papert, 1980], Papert identifies three core principles which must be applied to any method of teaching in order for it to succeed. In Papert's work these were the "Continuity Principle", the "Power Principle", and the "Principle of Cultural Resonance".

A concept satisfies the continuity principle if it builds on a student's current knowledge in predictable and expected ways, and does not contradict their current knowledge. It satisfies the power principle if a student is empowered by it, that is that they can see immediate rewards or benefits for learning the material. Finally it satisfies the principle of cultural resonance if it enables the student to move on to more advanced or complex applications of the material, and provides a larger social context for the material. These principles are discussed in more depth in Chapter 2.

In terms of a tool to assist with learning computer programming these principles mean that the tool must behave in ways which the users expect, provide immediate feedback and results, and enable the user to learn skills and principles which can later be transferred to a commercial grade programming language. These principles form the basis for the design principles of SCOOT.

In this research the need for a textual OO teaching tool designed for novices, which behaved in-line with these goals, was identified. No language or tool was found which satisfied all these criteria, which led to the development of SCOOT. SCOOT is a textual OO teaching tool, with a simple English-like syntax. It introduces users to a set of interpreted commands (providing immediate

feedback to users), and the syntax and programming constructs are designed to teach OO principles which are common, and easily transferred to other OO languages. These three features (English-like syntax, interpreted, and common code constructs and syntax) satisfy the Continuity Principle, the Power Principle and the Principle of Cultural Resonance respectively.

1.4 SCOOT - Aims and Objectives

1.4.1 Research Aims

The development and testing of SCOOT aims to determine the usefulness of a teaching tool, designed explicitly to teach object-oriented programming concepts to novices. It also aims to see if a textual teaching tool can reduce its complexity, while still teaching enough programming concepts to be useful. It is argued that an OO teaching tool designed for novices as a stepping stone to other languages will allow users to learn programming concepts and skills more easily than currently available options. The core hypothesis of this dissertation could simply be stated as:

To assess the usefulness, and impact on learning outcomes, of a text based teaching tool designed to teach novices the basics of object-oriented computer programming.

1.4.2 Research Objectives

The core hypothesis is broken down into six hypotheses which can be measured through testing. Due to teaching requirements, SCOOT was compared to the programming language used in the introductory programming course at the University where this project was conducted. At the time of testing (2009) this language was C++. The interface of SCOOT is minimalistic in design, as discussed in Chapter 2, and so a customisable, minimalistic IDE (Notepad ++) was used for testing purposes.

The first three look at programming performance:

- H1 - SCOOT is seen by novice programmers as easier to use than C++, programmed in a customised version of Notepad++.
- H2 - SCOOT is seen by novice programmers as more useful than C++, programmed in a customised version of Notepad++.
- H3 - SCOOT is seen by novice programmers as more enjoyable than C++, programmed in a customised version of Notepad++.

The remaining three look at user experience, since research indicates that user experience is critical for software tools, including educational environments [Calisir and Calisir, 2004, Trigwell and Prosser, 1991, Davis, 1989] :

- H4 - When given a simple programming task, users of SCOOT are able to write a correct solution more quickly than users of C++.
- H5 - When given a simple piece of code, users of SCOOT are able to correctly understand it more quickly than users of C++.
- H6 - When given a piece of code with errors in it, users of SCOOT are able to identify and correct the errors more quickly than users of C++.

SCOOT was developed to meet the need for a teaching tool designed to teach object oriented, textual computer programming to novices, and preliminary testing has yielded positive results in terms of user experience and demonstrated programming comprehension. The results of testing these hypotheses are shown in Table 1.1.

1.4.3 Contributions

The development of SCOOT incorporated knowledge from computer science and education, and focused on creating a text based tool for teaching OO principles to novices. This project offers the following contributions:

- SCOOT, a teaching tool which meets an identified need, has been developed and initial testing has shown improvements in all areas of

Table 1.1: Results Of Hypothesis Testing.

Hypothesis	Result
H1 - SCOOT is seen by novice programmers as easier to use than C++, programmed in a customised version of Notepad++	Supported
H2 - SCOOT is seen by novice programmers as more useful than C++, programmed in a customised version of Notepad++	Not Supported
H3 - SCOOT is seen by novice programmers as more enjoyable than C++, programmed in a customised version of Notepad++	Not Supported
H4 - When given a simple programming task, users of SCOOT are able to write a correct solution more quickly than users of C++	Supported
H5 - When given a simple piece of code, users of SCOOT are able to correctly understand it more quickly than users of C++	Supported
H6 - When given a piece of code with errors in it, users of SCOOT are able to identify and correct the errors more quickly than users of C++	Supported

programming skill, as well as reported improvements in user experience. Testing of SCOOT produced results which indicate a connection between user experience with a learning tool, and positive learning outcomes. There also seems to be support for the benefits of learning a 'zeroth' programming language for students of computer programming, and evidence that a text based language, with a simple and consistent syntax may avoid the pitfalls of a visual language.

- This dissertation examines a potential weakness in visual programming environments, and proposes a simple text-based tool as a stepping stone to more commercial programming languages.

1.4.4 Constraints and Assumptions

The restrictions encountered in the course of this study are as follows:

- Development assumption - While discussion continues regarding the introduction of objects before procedural concepts, it was decided to

proceed with an object-oriented tool for novices.

- Testing constraint - Due to teaching requirements, and limited resources at a regional university, SCOOT had to be compared to the programming language used in the introductory programming course at the University where this project was conducted. At the time of testing (2009) this language was C++, which is not an ideal language for comparison since SCOOT is specifically designed for teaching and C++ is a commercial grade language.

- Objects-First constraint - The first wave of testing was conducted with participants from the first year programming course, and due to restriction on their curriculum they had completed a semester of procedural programming before participating in the experiment. This means that the first wave of testing does not introduce objects first.

1.4.5 Dissertation Synopsis

Chapter Two of this dissertation is a review of existing literature in the field of novice computer programming languages, and teaching computer programming. This chapter examines core programming problems that have been identified, and their proposed solutions. It also looks at educational and programming concepts focussed on in this work. An overview of the objects first debate is also given and several key novice programming languages are examined and trends in language design are identified. Several novice languages are discussed, including Turing, Pascal, Scheme and Alice. Alice was originally designed as a children's language, and so has a highly visual interface. To aid the examination of visual languages, similar visual languages were examined for reference. Papert's educational principles (Continuity, Power and Cultural Resonance) were then used to examine these languages to see what worked and what did not. This chapter then identifies the need for SCOOT. The sum of what worked well in other languages was used as the basis for

SCOOT's command structure. Cognitive load theory (CLT) suggests that the effort required for any task is made up of intrinsic load (which cannot be avoided) and extraneous load, which is added by the tools or system being used [Chalmers, 2003, Feinberg and Murphy, 2000]. In order to control any additional load added by the user interface the interface of SCOOT was designed to be minimalistic. Finally common programming constructs are examined, and their importance or value for a novice bridging language is discussed. This provides the basis for deciding which language components would be supported by SCOOT and which would not.

Chapter Three then discusses the fundamentals of language design, lexers and parsers in order to discuss how SCOOT was implemented. Creating SCOOT required a new set of commands and syntax. The set of keywords and the syntax of commands are the parser and lexer rules of a language interpreter, respectively. A language generation tool (ANTLR [Parr and Quong, 1995]) was used to define these rules. A Java engine was then developed to take the processed input and produce relevant responses for the user. A detailed discussion of lexers, parsers and their implementation is given in this chapter. The components of SCOOT syntax and behaviours are given and discussed.

Chapter Four explains how SCOOT was tested, containing details of the design, recruitment and running of the test sessions. In order to test SCOOT it was compared to another language. At the time of testing the programming language taught in the first year subjects was C++, with objects being introduced in second semester. The timing of the testing allowed for the introduction of objects in the testing sessions before they were learned in the course, so that the performance students learning SCOOT for the first time could be compared to students who had a semester of experience with C++. During later rounds of testing C++ continued to be used for the sake of consistency. It is also known that the ability to write code is not necessarily an indicator of comprehension [Winslow, 1996] and so code reading and debugging tasks were incorporated into the testing to more accurately

measure understanding [Martin, 1996].

Finally, in Chapter Five the results of the testing are given, and a discussion of the implications is found in Chapter Six.

Chapter 2

Literature Review

2.1 Introduction

The key abilities of a computer programmer are problem solving skills, understanding syntactic rules with unfamiliar names and understanding how programs execute [Kelleher and Pausch, 2005]. Learning all these abilities simultaneously is why studying computer programming is challenging, and why many researchers since the 1960's have tried to simplify the process. In addition to the variety of abilities involved, the very act of learning a computer programming language is affected by at least three distinct components: the language itself, the IDE, and the support structures provided externally to the learning environment. IDEs vary in complexity from a simple text editor, to a full-scale commercial IDE like Visual Studio [Microsoft,]. Some languages also have a well developed support system, which provide extra resources and motivation for novice programmers. Scratch [Malan and Leitner, 2007] is one example of this, and there is an active on-line community for Scratch programmers to share their creations, and comment on others work. Note that this research project does not extend to include such support structures but there is a great deal of work being done in this area. [Diwan et al., 2004] introduces a tool to encourage first-year computer science students to

explore programs collaboratively, and [Ivanović and Pitnerl, 2011] compares two e-learning tools for teaching Java. Massive Online Open Courses have also been been introduced covering a variety of subjects. A critique of two such courses in Computer Science was recently conducted [Ben-Ari, 2011]. These studies focus on the impact of course structure and teaching tools on the effectiveness of novice programming courses. They are therefore outside the scope of this research, which focuses instead on languages and their respective interfaces. Other recent alternatives to traditional teaching methods include live-coding [Rubin, 2013] and interactive test-driven labs [Janzen et al., 2013]. Another promising area of research is the use of partially completed examples for students to finish, as implemented in CORT [Garner, 2004]. Work is also being done to encourage good program development processes [Caspersen and Kolling, 2009].

Next, an overview will be given of the educational principles used in this project to understand how programming is taught. Programming language concepts will then be introduced, along with a discussion of traditional and modern approaches to the order in which these concepts are introduced to students. This is followed by a brief overview of several key novice programming languages, developed between 1968 and 2007, and the chapter closes with a proposal for a new text-based, object-oriented programming teaching tool named SCOOT. The proposal includes conclusions developed in terms of the goals and design principles of the SCOOT commands, having examined several key trends in novice programming languages. From an educators perspective, the purpose of a programming teaching tool is to improve a student's understanding of core programming principles. If a new tool does not develop skills which are beneficial for learning future then it is not a useful teaching tool. The core focus of this project is in the acquisition of skills which can be transferred to commercial computer programming languages.

2.2 Principles of Learning

Educational Principles

Seymour Papert, a psychologist specialising in children and education, was involved in the development of LOGO, one of the first "educational" programming languages [Papert, 1980]. Papert developed a series of principles for teaching mathematics, which can be applied to any field of teaching for any age group. These principles are: ([Papert, 1980])

- Continuity Principle;
- Power Principle;
- Principle of Cultural Resonance.

Continuity Principle

The Continuity Principle states that the material being learned must be continuous with well-established personal knowledge, from which a student inherits a sense of value. For example, a student will find it more difficult to understand something with which they have no familiarity than to extend a familiar field of knowledge. In order to effectively teach computer programming it is necessary to build on a student's current understanding of related concepts, such as objects (from the real world), functions and variables (from mathematics), in order to minimise difficulties in learning, and reduce the learning curve.

This principle was applied to the design of SCOOT by ensuring that the syntax and semantics of SCOOT's command were consistent with the grammatical rules and principles of English. Phrases and keywords were chosen which had meanings consistent with the programming principles being used and learned. Additionally we made every effort to ensure that SCOOT command were internally consistent, and did not introduce confusion, either by having

similar concepts use different syntax, or by having dissimilar concepts using the same syntax.

Power Principle

In order to engage students, and convey a reason to learn, the material must provide them with immediate rewards of some kind. The knowledge must allow them to engage in activities which were previously impossible for them.

SCOOT is an interpreted language, and the current interface evaluates each line as it is entered by users (with some exceptions). This direct connection between entering a command and seeing the response was made in order to implement this principle, and provide immediate feedback to users. Languages like C++ or Java require several lines of code to be written before even the most basic program can be run. Within SCOOT a first time user can enter a single line of code and see an immediate response. The foundational goal of SCOOT is to teach OO principles, which in itself aligns with the Power Principle.

Principle of Cultural Resonance

Material that builds upon previous knowledge, and provides immediate rewards, must also be seen to be relevant and meaningful in future situations. The material must also be relevant and meaningful within a larger social context, that is it must have some use beyond the classroom. It is outside the scope of a piece of software to convey this degree of relevance, which can only be passed on through a human educator, or guide. In terms of a "teaching program" this area of relevance will be shown through the teaching style employed, and cannot be controlled in the software system.

This principle reinforces the premise that a tool which teaches programming skills must teach skills which can be transferred to a commercial grade programming language. If those skills are of no use when moving to more advanced languages then the tool has failed to meet the requirements of this principle. SCOOT was designed in order to apply this principle to a novice OO

programming tool

In the course of developing a teaching tool for programming, it is sensible to study other environments, and the teaching principles they employ. It is also necessary to study the learning patterns of new programmers, in order to identify particular hurdles and difficulties. This knowledge can then be used to support the design of SCOOT.

2.3 Programming Language Concepts

When designing a tool for teaching computer programming it is essential to determine which programming concepts to provide. Here some of the common concepts in programming are discussed, followed by a discussion of the order in which these concepts may be encountered in an introductory course.

2.3.1 Basic Concepts

Variables Central to any computer programming language is the concept of a variable, that is a named value which contains some information. Depending on the language the creation of a variable can vary considerably, as shown in Table 2.1. Some languages require the type of information to be explicitly defined by the programmer, while others will dynamically determine the type of data based on how it is used. Other languages only allow the creation of variables within an object, while others allow global variables.

As can be seen in Table 2.1 there are a variety of ways in which variables can be declared in different languages.

Collections Many programming languages allow for multiple pieces of data to be accessed through a single variable name, this is commonly called a collection. Collections range from simple array structures found in languages like C++, through to much more powerful feature-filled structures like lists in

Table 2.1: Variable Declarations in Various Languages

Language	Variable declaration	Notes
Visual Basic	dim num As Integer	Automatically assigns default value on creation, type is optional (a default type will be applied)
C++	int num = 0	Type is required, initial value is optional
Python	num = 0	No type required, Python interpreter identifies the best type to use based on value

Python. Collections are an essential component of any programming language which allows users to write programs beyond basic data manipulation. Some languages, including C++, Turing, and Pascal require that all members of a collection be of the same data type, while other languages like Python relax this requirement, allowing for more powerful collection structures. More recent languages such as Java and Python provide useful functions as part of their collections, and this is especially true of Python where common collection operations including sorting and searching have been catered for within the language.

Functions A function, also known as a procedure, is a segment of code which can be run (or called) at various times within a program without the need for repeating the code. Functions are an essential part of program design, and are common in almost all programming languages. Most languages allow for a function to produce a single piece of information (say a word or number) but some languages, including Python, allow for multiple pieces of data to be returned from a single function. Other languages such as C++ provide alternative mechanisms for returning multiple pieces of data, such as allowing inputs to be passed into the function through a memory reference. More object-oriented languages (such as Java) minimise the need to produce more than one piece of data through the data access provided through the object-oriented model.

It is also worth noting that many programming languages have a global scope allowing functions to affect each other, know as side-effects. Unintentional side-effects are the source of many errors and for this reason SCOOT, like Turing, does not permit them.

A method is a function declared within an object, while a function may exist without any object to contain it. Since methods are bound to the object itself, they are also known as type-bound procedures. In this dissertation the two terms will be used in this way, except when discussing Alice in Section 2.4.1 below, where the designers have their own definitions of those terms.

Classes Classes, and their instantiations (called Objects) are the core concept in any object-oriented programming language. A Class is essentially the blueprint, and an Object is a particular variable based on that blueprint. An object is a data structure, accessed through a single identifier, which may contain several pieces of information describing it (for example, a "Car" object may contain variables to store make, model and year) and a set of methods which control its behaviour (the colour of the car may be changed, or it could move to another location). Some languages provide support for objects while not being explicitly object-oriented, such as C++ or Python. Other languages enforce the object-oriented paradigm at every level - like Java, which only allows variables and functions to be declared within an object.

Conditionals Any program beyond a simple calculation will require the computer to be able to make decisions based on variables. This is a conditional statement, which means that certain lines of code will only be run if certain conditions are met. There are two standard forms of conditional statements: "if" and "if/else". An "if" statement, sometimes know as a "unary if", makes a decision about whether to run certain lines of code, otherwise it will skip those lines (Figure 2.1).

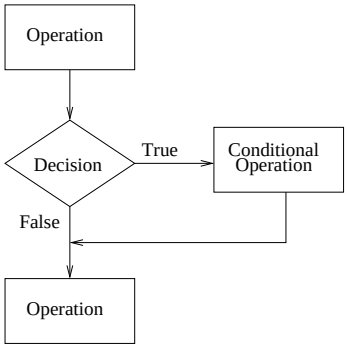


Figure 2.1: Unary If Statement.

An "if/else", or "binary if" statement, provides for two alternate sets of code, where only one will be run, and the other will be skipped (Figure 2.2).This may be due to the continuing difficulty Pascal has as a commercial grade language being used for educational purposes [Kruglyk and Lvov, 2012]

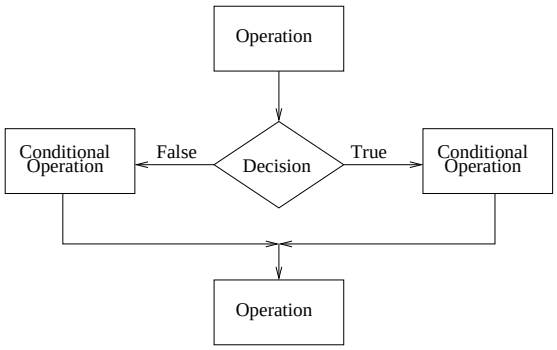


Figure 2.2: Binary If Statement.

A third alternative is to allow for multiple paths, either with several "if/else" structures or a simpler "switch" statement, supported by some languages. A switch statement allows a nested if statement with multiple paths to be written in a way which is more readable, as long as all conditions test the same variable. Depending on the language there may also by type or comparison restrictions on the use of switch statements as well.

Loops Another programming concept which requires a decision to be made is the loop, which allows for a set of commands to be repeatedly executed until

some condition is met. The standard loop structure is a "while" loop, which repeats while the looping condition is true. The test in a "while" loop is at the start of the loop, commonly known as a "pre-test loop", which is not ideal for certain situations. Loops which would perform better if the test comes at the end of the commands can be implemented using a "do-while" loop. This operates the same way as a "while" loop, but test the loop condition at the end, instead of before, known as a "post-test loop". A "do-while" loop will run at least once, no matter what, but that is not the case with a "while" loop. A common alternative to the "while" loop, specifically designed for counting loops, and accessing collections, is the "for" loop. Not all looping structures are available in all languages, though most modern commercial languages provide for the while and for loop structures. Some languages (such as Python) do not support a post-test loop at all, and so there is no do-while structure in those languages. Some Python texts suggest that a post-test loop can be implemented through the use of an infinite loop with a break statement within an if-statement at the end. Most professional programmers would not agree with the use of infinite loops with a conditional break statement being encouraged, especially in an introductory language.

2.3.2 Teaching Programming Concepts

The 1950's saw the rise of the imperative approach to problem solving (solving a problem with a set of sequential steps) and the 1960's saw the introduction of the object-oriented (OO) approach (solving the problem by creating a model). FORTRAN, in the mid 1950's was the first imperative language to gain widespread acceptance [Backus and Heising, 1964]. Due to hardware limitations at the time the imperative approach was successful, and with the introduction of subroutines (also called functions or procedures) languages which used this approach became known as procedural programming languages. As computer hardware capabilities increased the OO approach became feasible and languages such as C++ were developed, adding support for objects to a pre-existing

procedural language. Other languages, such as Java, were designed from scratch with the OO model in mind.

This brief history of the progression of the dominant programming approaches, from imperative, through procedural, to OO, reflects the model many programming courses followed until the last decade or so. Simple stand-alone commands were introduced, along with the base concept of variables. These statements would then begin to be chained together into a simple top-bottom algorithm of several steps. Conditional and iterative structures would then be added to the student's repertoire, allowing for branching programs which still followed a mostly top-bottom flow.

After these concepts were established functions would be added, allowing students to create their first truly procedural programs. At some point concepts such as containers may also be covered, depending on the language being taught and the centrality of containers to that language. Only when all these concepts have been covered would the concept of an object be introduced, as a kind of "data-type" which allowed programmers to have a container with several pieces of data, as well as functions which control the use of that data.

While this is a logical progression, there is a significant shift in the design approach to programming when moving from a procedural model to an OO one. This shift often causes serious issues for students, and is part of why debate continues regarding when objects should be introduced.

Recently there has been a push to introduce OO programming first [Cooper et al., 2003, Braught and Wahls, 2008, Goldwasser and Letscher, 2008, Lorenzen and Sattar, 2008], establishing objects as a framework for containing data and functions, which may contain sequential, conditional or iterative code. The logic behind this approach is that the shift from an OO approach to a procedural approach is less jarring to students, since object methods will contain procedural-style code. There is a great deal of debate in this area [Lister et al., 2006a, Reges, 2006], and as yet the research does not strongly point one way or the other.

The argument against introducing objects early is that the concept of objects is intrinsically difficult, and moving them earlier in a course simply moves the hurdle to the beginning of the process of learning programming, adding to an already difficult task [McIver and Conway, 1999]. Introducing it later allows for a stronger foundation to be laid, which the concept of objects can be built upon.

On the other hand, studies have shown that an appropriate teaching method allows for the introduction of OO concepts earlier than traditional teaching methods, as long as information being taught builds on currently known concepts [Goldwasser and Letscher, 2008, Lorenzen and Sattar, 2008, Bloch, 2003, Bloch, 2000]. This ties in with Papert's Continuity Principle. In this approach it is argued that the core difficulty of the transition to OO design is not intrinsic to the concept, but is a product of having taught procedural programming methods first.

It is our argument that the process of learning objects after procedural concepts involves having to unlearn certain concepts before the new material can be learned. The core elements of procedural programming are still needed, as object methods will still often include procedural elements, but the fundamental design of programs changes dramatically. This contradicts the Continuity Principle, and therefore does not line up with our design principles.

SCOOT is therefore designed to teach OO programming principles, and all variables and methods must be created with respect to an object.

2.4 A Brief History of Novice Languages

As can be seen in Figure 2.3 there have been several novice languages developed since the 1960's, and this is by no means an exhaustive treatment. This section discusses several novice languages, which are notable because of the lessons which can be learned from them.

2.4.1 Novice Languages

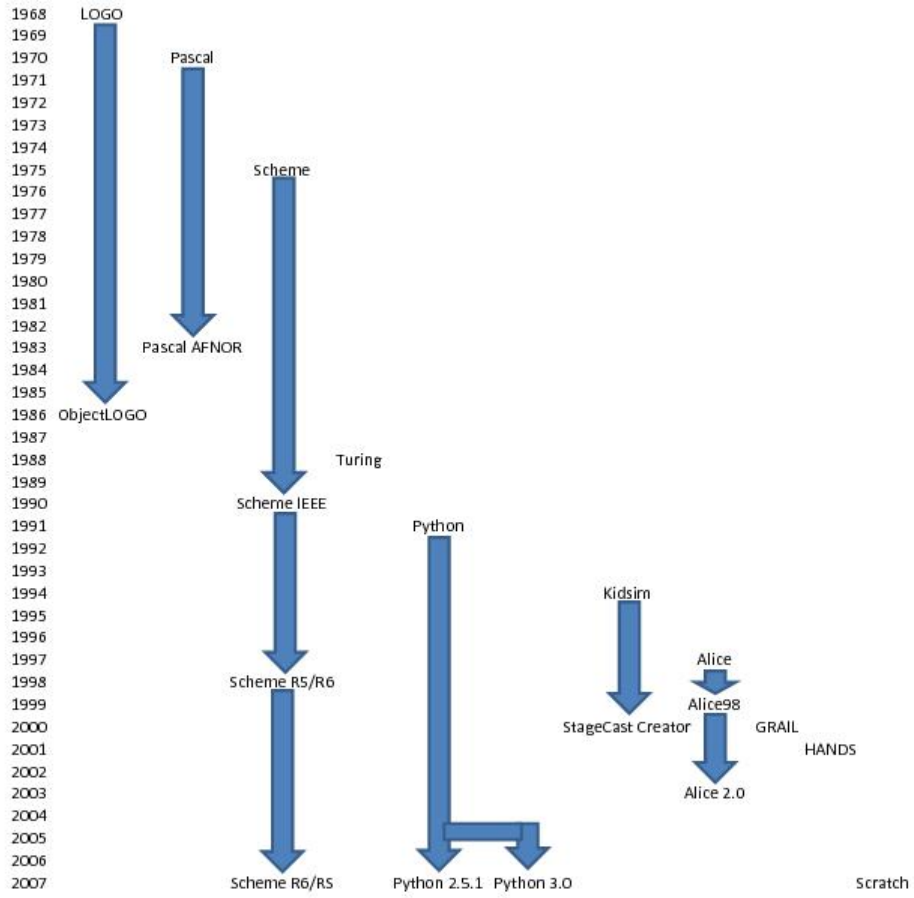


Figure 2.3: Selective History Of Novice Languages.

Within the realm of novice languages there are many different approaches, depending of what 'problem' of learning programming the designer is attempting to solve. An excellent survey of the main problems, and languages which aimed to solve them, is found in [Kelleher and Pausch, 2005]. The languages discussed in this section primarily focus on simplifying the entry of code, the problem this project is focussed on. The languages were chosen as representatives of the two major solutions to this problem, those solutions being:

- Text-based - Languages which keep a text-based interface, which has been designed to be easier to learn.

- Visual - Languages which replace the entry of code with a visual interface. These are often designed with children in mind, and the development of simple games or animations used as the motivation for learning. Within this category there are also two major sub-categories. They are:
  - Block - Different programming elements are represented by differently shaped blocks, indicating where they can be used and how elements can be combined.
  - Metaphor-based - Programs follow a particular design metaphor (such as a card game, or trains following a track), designed to be more concrete for novices than the abstract concept of programs running on a computer.

Text-Based Languages

LOGO LOGO is one of the most famous programming environments designed for children and currently exists in many different forms, some simply mimicking the original version and others being nothing more than the turtle graphics extension to LOGO which helped make it so popular [Layman and Hall, 1988]. It was created primarily as a tool to teach mathematical skills in the classroom, and not as a tool to teach programming [Blackwell and Bilotta, 2000]. However, the developers did not see programming as a means to an end (that is, teaching programming but wanting people to learn mathematics) but were instead motivating the learning of programming by proposing that it would help enhance a user's understanding of mathematical concepts [Howe, 1982, Feurzeig et al., 1970].

Since this was the core motivation, the language structures of LOGO, initially a primarily textual language, were not particularly english-like, but were instead more mathematical in structure. Case studies using LOGO in the classroom had generally positive results, and many educators found an

improvement in many skills (mathematics, logic, problem solving) in their students, arguably as a result of using LOGO [Howe, 1982].

However LOGO is not without its detractors, and its faults. In [Layman and Hall, 1988] it is pointed out that while LOGO may be a good tool for teaching certain related skills, there is no firm evidence that it succeeds in teaching transferable programming skills. Additionally other concerns have been raised with certain programming habits, which are discouraged or outright avoided in the commercial programming community, being acceptable or even forced in some versions of LOGO. Examples given in this critique include separation of a variable name from its value as shown below (in some versions, including Logotron), using numbers as variable names (impossible in any other language) and encouraging, sometimes requiring, the use of global variables.

```
MAKE "SIZE :SIZE + 1
```

The above example of LOGO code would be interpreted as "Make the *value* of the variable named SIZE equal to its previous value plus one". This adds a level of precision allowing users to differentiate between the *name* of a variable and its *value*, adding complexity to a concept which already causes young learners some difficulty [Layman and Hall, 1988].

More recent studies, for example [Blackwell and Bilotta, 2000] find problems with the syntax of LOGO. In a short term study comparing error rates of novices learning either LOGO or GRAIL, the LOGO group produced code containing significantly more errors. Not all recent studies are negative however. A recent study comparing StageCast Creator (SCC) to MicroWorlds Logo (MWL) [Louca, 2005] found that, while novice programmers found MWL harder to learn, they also demonstrated skill more akin to "real programming" than the SCC students. During the design process this included a focus on the program structure, rather than its features or interface. During coding the MWL students wrote the code they had planned, which was then debugged and fixed until it worked. The SCC students wrote code a line at a time, with each line being

tested and deleted if it did not work. SCC students, however, did not seem to understand their individual lines of code, and described their program by its interface and behaviour rather than by the code they had written.

Pascal Pascal is one of the first programming languages designed with a focus on teaching, and is based on the ALGOL 60 language. In addition to providing a simpler and more consistent syntax than ALGOL 60, it also provided better data structuring facilities.

In [Martin, 1996] teaching experiences with both Pascal and Turing were compared. Most student difficulties with Pascal were syntax related, either with common syntax errors being repeated, or with difficulty using the program headers, and begin/end blocks required by Pascal. A more commercial grade criticism of Pascal as a programming language is found in Kernighan [Laboratories and Kernighan, 1981]. He criticises the power and implementation of Pascal as a programming language, and is not concerned with its use in teaching. He raises several concerns, mostly related to the power and capability of Pascal, and conclude that Pascal is not "full" enough to be considered a "real" programming language.

Pascal would seem to have fallen into a trap which is common to many teaching languages, that is the desire to be a "real" language. Any good programming language will need to provide several core data and coding structures, and allow for a variety of interactions and combinations of those. This will inevitably lead to a reasonable degree of syntactic complexity in the language, which will make it less "friendly" to novice programmers. Pascal is discussed here as an early example of a mistake that continues to be made in many current novice programming languages, that of providing too much power and sacrificing simplicity.

Pascal, along with other novice languages, has attempted to find a middle ground between the two design goals ("commercial language" and "teaching language") and sadly seems to be too complicated to be the latter, and not

powerful enough to be the former. There have been several more recent languages in the Pascal family, including Oberon-2 and Delphi which focused on Object-oriented programming, but none are currently used significantly at the university level. This may be due to the continuing difficulty Pascal has as a commercial grade language being used for educational purposes [Kruglyk and Lvov, 2012] An Australian study conducted in 2002 showed only one university using a Pascal derivative (Delphi) with plans to stop using it that year [De Raadt et al., 2002].

This dissertation proposes the need for a teaching tool designed explicitly as a stepping stone to more commercial languages. This tool should have a limited set ofunderstandable commands, a simple syntax, and should not be targeted at commercial grade application development. While Pascal is no longer commonly used, or even discussed, in the field of novice languages, it is essential to recognise that using a commercial language to teach novices will encounter the same issues, without careful curriculum design (and possibly restricting information, which is not a good approach).

Scheme Scheme is a functional programming language which became popular as a teaching language in the 1980's and 90's. There have been several interfaces developed for it, and attempts to make it a more widely used teaching tool. Examples include the DrScheme environment [Findler et al., 2002] and the TeachScheme! Project [Felleisen et al.,] in the early 90's.

In a case study involving first year university students, Scheme was used as their first programming language, and the class then transitioned into Java half way through the first semester [Bloch, 2000]. The author initially had strong reservations about using an "introductory" language, since he felt that the transition would negate any benefit to the students. However he found that most students handled the transition better than he expected. This is a useful indication that an introductory programming language, designed to instil users with basic transferable programming skills, would be of general benefit.

In [McIver and Conway, 1996] it is noted that Scheme falls into the trap of oversimplification. In an attempt to ensure a simple syntax and program structure there is only one type of data within scheme and one real operation. The data type is a list, and that list can be evaluated. While this leads to a simple language, it is often difficult to put concrete real-world programming problems into such a framework. As such students often only see a specific set of list-oriented problems while learning to program with Scheme, which limits the transferability of the skills acquired.

Of greater concern for us, and in direct contradiction to Papert's Continuity Principle [Papert, 1980], is the syntax of Scheme. In an attempt to create an unambiguous syntax for Scheme, the developers have made the entire language pre-fix notation. As a simple example, the calculation to convert Fahrenheit to Celsius is shown below, in infix notation (which most people are familiar with):

```
(5/9) * (fahrenheit - 32)
```

in Scheme this would be written as:

```
(* / 5 9 ( - fahrenheit 32 ) )
```

This is an unusual syntax for most novice programmers, unless they have a high level of familiarity with mathematics. If a language is designed to teach transferable skills to novice programmers then, ideally, it should only teach those skills, and limit the amount of extraneous material. No mainstream commercial grade languages uses the prefix notation Scheme enforces, and students are unlikely to have had previously experience with it. It is recommended that an introductory language should focus on teaching those skills which will be of use in future programming languages and environments.

Turing Turing was designed as a "life-cycle" language, intended to be useful for teaching children to program, while still being powerful enough for serious software development [Holt et al., 1987, Holt and Cordy, 1988]. It is described as being like Pascal+, adding features such as dynamic arrays,

modules and variable length strings to the simple language structure of Pascal. It has been proposed as suitable as an introductory language for teaching object-oriented programming to undergraduates [Holt, 1994] and fared favourable in a comparative study comparing it to Pascal [Martin, 1996].

This study compared college students using Pascal to high-school students using Turing. Turing students had higher level of success in correcting syntax errors, but struggled more with syntax issues when writing their own code. In a critical paper identifying the common flaws of novice programming languages, Turing is identified as committing three of the seven [McIver and Conway, 1996]. These three are "Less is More", "Excessive Cleverness" and "Violation of Expectations".

Firstly, "Less is more". In order to limit confusion Turing enforces the rule that functions must have no side effects, which can be frustrating for seasoned programmers. Secondly, "Excessive Cleverness". Turing provides some features which allow for very clever manipulation of data sets and memory structures, but it is let down by a confusing syntax (made confusing as it tries to be overly simple for the sake of consistency). Thirdly "Violation of expectations". Turing uses the "%" symbol to indicate the start of a comment, when the same symbol has an established meaning for any student of mathematics.

Turing, like Pascal, seems to have become too advanced and powerful to be genuinely considered as an introductory teaching language, as is is too complex for many novices.

Python Developed in 1991 Python is an interpreted OO language designed to be easy to learn due to its simple syntax [Scripps and Sanner, 1999]. It provides all basic programming language constructs, including variables, functions, objects, loops, conditionals. It also provides for collections of data using either lists or dictionaries. Lists are similar to arrays in other languages, and dictionaries are essentially associative arrays.

Python has been used in first year university courses and several case studies

including [Miller and Ranum, 2005] have had a very positive experience using Python. They found that its simple interpreted syntax removed the unnecessary initial overhead of programming found in other languages like C++ or Java. It also allowed students to see a clear correlation between their commands and the computer's response, since it runs one line at a time.

Another study [Goldwasser and Letscher, 2008] has had similar results with Python as an introductory language, noting that while the transition to C++ was somewhat difficult, most students were able to transit to Java with relative simplicity. Similar results have been found using Python with high school students [Grandell et al., 2006].

Studies have shown that while the syntax and interface of Python allow for extremely fast pick-up of programming concepts, there are ways in which Python is not an ideal introductory language. Most of the difficulties encountered as educators relate to lists, and the incredible power they give to programmers. A list, as mentioned already, is similar to an array, but differs in several key ways. Firstly there is no requirement that all elements of a list be of the same data type. Secondly, lists in python are intrinsically dynamic in size. Finally many common list operations (sorting, finding the smallest element and many more) are provided by the language using a single command. There is no need for programmers to explicitly "type" a variable, Python is a dynamically typed language, and it is even possible to return several variables from a single function at once (Python implicitly creates a tuple and returns a single object from its functions).

All of these features are excellent in a commercial grade programming language, and programmers find them extremely useful. However, these features are not common to all programming languages, and the current result is that students transitioning into second semester using Java find common programming languages unnecessarily complex. They do not know why they have to specify the types of their variables, why a function cannot return more than one piece of information, or how to perform basic array operations such as

finding the smallest element, average or even sorting.

Python is a very easy language to learn, however some of its structures are too unique, and do not provide a full set of transferable skills to end users. The power of expression of the language is simply too high, and needs to be lowered in a novice language.

GRAIL GRAIL, developed by Linda McIver and Damian Conway, defines itself as a "zeroth" programming language [McIver and Conway, 1999]. It is a language designed explicitly as a introductory programming language, and is not intended to become a commercial grade language. For this reason it is made up of a very simple set of commands, with quite limited power, designed in accordance with four key design principles. They are:

- Syntactic predictability;
- Mimetic compatibility;
- Syntactic affordance, and;
- Minimalism.

The first principle is a refinement of the commonly desired principle of syntactic consistency. It is defined by Conway and McIver [McIver and Conway, 1999] to state that while similar concepts should share syntactic similarity, is is important that different concepts do not. A simple example is found in the programming language Pascal, where several blocks of code (loops, subroutines etc.) are bracketed by the keywords **begin** and **end**. These keywords are used in many situations, making it impossible for a reader to know (when reading only the **end** statement) which block is being closed. Compare this to the **function** and **end function** bracketing implemented in GRAIL, shown below.

The second principle ties neatly into Papert's Continuity Principle [Papert, 1980] , discussed in Section 2.2. Syntax used in a language will carry inherent meaning for most novice programmers, and developers of novice environments should build on those rather than try to change them. An example

given by McIver is the use of the asterisk (*) for multiplication, when almost any student of mathematics will automatically use the "x" character.

The principle of syntactic affordance encourages the use of syntax structures which encourage novices to write correct code. An example given in [McIver and Conway, 1999] compares an early function definition in GRAIL:

function identifier (paramlist *opt*) **returns** typename

statements

`return expression`

`end function`

with a redesigned version:

function identifier (paramlist *opt*) **returns** typename

statements

`end function returning expression`

This second form combines the "returning" of data with the end of a function, and removes the need for the student to remember to put the return statement last, since it now **must** be placed last. This new structure was also found to help novices separate the concepts of specification of a return (**returns**) with the invocation of the return (**return** in the first version, **returning** in the last).

The final principle defined is to ensure that the "zeroth" language stays exactly that. It should not offer powerful comprehensive commands, or multiple ways to perform a single operation (as is often the case for commercial grade languages such as C++). A minimal syntax with limited power will provide a language which encourages novice programmers in the learning of core programming principles while not overwhelming them with arcane syntax or formatting requirements.

In terms of language type, GRAIL is a text-based imperative language, which explicitly avoids object-oriented principles. In [McIver and Conway, 1999]

McIver argues that learning the object-oriented paradigm will always present a hurdle to any programmer, regardless of prior experience. She therefore opposes the introduction of objects first, since this approach simply moves the hurdle to the beginning of a novice programmers learning experience when "they are least equipped to deal with it".

This dissertation takes the view, supported by case studies [Cooper et al., 2003, Braught and Wahls, 2008, Goldwasser and Letscher, 2008, Lorenzen and Sattar, 2008, Bloch, 2000, Bloch, 2003], that learning any new concept, including object-oriented programming, will always present hurdles. However the evidence suggests that the greatest hurdle faced by programmers moving from a procedural language to an OO paradigm is not the OO paradigm itself, but is instead the unlearning of the prevailing procedural paradigm.

In fact it is proposed that the transition from the OO paradigm to a procedural one will present far fewer conceptual hurdles for novice programmers than the reverse transition (procedural to OO). This is because when a novice moves from OO to procedural paradigm they are taking part of what they already know (for example, their methods would have contained essentially procedural code) and making it more prevalent. Whereas a student moving from procedural to OO is taking an entirely new concept (an object, which contains both data and functions) and is required to re-frame their entire knowledge of programming so that it becomes a subsection of an entirely new paradigm. It could even be argued that in a procedural program the program itself can be viewed as an object, and the internal procedural code and subroutines are its methods.

For this reason, a tool is proposed which adopts the OO paradigm as the central concept while employing the design principles and interface of GRAIL. In such a system all variables and functions would be elements of an object, but the commands would be easier to learn than a standard commercial OO language.

Visual Languages

KidSim / Cocoa / StageCast Creator StageCast Creator (SCC) first began life as a graphical programming environment under the name "KidSim" around 1994. Under Apple Computers Advanced Technology Group it was later renamed Cocoa, and finally Creator after the project moved away from Apple Computing. The principal design idea of the original environment was to "move programming closer to human thought, rather than trying to move how we think closer to computers" [Smith et al., 1996]. While this is an extremely interesting idea, certainly from an educational perspective, it has not reached into the world of commercial programming with much success. Studies with children, however, have yielded very positive results with participants quickly picking up how to use the environment and beginning to create their own simulations [Cypher and Smith, 1995, Smith et al., 1994, Louca, 2004, Louca, 2005].

SCC combines two different alternatives to code entry, this has been identified as the core hurdle facing novice programmers. The two approaches are Programming By Demonstration (PBD) and Graphical Rewrite (GR) rules. GR rules allow the programmer to define a set of "before" and "after" rules. A sprite will check the list of rules and, if it finds that it matches a "before" rule, it will change its state to match the "after" rule. PBD allows the user to put the system into a "record" mode and perform a set of actions. Those actions will then be saved and can be re-executed later.

A difficulty encountered while researching SCC was the developer's use of the term programming. Along with making extraordinary claims about the success of the environment, for example, "Most kids learn to program within 30 minutes" [Smith et al., 2000, Smith et al., 1996] they also make some references to programming which do not line up with this work's definition of the term. At one point the developers describe a spreadsheet as "the most widely used programming technology" [Smith et al., 1994]. This indicates a significant disparity between this works definition of programming (the implementation of algorithms in order to control the computer) and that of the SCC developers.

An early trial of KidSim, taking place over the course of a year, found that most students were successful at creating an interactive project, and at describing what it did [Rader et al., 1997]. However when questioned about fixing, or changing, their programs, many students seemed to have a limited understanding of the programming concepts underpinning the programs they had written. Students showed a general reluctance to change anything, and could not seem to understand even simple changes (such as changing the direction a sprite moved). A common process which was observed in the development of their simulations was :

1. Decide on a simulation to make;
2. Make something similar;
3. Change their plan to match what they had made.

While this kind of development is not uncommon in any creative endeavour, it seemed to be born out of an inability to alter their programs, rather than a real "decision".

A more recent study, comparing SCC to Lego MicroWorlds indicated that the two different environments fostered two different programming styles [Louca, 2005]. The sessions were not guided, and students were free to write and develop code as they wished. The MicroWorlds students focused on program structure, thought about code constructs to use, naturally followed an iterative development process (Write, Test, Debug) and saw their code as a representation of phenomena, not as a simulation. SCC users on the other hand saw programming as "making games". They focused on the story behind the simulation, and instead of debugging code which did not work they simply deleted it and replaced it with something similar. This study noted that neither of these results is necessarily "bad", simply that educators should identify their desired outcomes and choose a programming environment accordingly. More research into how a novice programming language may impact, or foster, a particular development style is needed.

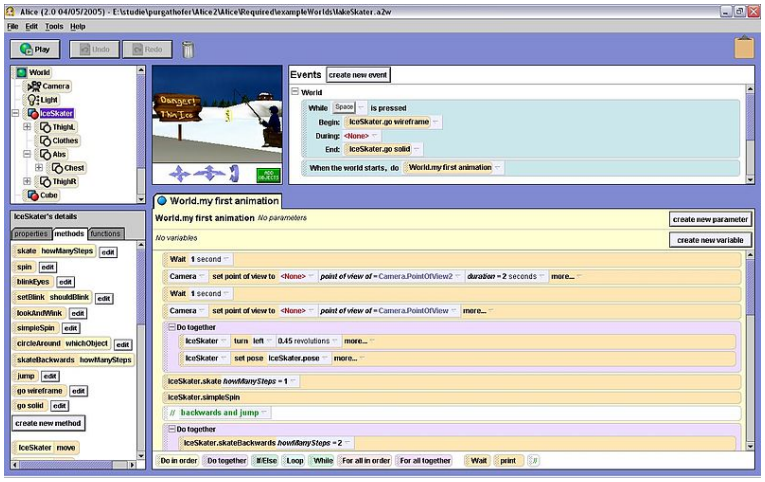


Figure 2.4: Alice 2.0 User Interface.

An even more recent study, which compares SCC, VB and HANDS, showed that while students saw it as easier than either of the other languages it was not their preferred choice for future use, nor was it seen as the most powerful [chuen Lin et al,]. While an equal number of students liked programming in SCC as in VB, it was still the second most favoured language (coming in second to VB). This would indicate that while SCC is certainly easier to use there is still a high level of interest in traditional, text-based programming languages.

Alice Alice is a visual programming language developed at Carnegie Mellon University, and has seen several revisions including Alice 98, Alice 2, Alice 2.2, Storytelling Alice and most recently Alice 3 [Mellon, , Brown, 2008, Kelleher and Pausch, 2005, Kelleher et al., 2007]. As can be seen in Figure 2.4 the programming environment provided by Alice has several main windows.

At the top, on the left hand side of the screen is a list of all the objects currently in the "world" the user is working on. Below that are listed the properties, methods and functions of the currently selected object. "Properties" are pieces of information about the object (for example, size, position), "methods" are things the object can do (for example, move to a new position or perform some animation) and functions allow programmers to get information

out of an object. The distinction between a method and a function, in Alice, is that a function makes the object do something (perhaps some calculations) and then returns a piece of information, whereas a methods never returns information. Commercial programming languages such as C++ and Java do not make such a distinction between value returning, and non-value returning functions, and this may be a source of confusion during a language transition. A personal experience with this was encountered recently when teaching an introductory course using VBA. VBA separates functions into 'subs' which do not return value, and 'functions' which do. Students displayed confusion about what could be done, and what could not, in a sub. In extreme cases students believed that a sub could not print to the screen, get data from the user, or change the value of any variable at all. They therefore couldn't understand why they existed, since they "couldn't do anything anyway".

In the centre of the screen, beginning at the top, there is a small window displaying graphically the contents of the world, and beside that is the event editor. Events are how Alice responds to user input, or triggers actions. An event could be the program starting ("When the world starts") or the user pressing a key or interacting with the mouse.. Below those two panels is the code editor, showing the method or function the user is currently editing. In Alice programs are built by dragging pre-programmed methods and functions from the list on the left side of the screen into the editor. Any options that need to be set (for example, time, direction, count) are modified through the use of drop-down menus.

Alice is widely hailed as a revolution in the teaching of programming [Adams, 2007, Dougherty, 2007, Lorenzen and Sattar, 2008, Rodger et al., 2009, Tedford, 2008, Dann et al., 2008], due to its easy to use and engaging interface. Storytelling Alice, in particular, is widely acclaimed for the success educators have had engaging students, particularly girls, in programming [Kelleher et al., 2007]. This is significant, since the ratio of girls to boys in introductory programming courses is on the decline. Some

researchers suggest that Alice is suitable for introductory programming courses at a tertiary level [Lorenzen and Sattar, 2008, Dougherty, 2007]. However there is some cause for concern over the effectiveness of Alice as an introductory programming language [Brown, 2008, Powers et al., 2007, Mullins et al., 2009].

It is currently unclear just how successful students are at transitioning from Alice to a more commercial programming language such as Java. One study at Bridgewater State College [Lorenzen and Sattar, 2008] briefly notes that "they transitioned well into Java". Minor frustrations with the text editor used in Java were noted, but otherwise no real indication was given of the success of this transition. On the other hand a similar study at Converse College [Brown, 2008] identified serious and significant issues faced by students on transitioning from Alice to Java. The two core issues were of syntax, that is the transition from a drag-and-drop environment to a text entering one, and of concepts, which stems from Alice's heavy use of 3D graphical objects in programming.

The issues of syntax are significant, and in this study had taken some time into the following course to resolve (and had limited the material which could be taught as a result). Another study [Powers et al., 2007] indicated that the weaker students met with success while using Alice, but when they were introduced to Java they mentioned that they "felt lied to". They felt that what they had done in Alice wasn't "real" programming, and lost a great deal of confidence as a result. The second issue identified in the Bridgewater study [Lorenzen and Sattar, 2008] is one of concept. In Alice all programs revolve around a graphically represented 3D world. This means that all the code, all the objects, and all the behaviours are easily visible, and therefore become conceptually concrete. That is to say, programmers can see the changes their code changes make, and do not have to conceptualise the ideas. However, anything beyond the most basic programs (or programs specifically written for a 3D system such as a game) require the programmer to be able to handle abstract concepts. This is a skill that must be learned by any programmer, and it does not seem to be when using Alice.

There is no doubt that Alice is widely successful, in particular at attracting people who may not have a prior interest in programming. However, despite its popularity, it is not the ultimate solution to all problems with teaching programming, and should not be treated as such. To quote Brown [Brown, 2008]:

> Alice, like so many CS ideas before it, is no silver bullet.

Alice has its place, and as an introductory programming environment to engage children it is extremely effective. Unfortunately, from a programming education perspective, there appear to be serious issues with the transferability of the skills learned when moving to a more commercial grade programming language. There are languages which attempt to bridge the gap between visual programming skills, such as those learned in Alice, and text based skills required in C++ or Java [Cheung et al., 2009]. However, this may simply add another step for programming educators who may be unwilling to introduce two introductory languages before moving onto a commercial language. Alice 3 was developed with a focus on transferability to other languages such as Java. How successful it will be remains to be seen, though a recent review of the literature suggests that using Alice is beneficial, while noting that there are 'few experimental results on the effectiveness of Alice programming language to introduce students in learning how to program" [Costa and Miranda, 2017].

Scratch Scratch is a visual programming environment, designed for younger children, with an impressive library of online support tools, including an active online community [Maloney et al.,]. It is designed to create media-rich interactive projects, and all Scratch projects can contain both media (images, sounds) and scripts. The development team had previously worked closely with the Lego Company on Lego Mindstorms, and were inspired by the way children would experiment with Lego blocks, seeing what fitted together and what didn't [Mitchel Resnick and Eric Rosenbaum, 2009]. This "block" metaphor forms the basis for the visual layout of Scratch, which heavily relies on shapes

Figure 2.5: Scratch Interface.

to indicate which commands can be combined. The block metaphor has become more popular in novice programming environments, especially those aimed at children, including Scratch Jr, Blockly and AppInventor

Users create commands in a drag-and-drop interface, which allows them to "snap-together" brightly coloured blocks to create their scripts. Each object is represented by a block of a particular shape, which clearly indicates to the user where that block can be used, as seen in Figure 2.5.

Scratch removes the need for users to fix syntax errors entirely, since like Cornell [Teitelbaum and Reps, 1981] users cannot create syntactically incorrect code [Kelleher and Pausch, 2005]. Another feature of Scratch, intended to minimise user overload, is a limited syntax set. To help implement this while still offering reasonable power, certain command sets are hidden from the user until the Scratch environment detects that they may be needed [Maloney et al.,] (for example, a set of instructions to control a LEGO WeDo robotics kit will be hidden until the appropriate USB device is connected to the computer). While all sprites in Scratch are technically objects (encapsulating attributes and behaviour) there are no explicit classes, and no capacity for inheritance. Scratch developers therefore refer to it as an object-based language, not an OO one [Maloney et al.,].

In an 18 month trial of Scratch at an after school centre fairly positive results were gained, though it must be noted that the sessions were mostly unguided and unstructured. Of the 536 projects created, 220 used looping structures, 111 used conditionals and 111 did not use any scripts at all. When asked about their experiences with Scratch most students described programming in Scratch as more like an arts subject (painting or sculpting) than anything scientific [Maloney et al., 2008]. While there is certainly an artistic/creative component to programming, most commercial grade languages require a more analytical and scientific approach. It would seem to indicate that Scratch, like Alice and StageCast Creator, is an excellent tool for introducing children to the idea of programming, or using computers creatively, but it may not be an ideal introduction for novices looking to learn professional programming.

Another trial study, proposing the introduction of Scratch as suitable for a first year university course, found that 76% of students felt that learning Scratch helped them understand what was happening when they transitioned to Java [Malan and Leitner, 2007]. Only 2% felt it had a negative impact and 16% felt it had no effect at all (though all 16% had prior programming experience). While this certainly builds the case for a "zeroth" programming language to be used as a stepping stone into higher level commercial programming languages at the university level, it is proposed that a tool designed to teach textual programming skills and code entry would be of more benefit than a highly graphical environment designed for much younger students. A more recent study [Sáez-Lòpez et al., 2016] showed the positive impact a language like Scratch can have on children when programming is incorporated across the school curriculum.

Greenfoot [Kölling, 2010] is another development designed as an alternate to Alice, which may well enable children familiar with Scratch to become more comfortable with code over pure visualisation. Greenfoot combines a visual interface (comparable with Alice) with Java. Novices are protected from the file structure of Java, while learning its commands and syntax. Positive results with

Greenfoot have been seen recently in a re-designed introductory programming course [Jonas, 2013].

While there are advances being made in the use of visual languages, the focus of this research is on the effect a textual language for novices has, rather than a tool where the entry of code is optional[Bloch, 2000].

HANDS HANDS is a graphical programming environment designed explicitly for children with a heavy focus on usability as its core design principle [Pane, 1998]. It uses a card-game metaphor in order to aid children in viewing a program running on the computer (a highly abstract concept) as a more concrete one. A "game" consists of a table which has several player around it each with their own hands of cards (active instructions) as well as a pile of other cards (inactive). Cards placed on the table are visible to all players, to demonstrate the concept of information sharing.

HANDS developer Robert Pane based the interface design on research conducted into psychology in order to make the environment as "usable" as possible. It was his contention [Pane and Myers, 2002, Pane, 1998] that a more usable system would aid in the learning process and improve novice programmer acceptance and learning.

Researchers in Taipei [chuen Lin et al,] ran a comparative study comparing HANDS to StageCast Creator and Visual Basic. The results of this study indicated that students saw HANDS as the least powerful language of the three, as well as the least popular and the least likely to be used in the future for other programming tasks. More research into novice programmer attitudes to environments is needed, but a trend appears to be surfacing in terms of attitudes to highly visual environments. If the end goal is to teach computer programming then it does not make sense to waste time teaching a language which many students will eventually feel is not real programming.

Recent Work Recent work in the field of programming education shows that there is an increasing awareness of a fundamental problem with the teaching

of OO principles. The problem is that OO principles are widely taught and yet no OO language exists which is an obvious first-choice for novices [Black et al., 2010]. In the past educators have used commercial languages such as C++ and Java, but these languages are often too big and powerful for novice users, and the education community is often unsatisfied with their use as a teaching tool [Ivanović and Pitnerl, 2011]. Simpler languages like Python are becoming popular choices, but some programming educators are wary of a language which does not conform to industry standards in some ways (including type definitions and list operations). Other attempts to solve the problem introduce new environments, designed to simplify the learning process. An example of this is Calico [?], a programming framework allowing the use of several languages, including Python, Ruby, Scheme and a new Scratch-inspired language called Jigsaw. Calico is designed as a transitional tool, so users could become confident building their programs in the visual Jigsaw language, and then transition into Python, Scheme or Ruby as desired. While the concept behind Calico is strongly in line with the goals of this project, given its focus on simplifying the transfer of programming skills, a textual language must still be introduced at some point [Black et al., 2010]. While many environments and languages exist to teach computer programming to novices, and especially children, there is very little rigorous measurement of their effectiveness [Stolee and Fristoe, 2011]. In a recent study of the Kodu Game Lab it was shown that many core programming concepts can be expressed in a visual language, and that many novice programmers used those concepts in their games. Earlier studies on visual languages, however, have indicated that use of a concept is not equivalent to comprehension of that concept [Rader et al., 1997]. Work is being done to improve the assessment of comprehension amongst novices [Kunkle and Allen, 2016]. Additional work has been done in examining the use of block languages (such as Scratch), and in possible ways to transition students from these block languages to more commercial text based languages. Two key approaches at the current time are to provide a bi-modal language (such

as Flip [Good and Howland, 2015, Howland and Good, 2015]) or to provide an intermediate language to bridge the two approaches (as suggested by [Kölling et al., 2015]).

2.4.2 Historical Trends

Several historical novice languages have been discussed above, ranging from the 1960's until now. Three trends in novice language design will now be discussed, and the above languages examined in terms of each of these three scales.

The three scales being examined are :

- Interface;
- Simplicity of Expression;
- Object-orientation

Alignment with Papert's Educational Principles These three trends are being examined in order to determine how the languages align with Papert's principles [Papert, 1980]. The purpose of SCOOT is to introduce novices to skills and concepts which can be transferred to other languages. This is in line with the Power Principle (enabling users to learn concepts that will enable them to do something they previously could not) and (in a limited sense) the Principle of Cultural Resonance, as the material fits into a bigger picture (though this lacks the social context, as discussed in Section 2.2). Additionally SCOOT was designed to be familiar, in terms of is language constructs. This was done specifically to align with Papert's Continuity Principle. It is out intent to discover if there is a need for an Object-oriented textual programming language with a high level of simplicity of expression.

Interfaces - Textual vs Visual Regarding the user interface of novice languages there is a very clear trend towards visual languages, with very few examples of recently developed textual novice languages (for example, GRAIL).

This trend away from textual interfaces is often motivated by the view that code entry is the biggest hurdle to programming, therefore it must be avoided. Alice, in its initial release, allowed for text-based code entry, though this was removed in later releases in favour of an entirely graphical system. This has been a source of several issues, to the point that Alice 3.0 allows for code entry again. The majority of novice languages, especially those developed for young users, have a highly graphical interface. Languages such as Scratch, HANDS and StageCast Creator are almost entirely graphical in terms of their interface, while Pascal, Turing, Scheme, Python and GRAIL are at the opposite end of the spectrum. LOGO, which is ironically most famous for its Turtle graphics, was initially an entirely text based language, with Turtle graphics being simply an extension.

In the past decade the field of novice programming languages has seen visual languages designed for children used at a university level [Brown, 2008, Mullins et al., 2009]. This means any examination of novice programming languages must examine the characteristics of children's languages as well, as there are now languages in both camps. Putting aside all differences of programming paradigm (imperative, procedural, OO), there seem to be two distinct approaches to designing novice programming languages. On one side there are those who argue that the purpose of teaching computer programming should be for its own sake. That what matters most (in terms of the language or system being designed) is that users develop an understanding and appreciation for programming itself. This may be applied to certain tasks (for example simulations, mathematics or games) but at their core these systems try to impart some skills which will aid users in learning a more commercial language. The other side argues that programming should only be seen as a means to an end. It is used to produce simulations, demonstrate physics principles, or for entertainment or educational purposes. These systems make concessions in terms of language structure, interface and so on that the other set of systems would not be willing to make, so that the system is more accessible, usable and

fun.

How designers see the purpose of teaching programming (for its own sake, or a means to an end) will directly impact the core problems that systems will attempt to solve, and their approach in doing so. It is necessary to be aware of the different approaches when examining the range of solutions, so that a proper perspective of the field may be achieved. While the motivation behind the development of a programming language is often hard to discern, it is an important aspect to consider when discussion their use for teaching novices. Some languages such as SCC, Scratch and recent versions of Alice focus on the idea of engaging children in computer use, and on using computers to create multimedia projects. Other languages, including Turing, Pascal, Python and GRAIL, have been designed explicitly to teach the skill of programming. This does not mean that an engaging language such as Scratch cannot teach programming skills. However it does mean that certain core programming practices (including code entry or even seeing code) may be removed, as is the case with HANDS, SCC and several versions of Alice.

In terms of SCOOT's design it was essential that the focus be on the programming concepts being learned, for this reason we conclude that there is a need for SCOOT to be a textual language, allowing novices to learn code entry the way it is commonly done, in line with both the Power Principle and the Principle of Cultural Resonance [Papert, 1980].

Other than the actual coding interface, there is another interface that must be considered, that being the IDE used by the programmer. Some languages (Alice, Scratch) tightly link the interface to the language, while other languages (C++, Java, Python) have more than one possible interface, ranging from the simple (IDLE (Figure 2.6), Notepad++ (Figure 2.7)) to the complex (NetBeans (Figure 2.8), Visual Studio (Figure 2.9)).

Cognitive Load Theory (CLT) has been used recently in evaluating several different computer learning environments, including mobile app development [Raina Mason et al., 2015], VR simulators for surgical training

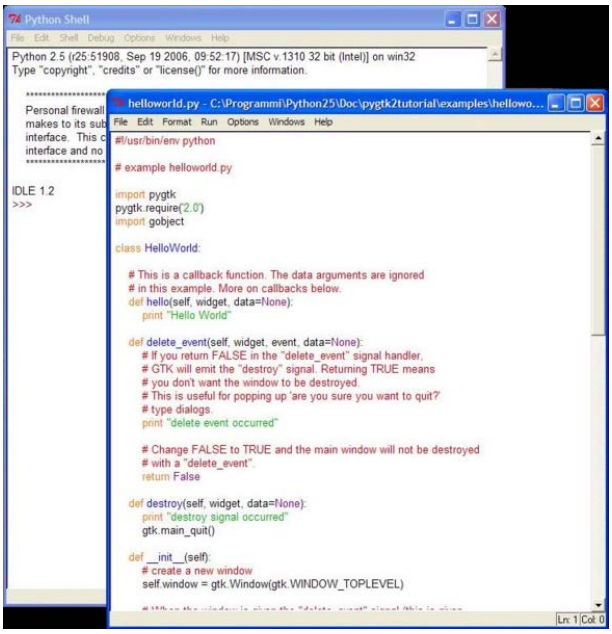


Figure 2.6: IDLE (Basic Python IDE).

[Andersen et al., 2016], as well as novice programming environments [Uysal, 2016, Moons and De Backer, 2013]. CLT, as it applies to any tool used to perform a task, defines two kinds of "load" applied to any user while using the tool. They are "intrinsic load", that load which is simply part of the task being performed, and "extraneous load", that load which is additionally created by the use of the tool [Feinberg and Murphy, 2000]. The

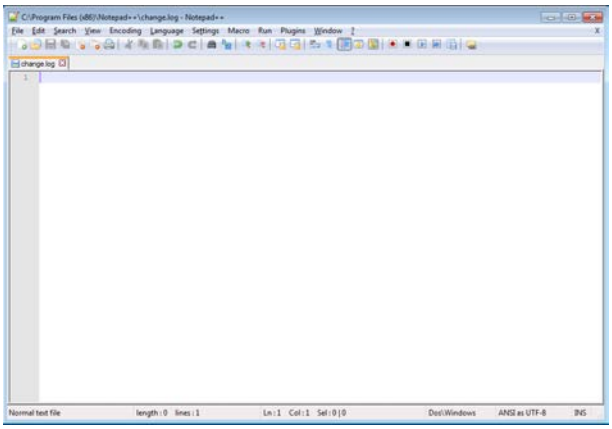


Figure 2.7: Notepad++ (Minimalist Editor, Supports C++ et al).

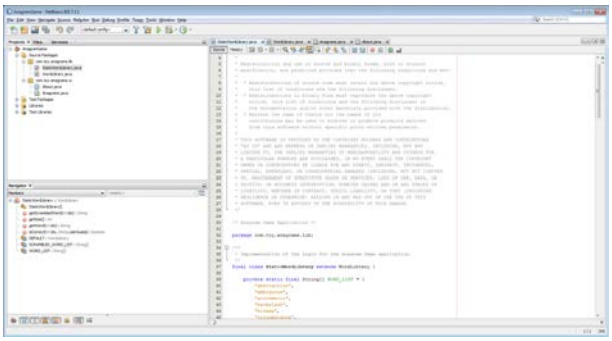


Figure 2.8: NetBeans (Popular Java IDE).

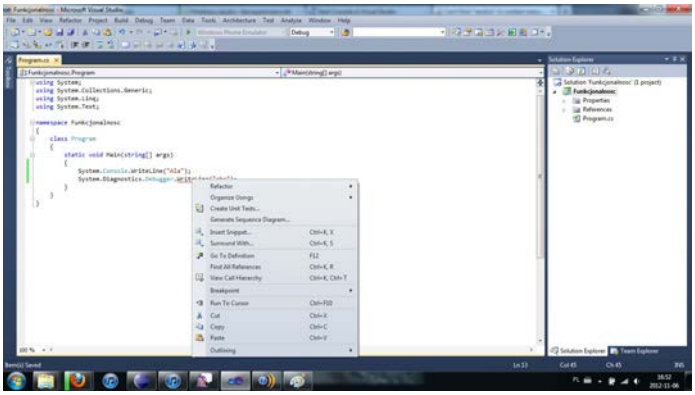


Figure 2.9: Visual Studio 2010.

greater the overall load the more difficult it is to perform the task. Note that in many cases it is difficult to clearly determine the line between intrinsic and extraneous load, especially when dealing with a programming language and environment. When examining teaching tools an additional kind of load must be considered, called 'germane' load [Andersen et al., 2016]. Germane load refers to the cognitive load of learning as a mental process. In the case of an introductory educational programming tool the intrinsic load is based on the complexity of the concepts being learned, the germane load is based on the process of learning itself, and any cognitive load caused by the use of the interface is categorised as extraneous.

It is also necessary to distinguish the process of learning programming from the process of programming itself. Once a programmer has familiarity with

language constructs and syntax many of the features of a high-end IDE become valuable tools, and in many ways decrease cognitive load because they can focus on solving the problem rather than smaller details of syntax and structure. However, learning requires a focus on those lower level concepts, and so the features of many modern IDE's are not as helpful, and should therefore not be considered as benefits of IDE's [Uysal, 2016]. Introducing a novice to a complex concept (programming) and a complex environment/tool to do it could be seen as a violation of the Continuity Principle. As programmers become more capable the features of a more powerful IDE become understandable, predictable and even necessary, however the best was to abide by the Continuity Principle is to have a cognitively light interface, allowing the novice to focus on the task at hand (learning to program)

In terms of a software interface to teach computer programming, it is essential to minimise the extraneous load in order to allow the user to focus on the learning at hand. A recent study [Uysal, 2016]evaluates several novice environments using measure of cognitive load, and an evaluation of SCOOT will be conducted with this approach in the future.

As has already been discussed, the act of learning programming is quite difficult already, that is it has a relatively high intrinsic and germane load [Uysal, 2016]. It was therefore decided that the interface for SCOOT be as simple as possible. As has been shown in Figure 2.6 the IDLE interface for Python is extremely simple and minimalistic. It was also desirable to more clearly define the difference between user-generated inputs and SCOOT-generated outputs. For this reason there is a command input field, as well as an output field. Additionally there is a history window showing all commands which have been entered into SCOOT, and a "Compute" button used to execute input commands. Layout of the SCOOT interface is shown in Figure 2.10.

Figure 2.10: SCOOT Interface.

Simplicity of Expression The degree of fine control to give to a programmer is often balanced by the availability of short, powerful commands. As an example the code to manually sort a collection of integers would be several lines in a typical language (approx. 20 depending on the algorithm). The same function could be carried out with a built in function of Python, or by importing appropriate libraries in other languages such as C++ or Java. Simplicity of expression, that is keeping commands simple, is a tricky question in novice language design. One on hand, novices can become overwhelmed with complex code, thus more powerful commands (reducing their simplicity) seems ideal. However it is our goal as educators to train programmers, not students who simply use pre-defined functionality without any real knowledge of its function. A balance is therefore required, allowing for simple use of basic data structures, but without excessive coding overheads. In alignment with the Continuity Principle the ideal novice language does not hide fundamental concepts and principles from the user, but does shield them from the low-level complexities of computer operations (such as memory allocations, hardware devices or output streams).

Most languages discussed above provide a reasonable level of control, without overloading novices too heavily. Languages which have attempted to become more powerful, such as Pascal and Turing, often create additional syntactic complexity which becomes a stumbling block for novices. In a similar way

Scheme, through its use of prefix notation, is immediately more complex to a novice who has never seen that particular form of expression. Languages such as Alice, Scratch and GRAIL provide a reasonable balance between the two extremes, while Python provides some functionality which is almost too powerful for novices. It should be noted that any commercial language will always include features not intended for novices, and these are typically not addressed in introductory curricula. However, this particular concern with Python is based on the fact that more powerful constructs have replaced simpler ones, rather than simply being an alternative. This is unlike many commercial grade languages such as Java, where more powerful data structures are available in extended libraries, but fundamental structures are still present.

Object-Oriented As discussed in Section 3.2.2 it is our view that teaching objects after procedural concepts violates the Continuity Principle unnecessarily. In order to align with this principle we argue that a novice language which focusses on object-oriented principles is needed. Several novice languages discussed above include objects, or are entirely object based. As we look further back in time the languages tend to be less and less object based, which is reasonable since the idea of teaching objects first is relatively new. Pascal, as has been mentioned, has several object-oriented derivatives, and Alice, Scratch and StageCast Creator are all highly object oriented. Scheme does not natively support objects, but there are many implementations which allow for their use, and GRAIL was explicitly designed to avoid objects altogether.

2.5 Proposal for a new teaching tool

As can be seen in Figure 2.11 there is no one language which meets all the criteria of object-oriented, textual and with a high simplicity of expression. The development of a new prototype teaching tool for computer programming called SCOOT (The Student-Centric Object-Oriented Teaching Tool) is proposed. SCOOT would have a limited set of instructions, simple english-like syntax,

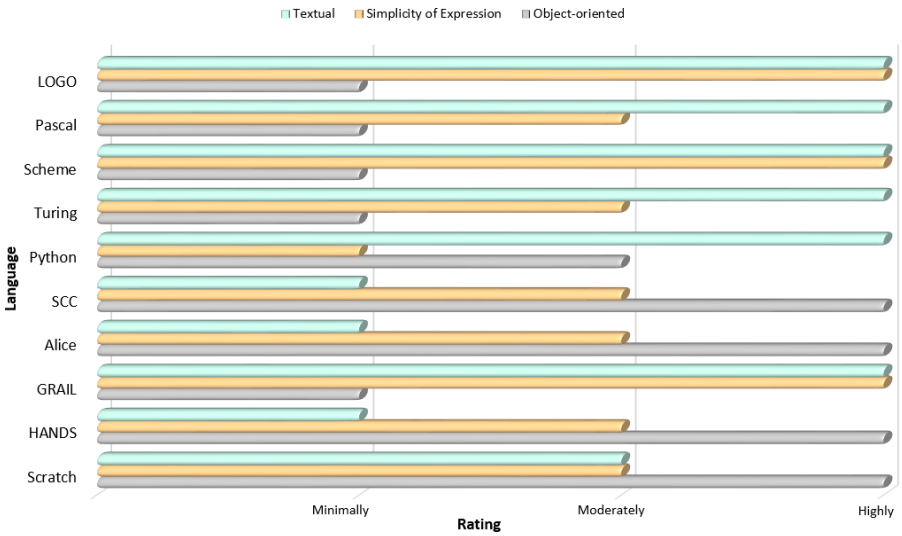


Figure 2.11: Summary of Language Evaluation

using text-based code entry and a minimal user interface, designed explicitly to teach object-oriented programming skills to novices. The syntax of SCOOT will be designed along the same lines as a zeroth programming language, that is a stepping stone into other languages, and this tool is not intended to be used for any purpose beyond introducing novices to programming. In this way SCOOT will not fall into the same trap as Pascal and Turing.

The most commonly addressed problem in teaching a computer programming language is the complexity of the language itself. Novice programmers may know what they want the computer to do, but are unsure of how to tell the computer how to do it. Moreover, most novice programmers don't know how to fix syntax errors or logic errors that emerge from their programs. There is a range of solutions to this problem [Kelleher and Pausch, 2005], such as design principles that keep the language simple, clear and consistent, or remove the textual language altogether, as is the case with visual languages and PBD systems.

Research into visual and PBD languages suggests that this approach may remove the initial hurdle of language complexity, only to introduce a larger hurdle later on when novices attempt to learn another, more commercial

programming language, such as Java or C++ [Powers et al., 2007, Brown, 2008, Rader et al., 1997]. Since this project is focused on teaching transferable programming skills, it was decided that a visual approach, or any approach which removes the entering of code, was inappropriate. It was therefore decided to identify a set of design principles which would guide the development of a new language, in order to minimise the learning curve.

As discussed above, GRAIL [McIver and Conway, 1999] developers Linda McIver and Damian Conway defined a set of language design principles which reflect the goals of this study, and a modified version of their principles were followed for the design of SCOOT. These principles were chosen because of their close alignment with Papert's principles [Papert, 1980] and due to the developers focus on a zeroth programming language. The modified principles used in the development of SCOOT are:

- Predictability;
- Familiarity;
- Simplicity.

Predictability Predictability means that similar concepts share a similar syntax, and that separate concepts do not share similar syntax. SCOOT attempts to ensure that similar structures follow similar patterns, while staying unique. For example, both conditional and iterative blocks begin with a test, followed by the beginning of the block (either an if or a while instruction) and end with a unique command, terminating either the conditional or iterative block.

Conversely the instructions for adding either an attribute or a method to an object are distinct, attempting to highlight to the user that an attribute is a piece of information the object **has**, while a method is a behaviour the object **performs**.

Like GRAIL, SCOOT's syntax and semantics are isomorphic, indicating a one to one correspondence between form and meaning. In programming syntax this means that each command has a single conceptual meaning, and the corollary (that different commands have different meanings) is also true . In SCOOT this means that each syntactic component has a single meaning and a single construct has a single associated syntax. More powerful languages [Holt et al., 1987] often offer several ways to do the same task, which are extremely useful for a professional programmer. However these options add confusion to an already difficult task for novice programmers. By limiting the flexibility of the commands novices can gain familiarity and confidence with core programming concepts, before being exposed to an otherwise overwhelming language. The uses of isomorphism in novice and children's environments is supported elsewhere [Weintrop and Wilensky, , Haiman, 1985, Sapounidis et al., 2015].

Familiarity This is the Continuity Principle (Section 2.2) in action. The syntax of SCOOT was designed explicitly to have an English-like structure, with each command being similar to an English sentence. In the same way, English commands are used for arithmetic expressions, thus avoiding the difficulty of introducing novice programmers to the multiplication and division symbols used in most languages ("*" and "/" respectively).

Simplicity SCOOT's purpose is to introduce OO programming concepts to novices. It is not going to be the new C++, and it is not designed to be used as a long term teaching tool. Its purpose is solely to serve as a simple, limited introduction to core programming concepts. There is a great deal which modern languages can do that SCOOT cannot. This is a deliberate design decision, in order to limit the complexity of SCOOT commands. To simplify the command list of SCOOT, its power has also been limited. This simplicity of design affects the control and data structures available, the language semantics and syntax, and the interface design.

Programming Concepts As mentioned above a teaching tool with a simple syntax, and limited power and functionality is proposed. While developing SCOOT it was decided to remove the need to define the data type from variable creation (as in Python), and only to allow variables to be created within an object (as in Java). This was done to simplify the variable creation process, while reinforcing the centrality of objects as the heart of SCOOT. When considering collections, it was decided that since the purpose of SCOOT is not to be a fully-featured language, but to provide an introduction to basic programming constructs, there would be no support for collections of any kind in SCOOT.

This means that there are no arrays, or array-like structures, of any kind. While these are an essential tool in many programming problems, the purpose of SCOOT is to be an introductory tool to teach OO concepts, and it is expected that any user will move to a more advanced language before using structures like arrays.

All object behaviours are implemented as functions and like variables, functions can only be created within an object, enforcing the OO basis of the tool. Support for input and output arguments has not been implemented in the current version of SCOOT. Most languages separate the definition of an object from its instantiation (a "class" is the template for an object, an "instance" of a class is the object itself). It was decided for the scope of SCOOT not to separate those two aspects, but instead to combine definition and instantiation (as in JavaScript), to remove initial confusion for the users. Because of the interpreted nature of SCOOT, it was also decided to allow on-the-fly modifications to an object's structure (behaviour and methods). This is not common in modern programming languages but seemed appropriate given the interpreted nature of the tool.

When looking at conditional statements it was decided to keep the concept as simple as possible in SCOOT, and so only a simple "if" statement is available. Since collections do not exist in SCOOT, and since simplicity was a core design principle, only a single pre-test "while" loop has been implemented. There are

no alternative looping structures in SCOOT.

2.6 Conclusions

This dissertation is based on the premise that the goal of teaching computer programming is that students are able to program computers using textual input. This would also result in an easier learning curve when moving to a commercial grade programming language such as C++ or Java.

Discussion

A teaching tool which attempts to reduce the impact of distractions from programming (for example user interface or unfamiliar syntax) while still maintaining a text-based command entering approach has been created. It is expected that this approach will result in a tool which is simple to learn and easy to use, while teaching transferable skill to novice programmers. The goal of this project is to provide novice programmers with skills which can be transferred to more commercial programming languages. In keeping with Papert's Power Principle (immediately providing benefit to the user) and Principle of Cultural Resonance (what is being learned must fit into a larger framework) code entry is the problem being focussed on. This dissertation also focusses on developing core programming skills, and so is not interested in a tool which does too much work for users. The benefits of introducing objects-first to novice programmers have also been discussed.

Usability, and related aspects of the user-interface, are critically important when designing software used to teach. [Squires and Preece, 1996] notes that "Thinking of usability and learning as separate [...] leads to superficial evaluation of educational software". Usability and learning outcomes are also linked in [**?**] who assert that "there is a need to help evaluators consider the way in which usability and learning interact". They also point out that poorly designed or frustrating interfaces will impact negatively on learning, and stress the need

for the interface to avoid any distractions from the material being learned. Learners complaining about web-based training often cite the confusing menus and unclear buttons which scare them off.

Research into Enterprise Resource Planning software in Turkey noted a strong link between usability of the systems and user satisfaction [Calisir and Calisir, 2004], perceived usefulness and ease-of-use have been strongly linked to user acceptance of technology [Davis, 1989], and research into university courses indicate a link between student experience and positive learning outcomes [Trigwell and Prosser, 1991]. It is clear from this wide spectrum of research that the perceived usability, usefulness and ease-of-use of any software will be directly linked to how that software is accepted and experienced by users. For educational software this may in turn positively impact their learning outcomes. This link has not been positively shown however, and at least one short term study has found no correlation between perceived usability and learning outcomes [Sim et al., 2006].

It can be seen from the research that a negative link between usability and learning outcomes exists, that is poor usability features are detrimental to the users learning outcomes. While no positive link has yet been found, it is entirely possible that it does exist. While it is not possible within the scope of this project to measure any meaningful learning outcomes, more research should be done in this area to see if any of the measures used in this study indicate a positive correlation between student experience of the software and their learning outcomes.

Project Hypotheses

This project examines users learning to program using SCOOT and C++ in terms of their experience and learning outcomes. It must be noted that measuring learning outcomes is not a concrete exercise, and that any learning outcomes recorded in this project were only from a very short exposure to the programming language or tool being used.

With that in mind the core hypotheses of this research project are :

H1 - SCOOT is seen by novice programmers as easier to use than C++, programmed in a customised version of Notepad++.

H2 - SCOOT is seen by novice programmers as more useful than C++, programmed in a customised version of Notepad++.

H3 - SCOOT is seen by novice programmers as more enjoyable than C++, programmed in a customised version of Notepad++.

As has been previously stated, C++ is a commercial programming language and SCOOT is designed for teaching, so we expect a positive result for H1. However,

we cannot assume that this will be the case and so it must be checked, no matter how likely it may seem. It is also interesting to see if learning OOP with SCOOT produces better learning outcomes than learning OOP with C++. Due to the short-term nature of the experiment, and the ever present difficulty in accurately measuring "learning outcomes" this was determined to be outside the scope of this research. However, any trends within the measures taken will be discussed. Issues of time restrictions aside, the hypotheses regarding learning outcomes are :

H4 - When given a simple programming task, users of SCOOT are able to write a correct solution more quickly than users of C++.

H5 - When given a simple piece of code, users of SCOOT are able to correctly understand it more quickly than users of C++.

H6 - When given a piece of code with errors in it, users of SCOOT are able to identify and correct the errors more quickly than users of C++.

The mechanical details of measurements taken for all hypotheses are discussed in Chapter 4.

Chapter 3

Framework and Language Design

3.1 Language Design Basics

It is necessary before discussing language design to clarify terms used when discussing aspects of a computer programming language. It is essential to differentiate between the language itself, and the programming environment. The language itself is the set of rules, keywords and control structures which can be used. Common commercial examples are Java or C++. The programming environment is the actual tool that a programmer uses to create programs, and may incorporate compilers, debuggers and other utilities. Examples of a programming environment (referred to as an Integrated Development Environment, or IDE) include Visual Studio, or NetBeans. The SCOOT tool incorporates a language with its own programming environment, and so the design decisions made will incorporate decisions about the language itself, and decisions made about the programming environment. These decisions will be discussed separately in the following sections.

Section 3.2 describes the tools used to define the components of the language,

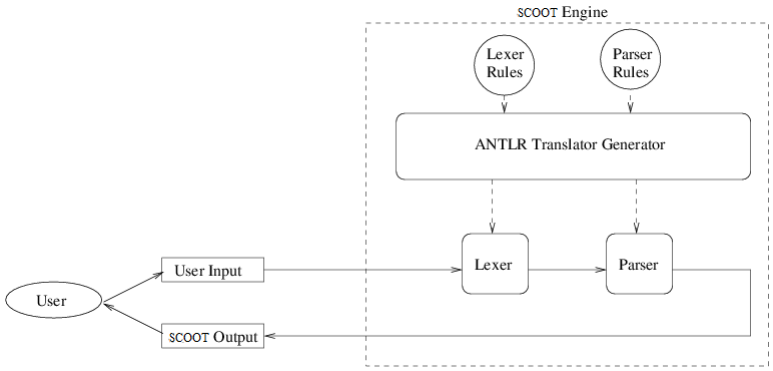


Figure 3.1: SCOOT Interaction.

and 3.3 discusses the internal structure of SCOOT. Section 3.4 identifies the keywords (or tokens) defined for use in SCOOT, and section 3.5 describes each component of the language. Finally section 3.6 describes the design decisions made about the programming environment users will use when programming in the SCOOT environment.

3.2 Lexers and Parsers

In order to understand the design of the commands within SCOOT, it is first necessary to understand the tools used in the development of the language. The two core components of the SCOOT engine are the lexer and the parser. Each serves a role in the process of taking user input, and responding appropriately. The interaction between the user and the SCOOT engine is shown in Figure 3.1. As can be seen the user input is processed by the lexer and then the parser, both of which are defined by the ANTLR tool and the SCOOT rule set. Once user input has been parsed the SCOOT engine will produce output in response, which is displayed to the user. All of the operations within the SCOOT engine are invisible to an end user.

Lexer

The lexer is used to take user input (as a series of characters) and break it down into recognisable tokens (or words). The lexer contains a list of all acceptable tokens, and user input is matched to that list. The list will contain all the defined keywords of the language, and will provide "variable" tokens to allow for the unique object (or attribute) names and values that users will use within their programs. The easiest way to illustrate this is by example. Consider a command which allows the user to set the value of a variable. The user may enter :

```
Set <variable name> to <value>
```

In order for the SCOOT engine to be able to process this input, several tokens must be defined. First of all the two keywords "*Set*" and "*to*", and secondly a means to handle both the variable name, and the value. Assume (for simplicity's sake) that the both the variable name and the value will be words. In order to handle the two keywords two tokens will be defined as below :

```
assign = "Set"
valueof = "to"
```

That is the two tokens are named *assign* and *valueof* and their values are *Set* and *to* respectively. To handle the variable name a more technical solutions is needed:

```
letter = ("a".."z"|"A".."Z")
word = (letter)+
```

Here 3 tokens are defined. The "|" symbol means "or", and the ".." means that the accepted value is between the two given values, inclusive. That is a *letter* is a single lowercase letter (between a and z) or an uppercase one. The "+" denotes that one or more instances of the character in brackets is required. That is a *word* is any pattern of letters, at least one character long. If these four tokens were defined in the lexer, then the user input :

```
Set apples to three
```

Will be broken down into the following pattern of tokens:

```
assign word valueof word
```

This pattern of tokens will then be passed to the parser for examination and handling.

Parser

The parser is responsible for taking a set of tokens as input, and producing the appropriate output. Within the parser will be a set of "rules" which match certain patterns of tokens. Attached to each rule is some behaviour to perform (in the form of code to execute). The combination of a rule, and its associated behaviour, is a grammar. User input is compared to each rule in order, until either a match is found, or there are no rules left. This means that the rules must be defined in such a way as to avoid confusion between them, a task made easier by SCOOT's use of the Pred-LL(k) parser ANTLR (see the following section for more discussion). It is also necessary for language designers to handle incorrect input, which matches no valid rule. Continuing with the example, in order to handle the above input the following rule must be defined :

```
setvalue : assign variable:word valueof value:word
```

Note that the phrase "*variable:word*" means that the parser will look for a *word* token, and will give it the variable name *variable*. Once this rule has been matched the values of the two *words* will be stored in the appropriate variables (in this case, *variable* = "apples" and *value* = "three"). These values can now be used (if desired) in the behaviour associated with the rule.

As a simple example, the following Java code could be used :

```
{
    System.out.println(“The number of “
    + variable.getText() + “ has been set to “
    + value.getText());
};
```

Which would display the text “*The number of apples has been set to three*” on the screen. Note that this contrived example does not actually set the value of a variable called *apples* to *three*, it simply produces output.

Predictive Parsing

Parsers allow the interpretation of human-readable statements. When examining a statement the parser analyses the characters of the statement from left to right and constructs a derivation. A parser which constructs leftmost derivations is an LL [Parsons, 1992] parser, and a parser which constructs rightmost derivations is an LR [Parsons, 1992] parser. The leftmost derivations constructed by the LL parser are evaluated from left to right, (top to bottom in a tree structure) allowing for recursive descent. Parsers lookahead k tokens and the larger the value of k, the more complex the tables, and the more powerful the grammars become at recognising statements. The particular LL and LR parsers which allow dynamic values of k are called LL(k) and LR(k) parsers respectively.

Due to the variable nature of some tokens, it is possible to find a situation where the difference between two rules cannot be seen [Parr, 1997]. It is possible to define a set of rules for defining this differentiation in the lexer using predicates. These predicates are applied before parsing, and allow the parser to determine the difference between keywords, and ID's. Predicates also allow for an earlier differentiation between rules, reducing the complexity overhead of using LL(k) parsers. In order for this to occur, however, the compiler

generator tool being used must support predicates [Parr and Quong, 1995]. Careful definition of these predicates and grammars will allow a natural language interface to be created for a pre-existing programming language using predictive-LL(k) parsers [Parr, 1993].

Two modified versions of LR(0) parsers, which are more powerful than LR(0) but not as good as LR(1), are SLR and LALR [Parsons, 1992] parsers. LALR is a popular version of LR(0) which uses extra "lookahead" contextual information for table generation. Predictive-LL(k) parsers allow simpler compiler writing, while being powerful enough to describe languages which could only be previously captured by using LALR.

The remainder of this chapter will discuss the tokens defined within the lexer in SCOOT, and then examine in detail the grammars of the language. A justification for certain programming constructs will be given along with the SCOOT commands associated with each, and an overview of the internal behaviour of SCOOT in each case.

3.3 SCOOT Infrastructure

In order to implement objects as required in SCOOT it was decided to create three classes, each of which implements a different aspect of an OO environment. All details of this implementation are hidden from programmers. Using the Java API, a TreeMap of instances of the *scootObject* class, named *scootObjects,* is used to maintain information about all the objects within SCOOT. TreeMap is a sorted tree structure, and allow efficient access to elements based on a key value. All objects in SCOOT are identified by their *Name*, and the TreeMap implementation will allow access to objects in the same way. A simple TreeMap is shown in Figure 3.2. A TreeMap can store any object in its *value* field, and only requires that all keys be unique and comparable (i.e. they can be legitimately compared to one another).

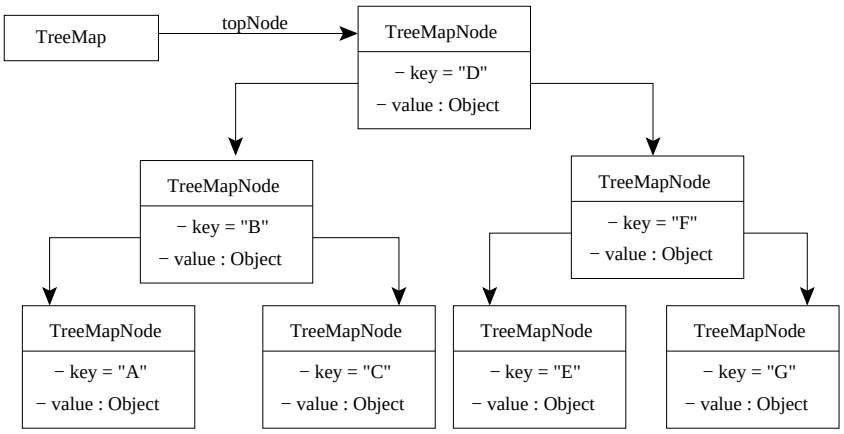


Figure 3.2: TreeMap Data Structure, Showing TreeMapNodes.

As shown in Figure 3.3, a *scootData* object contains a *Name* and a *Value*, and a *scootObject* object contains a *Name* and a TreeMap called *Data*. The *Data* TreeMap contains all the *scootData* objects associated with this *scootObject*. The TreeMap of *scootData* objects is keyed on the *Name* of the data element. Figure 3.4 shows the *scootObjects* TreeMap for a simple state with one object (named *Rectangle*) containing 3 pieces of data (*Height*, *Width*, and *Colour*).

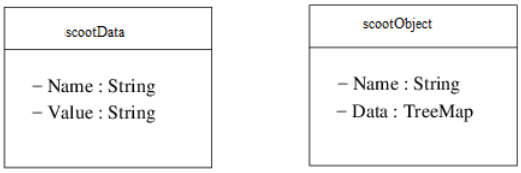


Figure 3.3: scootData and scootObject Class Diagrams.

When the implementation of methods was planned, it was decided to logically separate the methods from the objects they belong to. This was done in order to improve memory use in a scenario where several objects share a complex method. The method will be stored in a single location, with a TreeMap of all the objects it can be called by, whereas if methods were stored relative to their objects, there would be several copies of an essentially identical method. A TreeMap of *scootMethod* objects called *scootMethods*, which uses the *Name* of the method as the key, is used to store all the *scootMethod* objects within

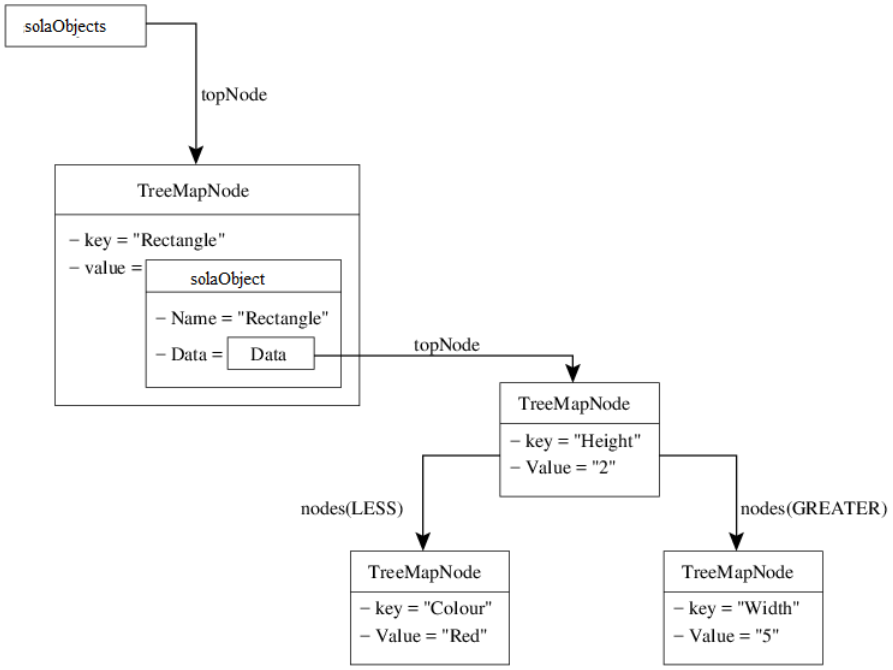


Figure 3.4: scootObjects Breakdown.

SCOOT. As shown in Figure 3.5, a *scootMethod* is defined by four variables:

- *Name* - used to store the name of the method,
- *Types* - a TreeMap containing the names of all *scootObject*s this method belongs to,
- *Steps* - the full method listing, containing all the SCOOT commands for this method, and
- *Count* - the number of steps in the method.

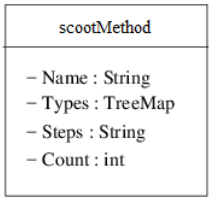


Figure 3.5: Data Structures Used By SCOOT To Maintain Methods.

There is no imposed limit on the number of *scootObjects* or *scootMethods* that can exist, or on the number of *scootData* objects that can be attached to each object. All rules, unless otherwise stated, execute their associated code when matched if the Boolean value *doingParse* is set to true, otherwise the input stream is copied into a variable called *Input*, and the *steps* counter is incremented. This is used for defining methods, loops and conditionals as will be discussed later. In addition to this, if a rule accesses an object, an attribute or a method that does not exist, an exception will be generated.

3.4 Lexer Tokens

When creating the grammar rules which would be used to define SCOOT, it was necessary to define the set of pre-defined tokens (or keywords), which would have special and particular meaning within the language. A comprehensive list of these tokens is given here in EBNF notation.

Symbol	Meaning
→	Associates a token name on the left with the matching string, on the right.
|	Indicates alternate forms, such as upper or lower case
*	Indicates that the preceding tokens in brackets may not occur, or may occur many times
+	Indicates that the preceeding tokens in brackets must occur at least once, but may occur many times
?	Indicates that the preceeding tokens in brackets occurs at most once, or not at all

Lexer Tokens

The tokens for matching letters, digits and IDs are:

$$
\begin{aligned}
LETTER &\rightarrow ('a'..'z' \,|\, 'A'..'Z') \\
DIGIT &\rightarrow ('0'..'9') \\
ID &\rightarrow LETTER(LETTER \,|\, DIGIT)*
\end{aligned}
$$

To explain, a *LETTER* is a single character, either lowercase or uppercase. A digit is a single numerical character between "0" and "9", while an *ID* is a *LETTER* character, followed by zero or more *LETTERs* or *DIGIT*s.

The tokens for matching mathematical operators are:

$$
\begin{aligned}
PLUS &\rightarrow + \\
MINUS &\rightarrow - \\
MUL &\rightarrow * \\
DIV &\rightarrow / \\
MOD &\rightarrow \% \\
POW &\rightarrow \hat{} \\
LPAREN &\rightarrow (\\
RPAREN &\rightarrow) \\
INT &\rightarrow (DIGIT)+
\end{aligned}
$$

Note that the above list includes LPAREN and RPAREN, allowing these characters to be entered by users. Pairs of brackets required for the +, * and ? metasymbols are not affected by this.

The tokens for matching key words are:

$$
\begin{aligned}
ISA &\rightarrow is\,a \\
ISAN &\rightarrow is\,an \\
HASA &\rightarrow has\,a \\
HASAN &\rightarrow has\,an \\
A &\rightarrow A
\end{aligned}
$$

$$
\begin{aligned}
AN &\rightarrow An \\
GET &\rightarrow Get \\
SET &\rightarrow Set \\
OF &\rightarrow of \\
TO &\rightarrow to \\
PERFORMS &\rightarrow Performs \\
EVALUATE &\rightarrow Evaluate \\
ENDMETHOD &\rightarrow End\,Method \\
COMPARE &\rightarrow Compare \\
IF &\rightarrow If \\
COMPR &\rightarrow (`equal|less\,than|greater\,than) \\
THENDO &\rightarrow then\,do \\
ENDCOND &\rightarrow End\,If \\
WHILE &\rightarrow Repeat\,while \\
ENDLOOP &\rightarrow End\,Repeat \\
QUIT &\rightarrow Quit \\
CANCEL &\rightarrow Cancel \\
REMOVE &\rightarrow Remove \\
FROM &\rightarrow from \\
ADD &\rightarrow Add \\
SUB &\rightarrow Subtract \\
MULTIPLY &\rightarrow Multiply \\
DIVIDE &\rightarrow Divide \\
BY &\rightarrow by \\
SEMI &\rightarrow ;
\end{aligned}
$$

These tokens allow a set of rules (grammars) to be defined which form the basis of the commands available within SCOOT.

3.5 Language Components

Deciding which core programming constructs should be included in SCOOT was a major part of the project. In order to make these decisions the purpose of SCOOT was examined, and constructs selected which would best serve to introduce OO concepts to novices. While each programming language has its own syntactic and semantic rules, there are many underlying concepts which are common across all languages. In SCOOT these constructs are introduced in a way that will allow novices to easily become familiar with the concepts, without the complexity of learning a commercial programming language. Figure 3.6 shows a conceptual outline of the structure of programming constructs within SCOOT.

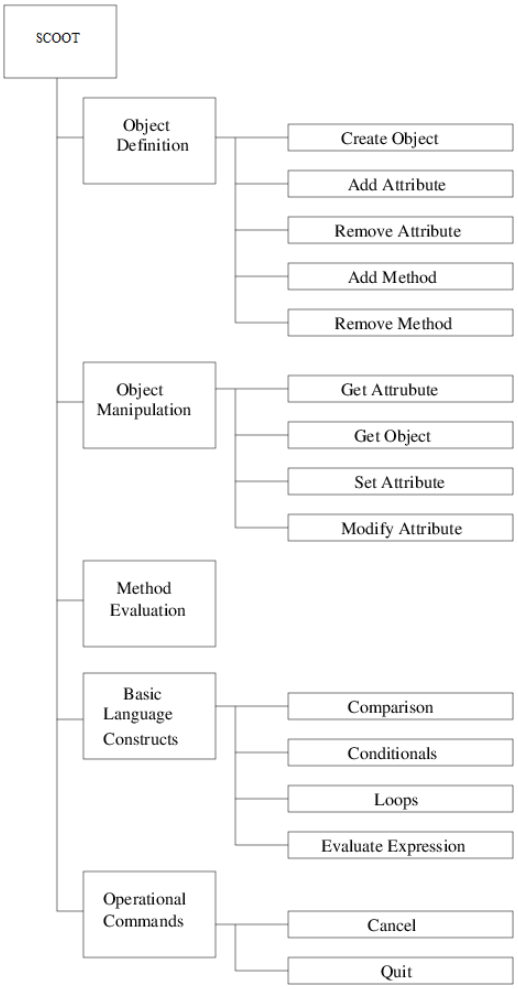


Figure 3.6: Conceptual Structure Of SCOOT.

In this section each programming construct introduced in SCOOT will be justified, as will the form of the SCOOT commands which users will enter. The parser rules used to match the input commands are also given, as well as an explanation of how SCOOT internally processes the instructions. Once all included programming constructs have been discussed, and their respective parser rules defined, interface design will be briefly discussed.

When designing the command semantics and syntax of SCOOT, each different phrase was checked to ensure it made sense, both inside the context of SCOOT, and in terms of the English phrases being used. SCOOT is heavily

based on the syntax and structure of the English language, and so learners from a non-English background may not benefit from the "English-like" nature of the language.

3.5.1 Create Object

The ability for programmers to create and define their own objects is essential to any OO programming environment. Without this ability the programmers could only manipulate pre-defined sets of objects, which reduces the degree to which the environment engages them. It was therefore decided that programmers must be able to define their own objects.

In SCOOT an object declaration (the specification of what composes an object) is not separated from its instantiation, and so it was decided that in the initial prototype, programmers would only be able to copy an existing object. In order to remain consistent with inheritance rules from commercial OO languages, all methods and attributes, including values, are copied from the original object. Note that since the code for an objects methods is copied, those methods can now be executed with respect to this object. In this way, a small degree of polymorphic behaviour is demonstrated, since methods with the same name can operate on different objects, but perform the same tasks.
Since all objects created by a user must be based on another object, an empty class (with no attributes or methods) called *Object* is built into SCOOT. Programmers will use this as the base for their new objects, unless an object is based on another class. This formalises a trend in many commercial OO programming languages, where all objects are related to a standard base class (as in Java).

It is worth noting here that inheritance in SCOOT is a simple affair, designed only to introduce the concept of a derived object extending a base object. As such certain aspects of SCOOT's inheritance are not standard. For example a programmer can modify a base class after an inheritance has occurred, and this will not modify the derived class. Inheritance in SCOOT is akin to object

extension, rather than pure inheritance.

The SCOOT command to declare a new object, remembering that all new objects will be based on pre-existing objects, is :

```
A <new object name> is a <old object name>;
```

The SCOOT command used for copying an object had to clearly express that the new object is based on an existing object, since in an OO paradigm, all objects are related.The phrase "*is a*" was chosen as a suitable semantic phrase, indicating the relationship between an existing object and the new. For example, "*A Rectangle is a Shape;*" would create a new object called *Rectangle* with the same attributes and methods (if any) of the object *Shape*. In order to maintain consistency with standard English, both the initial "A" and the "a" in "is a" can both be replaced with the word "an" for grammatical correctness (for example, "An Oval is a Shape;"). It should be noted that grammatical correctness is not enforced.

This input will be matched by the parser grammar:

```
A newObject:ID ISA oldObject:ID SEMI
```

Before creating a new object a check is performed to ensure that a *scootObject* does not exist with the *Name* of *newObject*. Once this is done, a new TreeMap is created, and the elements of the *Data* TreeMap of the object with the *Name* of *oldObject* are stored in an array. This is because a standard TreeMap copy is shallow, and any change to the new *scootObject* will affect the old one. The elements of this array are then used to populate the temporary TreeMap, and a new *scootObject* with the *Name* of *newObject* and the *Data* of *oldObject* is added to the *scootObjects* TreeMap. Once this is done, the *scootMethods* TreeMap is thoroughly searched, and any *scootMethod* which could act on the object with the *Name* of *oldObject* is updated so that it also operates on the *scootObject* with the *Name* of *newObject*. This search effort reduces the overhead of method storage.

3.5.2 Add Attributes

If programmers can create their own objects, it is also necessary that they be able to store information in those objects, and so users are able to add attributes to their objects. It was decided that in order to allow programmers to have a greater level of control, they would be able to dynamically add attributes to SCOOT objects. In a commercial language such as Java they would have to create a class which extended the base class in order to achieve this, but in a command based system like SCOOT there is no need for such limitations.

The ability to dynamically add their own attributes will allow programmers to interact with SCOOT more easily, since an object need only be partially defined in order to be used. In the same way, since attributes can be easily added to an object, a more complete set of attributes can be added at a later time, when the programmer is ready or has identified a new use for the object.

The SCOOT command to add an attribute to an object is :

```
A <object name> has a <attribute name>;
```

Determining the syntax for the addition of an attribute was fairly straightforward, a common English way of describing an object was identified. When describing the fact that an object has a certain attribute, the English phrase "A rectangle has a length" or "A rectangle has a width" is easily understandable. For this reason, this phrase was chosen to describe the fact that an object now has a new attribute to describe it. To add the attributes *length* and *width* to the object *rectangle* the programmer enters "A rectangle has a length;" followed by "A rectangle has a width;"

This form is a simple English phrase which we believe will be understood by novices, and so is ideal for use in SCOOT. No previous rules have been defined for adding to an object, so internal consistency is not a consideration here. As in Section 3.5.1 both the opening "A" and the "a" in " has a" can be replaced with the word "an".

A new parser grammar is defined to handle this input:

```
A object:ID HASA attribute:ID SEMI
```

When this grammar is matched, after basic error checking, the *scootObject* whose *Name* is *object* is accessed, and a *scootData* object whose *Name* is *attribute* and *Value* is NULL will be added to *objects Data* TreeMap.

3.5.3 Remove Attributes

Editing previous commands is impossible within SCOOT, and in early testing of SCOOT many programmers made mistakes in adding attributes to their objects. These errors involved misnaming them or sometimes getting confused about which object they wanted to work with. It was therefore decided to allow programmers to dynamically remove attributes from SCOOT objects. As mentioned above in Section 3.5.2, this is not a feature commonly available in commercial programming language, but is used here to minimise the potential negative impact of an interpreted programming environment.

The SCOOT command to remove an attribute of an object is simply:

```
Remove <attribute name> of <object name>;
```

So, in order to remove the attribute *Colour* from *Rectangle* the programmer would enter:

```
Remove Colour of Rectangle;
```

This structure reinforces the important OO concept that the attribute (*Colour*) exists only with respect to the object (*Rectangle*).

The parser grammar to match this input is:

```
REMOVE attrName:ID OF objName:ID SEMI
```

When this grammar is matched the *scootObject* whose *Name* is *objName* is accessed, and the *scootData* object whose *Name* is *attrName* will be deleted from *objects Data* TreeMap.

3.5.4 Add Method

Functions are a common concept in programming, and object methods are used in OO languages to define a function which operates on the object which called it. In order to introduce these ideas into a novice teaching tool, without worrying about such concepts as function overwriting/overloading, it was decided to allow programmers to add their own behaviours to objects. This is simply another means by which programmers can define their own objects. For simplicity, methods do not take input, they only provide output. Since no input parameters are accepted, method overloading is not incorporated into the prototype of SCOOT. The ability to specify input parameters, and hence overload methods, will be analysed as an addition to SCOOT in future work.

The SCOOT command to add a method to an object is :

```
A <object name> performs <method name>;
```

If internal consistency was the only concern, it would make sense for the syntax to be consistent with that for adding an attribute. However the phrase "A Rectangle has a DoubleSize" does not clearly convey the meaning that *DoubleSize* is a method of the *Rectangle* object, and does not linguistically differentiate between a method and an attribute. Therefore, in the interests of readability and understandability the syntax will not be directly linked to that of adding an attribute. It is still desirable to clearly indicate the link between the object and the method, and if possible signify ownership of the method by the object. To that end, to add a method called *DoubleSize* to the object *Rectangle* the programmer will enter "A Rectangle performs DoubleSize;"

This syntax clearly indicates that *DoubleSize* is something that happens, not simply an attribute, and that it is owned by the object *Rectangle*. In this way the syntax remains consistent with the programmer's experience with English, while not conflicting with standards already established in SCOOT. To finalise the input of the method the programmer enters:

```
End Recipe;
```

This is the first use of the keyword *Recipe* and may be confusing to users. For this reason two concessions have been made in the interests of improved usability:

1. When a programmer begins entering a *Recipe* as described above SCOOT will inform them that they have begun entering a *Recipe* and to enter ¨*End Recipe;*¨ to finish entering the *Recipe*, and

2. A revised form of the statement will be developed for use in a future version of SCOOT, which will take the form ¨*End DoubleSize;*¨ to finish entering the *Recipe* called *DoubleSize*.

To handle the commands needed to add methods to objects, two new productions are needed. Firstly :

```
A object:ID PERFORMS method:ID SEMI
```

When this production is matched the values of *object* and *method* are stored in variables within the parser, the parser variable *doingParse* is set *false*, and the counter *methSteps* is initialised. All input commands after this will be compared to the parser rules, to ensure correct form, and added to the *Input* variable, while *steps* is incremented. No input after this command is executed, until the input matches the production:

```
ENDMETHOD SEMI
```

Once this production has been matched, a new *scootMethod* object is created, and the *Name* of *object* is added to its *Types* TreeMap. The *Name* of the *scootMethod* is *method* and the value of *Steps* is copied from *Input*. In order to ensure that this method can operate on different objects effectively, any references to *object* in the command will be converted to the internal keyword OBJECT. The value of the *scootMethod*'s *Steps* is calculated from the value of the parsers *steps* and the value of *methSteps*. Once the *scootMethod* has been

created, it is added to the *scootMethods* TreeMap, and *Input* is cleared. SCOOT will then continue handling input like normal.

3.5.5 Remove Method

The ability to remove a method for an object on the fly is provided due, again, to the limitation of not being able to edit commands in the SCOOT environment.

The SCOOT command to remove a method is:

```
Remove <method name> from <object name>;
```

This reinforces the "ownership" of the method by the object, while remaining different from the syntax to remove an attribute. This was done intentionally to reinforce the concept that an attribute describes an object, while a method determines its behaviour. If a programmer wishes to remove the method *DoubleSize* from the *Rectangle* object they can simply enter the command "*Remove DoubleSize from Rectangle;*".

This input will be handled by the grammar:

```
REMOVE methName:ID FROM objName:ID SEMI
```

When this grammar is matched the *scootMethod* object whose *Name* is *methName* will have *objName* removed from its *Types* TreeMap.

3.5.6 Get Attribute

Accessors and mutators are those functions that allow programmers to respectively set and get the value of an object's attributes. They are a common and important aspect of OO programming languages, and so play an important role within SCOOT. Storing information within an object is only useful if the data can be accessed at some point, and that is the functionality provided here. Note that in a commercial programming language the programmer would be required to write the accessor functions themselves. However, given the introductory nature of SCOOT it was decided that providing built in accessors to all object data would make object inspection easier.

The SCOOT command to view the value of an attribute is:

```
Get <attribute> of <object>;
```

In order to keep the language within the bounds of what a novice programmer can be expected to be familiar with, and to establish certain programming standards, these functions are accessed using the keyword *Get*. In commercial OO languages, functions which return the value of an attribute are commonly referred to as get functions. To find out the value of the attribute *Colour* of the object *Rectangle* the programmer enters "*Get Colour of Rectangle;*". This syntactical format is similar to the language a novice programmer will be familiar with, while at the same time introducing the concept of set and get functions.

This input will be handled by the grammar:

```
GET attribute:ID OF object:ID SEMI
```

When this grammar is triggered SCOOT will access the *scootObject* whose *Name* is *object* and display the *Name* and *Value* of the *scootData* object named *attribute*.

3.5.7 Get Object

Late in the development of SCOOT it was decided that an object accessor, that is a single function which displays all the attributes of an object with their respective values, would allow programmers to easily gain an awareness of what comprises an object, and of its current state. Inquisitive programmers are now able to explore objects more easily.

To view the full contents of an object, the SCOOT command:

```
Get <object name>;
```

is entered. By keeping the format and style of similar or connected commands consistent, unnecessary complications are avoided and the learning curve of the language is reduced. When it was decided to add an object accessor function,

the syntax was naturally derived from the already established format for the attribute accessor. Therefore, when programmers wish to view the full state of the *Rectangle* object they enter "*Get Rectangle;*". By remaining consistent with the use of the keyword *Get,* a common usage of this keyword will be established in the minds of the programmer, thereby reducing the difficulty of learning the use of SCOOT.

The parser grammar:

```
GET object:ID SEMI
```

will handle this input. SCOOT will access the *scootObject* whose *Name* is *object* and display the name and value of all *scootData* objects in the TreeMap *Data* attached to that *scootObject*.

3.5.8 Set Attribute

Without the ability to manipulate and examine an object there is no interaction on the programmer's part, and the interactions of the objects within SCOOT would be meaningless. By providing mechanisms to allow programmers to manipulate the objects in the environment, the programmer becomes engaged and more advanced constructs can be defined. As in 3.5.6, this functionality would need to be coded by the programmer in a commercial language, but is provided here automatically for all object attributes to ease the learning curve for novice programmers.

The following SCOOT command will specify the value of an attribute, named with respect to an object.

```
Set <attribute> of <object> to <value>;
```

As described in 3.5.6, SCOOT was designed to establish common programming practices in the minds of novice programmers. In many commercial OO languages a function which specifies the value of an attribute is referred to as a "set function". Since the language is logical in English, and since it is consistent

with previously established programming practices, and with SCOOT's syntax, SCOOT uses the keyword *Set* to specify the value of an attribute. To set the *Colour* of *Rectangle* to "Red", the programmer enters "*Set Colour of Rectangle to Red;*".

Since a numerical value will be recognised as a number, and a textual value as a word, two separate grammars are needed to handle this instruction. They are:

```
SET attribute:ID OF object:ID TO value:ID SEMI
SET attribute:ID OF object:ID TO value:INT SEMI
```

The grammar for setting *attribute* to a specific *value* is divided into two separate sub-grammars, one for handling *value* as a string, and the other for handling a number. If *value* is a number, it is first converted to a string, then both grammars proceed by accessing the *scootObject* whose *Name* is *object* and setting the *Value* of the attached *scootData* object named *attribute* to the value of *value*.

3.5.9 Modify Attribute

Simply being able to specify the value of an attribute is not enough, users must be able to modify the value of a numerical attribute using basic arithmetic expressions. The earliest version of SCOOT only allowed for simple addition, and the language used was extremely ungainly. The desire for internal consistency (specifically with the "*Set*" command) led to the phrase (for example) "*Set Width of Rectangle to Width of Rectangle + 2;*", which all programmers found messy. By re-prioritizing the need for consistency with familiar English, and expanding the pool of operations, the following four commands were developed.

These four commands will, respectively, add, subtract, divide or multiply the value of the attribute, using the given numerical value:

```
Add <number> to <attribute> of <object>;
```

```
Subtract <number> from <attribute> of <object>;

Multiply <attribute> of <object> by <number>;

Divide <attribute> of <object> by <number>;
```

Note that although the commands are not identically structured, at the cost of some internal consistency, the familiarity to similar phrasing in spoken English was the highest priority here. With this new format the previous example, of increasing the *Width* of *Rectangle* by 2 would now be written as "*Add 2 to Width of Rectangle;*"

It is also possible to add two values together using the input:

```
Set <attribute 1> of <object 1> to <attribute 1> of
<object 1> + <attribute 2> of <object 2>;
```

This form allows two separate attributes, possibly from two separate objects, to be added together. In the prototype of SCOOT tested in this study a direct value copy is not possible, nor are other arithmetic expressions between objects. This simple extension will be incorporated in a future version of SCOOT.

The above commands will result in one of the following grammars being called:

```
ADD addValue:INT TO attrName:ID OF objName:ID SEMI

SUB subValue:INT FROM attrName:ID OF objName:ID SEMI

MULTIPLY attrName:ID OF objName:ID BY multValue:INT SEMI

DIVIDE attrName:ID OF objName:ID BY divValue:INT SEMI
SET attrName:ID OF objectName:ID TO compElement(continued next line)

PLUS rightAttrName:ID OF rightObjName:ID SEMI
```

For each of these grammars a few basic error checks are conducted, specifically to ensure that the values in question are numerical. Once that is confirmed the appropriate mathematical operation is conducted, with the resulting integer value being copied into the *value* field of *attrNames* entry in *objNames Data* TreeMap. For the fifth grammar, an additional step is performed to get the value of *rightObjNames* attribute *rightAttrName*, and ensure it too is numerical.

3.5.10 Method Evaluation

Since SCOOT objects have methods attached to them, it is necessary to provide a simple mechanism to invoke these methods. Including this feature is essential in order to allow programmers to understand the nature of objects in an OO environment, and excluding it makes the ability to add methods to objects meaningless.

Within SCOOT, methods (called Recipes) are a set of steps to be carried out, identified by the name of the recipe, and the object. In order to reinforce the notion that a method executes with respect to an object, and not independently, the programmer enters:

```
Evaluate <method name> of <object name>;
```

This command reinforces two key concepts: firstly, a method (or recipe) exists only with respect to an object and secondly, unlike an attribute a method must be evaluated (or run).

This input will trigger the grammar:

```
EVALUATE method:ID OF object:ID SEMI
```

Firstly, standard checks are performed to ensure that the *scootMethod method* exists and operates on the *scootObject object*. Then a new StringBuffer is created based on the *Steps* of the *scootMethod*, and this is used as the basis for a new instance of the parser and lexer. Since the value of *Steps* was altered to include the keyword OBJECT rather than the name of the object it was acting on, that change must be undone here. The String used to instantiate the StringBuffer is altered so that it contains the *Name* of *scootObject object* in the place of OBJECT. The first step is then fed into the parser, and as long as there are more steps to perform, SCOOT will parse the input stream. Once the method has been fully parsed, control is returned to the programmer.

3.5.11 Compare attribute to value

In order to provide two foundational programming constructs, conditionals and loops (see Sections 3.5.12 and 3.5.13) it is necessary for programmers to be able to compare the value of a particular attribute to a set value. In the prototype version of SCOOT it was decided to separate the comparison from the actual conditional/looping phrase, which is different from many commercial programming languages. This was done to establish the notion that the comparison is separate from the conditional or loop itself. For this reason all conditional or looping blocks need to be preceded by a comparison statement. The form of this statement is:

```
Compare <attribute> of <object> to <value>;
```

This form is consistent with the syntax developed thus far for SCOOT, and makes grammatical sense in English. The form of the conditional statement was selected as a blend of consistency with common programming languages, and familiarity with the English language. Note that the result of the *Compare* statement may be *equal*, *less than* or *greater than*. For example, the command "*Compare Width of Rectangle to 5;*" would result in the value "true" being returned, if *Width* of *Rectangle* was 5. For *String* comparisons the result will be based on a lexicographical comparison (for example, "Red" is greater than "Blue").

As in 3.5.8, different parser rules are required to handle numerical or textual input. User input will be handled by the appropriate grammar below:

```
COMPARE compElement TO value:ID SEMI
COMPARE compElement TO value:INT SEMI
```

where

$compElement \rightarrow ID\ OF\ ID$

A comparison tests if the two arguments, one an attribute of an object and the other a value, are equivalent. The comparison can take a string or an integer

as the value to compare. By choosing to type convert an integer to a string, the value comparison code is in fact shared. The *compElement* rule finds the value of the attribute in question and stores it in the parser variable *lhs*. This value is then compared to the value of *value,* and the result is displayed within the SCOOT environment and stored in the parser variable *compResult* for later access. When a conditional or loop statement is executed, the result of the comparison can be obtained by inspecting *compResult*. For use in a loop the comparison string is stored in the parser variable *comp*, since this value should be constantly updated.

3.5.12 Conditionals

Conditional statements are a key component of all programming languages. Even the most OO program will contain fragments of procedural code within it. The three core components of procedural programs are :

1. Sequence (one step after the other, in order);
2. Selection (having one or more blocks of code which are only evaluated under certain circumstances);
3. Repetition (a block of code which is evaluated until some condition is met, with that condition being checked at certain points in execution).

In order to enable novice programmers to understand that some code can be made conditional i.e. that some operations (or object actions) can be dependent on results (or objects attributes), SCOOT will allow programmers to use conditional statements. These can be used directly from the translator, or included in a programmer defined method. In order to limit the complexity of the system it is not possible to use nested conditionals, or the more convenient case statements found in many modern languages.[1] In SCOOT a conditional statement is a simple test as shown in Section 3.5.11, followed by commands that will be executed if the input condition matches the output of the test.

[1] Other languages, including Python, also do not support a case/switch construct.

Once the comparison has been performed, the programmer can begin a conditional block by entering:

```
If equal then do;

<other commands go here>
```

The commands that follow will be evaluated if the result of the preceding comparison was "equal" (that is, if the two values being compares are in fact equal). Once this is done the conditional block must be terminated by entering:

```
End If;
```

Once again, this form was chosen for its similarity to modern programming languages, as well as its consistency (or lack of inconsistency) with SCOOT syntax, and its clear meaning in English. The need to firmly denote the end of a block (rather than just to do so via formatting, or a single character) was to enforce the notion of code blocks having a fixed beginning and ending. Once this is established, programmers can easily adapt to less formal declarations such as those found in Python.

The initial command will be matched by :

```
IF condition:BOOL THENDO SEMI
```

When a conditional block is begun, *doingParse* is set to *false*, and the parser variable *ifCondition* is set to the value of *condition*. The *ifSteps* counter is initialised, the value *conditional* is set to *true* and the result of the previous comparison is copied into *condResult*. Input after this point is checked, to ensure it is valid SCOOT input, but not executed. Instead it is copied into *Input* and the counter *ifSteps* is incremented for each line.

```
ENDIF SEMI
```

Once the conditional block has been ended, the values of *doingParse* and *conditional* are reset, and the result of the comparison compared to the input

condition stored in *ifCondition*. If the two Boolean values are equal a new StringBuffer is created based on *Input*, and the number of steps in the block calculated from *ifSteps* and *steps*. Each statement in *Input* is then parsed and evaluated. After completion, *Input* is cleared.

3.5.13 Loops

Another common programming construct included in SCOOT is the loop construct. This construct allows an operation to repeatedly take place until a specified condition is met. Like the conditional statements, loops are not unique to object-oriented environments, but are common across all imperative programming environments. Unlike many languages, however, SCOOT only supports one variety of loop, that being a pretest while loop. This decision was made for two reasons:

1. To reduce the complexity of the loop structure for users by avoiding complications associated with multiple forms,

2. Taking advantage of a pretest condition to emphasise preparation occurring before any actual repetitive object actions

Once a comparison has been performed the start of a block which loops until the condition is not met (note the use of the word *while*) can be declared by entering:

```
Repeat while less than;
```

Again, the test condition used here could also be "*greater than*" or "*equal*". This syntax is consistent with the form of the *Do While* loops found in other languages, and is clear English, so confusion is minimised. Once the looping block is opened, programmers enter rules of the standard format until the looping block is completed. To close the looping block the programmer simply enters:

```
End Repeat;
```

The advantage of using the keyword *Repeat* over the more common programming phrase *Loop* lies in the familiarity of the word. To a computer programmer a loop is a basic construct, but to a novice it is not so obvious. However telling a novice that something repeats will have a far greater level of effectiveness in communicating the same idea. As in many languages, a poor choice of loop condition may result in an infinite loop. SCOOT does not yet handle this situation well.

```
Compare payRate of Employee to 55000;
Repeat while less than;
Add 1000 to payRate of Employee;
End Repeat;
```

The command to begin a loop will match:

```
WHILE condition:BOOL SEMI
```

When a looping block is begun *doingParse* is set *false*, *looping* is set to *true*, *loopCondition* is set to *condition*, *loopSteps* is initialised and *loopResult* is set equal to the result of the previous comparison. As with a conditional, input is tested and stored in *Input*, and the number of lines entered stored in *loopSteps*.

```
ENDLOOP SEMI
```

Once the loop has been ended the values of *doingParse* and *looping* are reset, and the SCOOT command for the comparison is appended to the end of *Input*. If the conditions are equal a new StringBuffer is created based on *Input*, and this is used as the basis for a new parser. When the end of *Input* is found, the looping condition is evaluated again, and if required the process of recreating the StringBuffer and parser will occur again, until the input and test conditions are not equal.

3.5.14 Evaluate Expression

In SCOOT it was desirable to make mathematical syntax as recognisable as possible. For this reason it was decided that only a very minor addition would be imposed on programmers, that is the use of the keyword *Evaluate*, as used in method evaluation. In this way programmers can enter an arithmetic expression, within brackets, in standard infix notation:

```
Evaluate <arithmetic expression>;
```

For example, in order to evaluate the arithmetic expression 3 * (7-2), users simply enter "*Evaluate (3 * (7-2));*". Mathematical evaluation in SCOOT allows for multiple levels of nested parentheses, the basic operations of addition, subtraction, multiplication and division, as well as the ability to perform exponentiation and modulo operations.

Productions used to evaluate an arithmetic expression.

$$
\begin{aligned}
numExpr &\rightarrow LPAREN\ sumExpr\ RPAREN \\
sumExpr &\rightarrow prodExpr\ ((PLUS \mid MINUS)\ prodExpr)* \\
prodExpr &\rightarrow powExpr\ ((MUL \mid DIV \mid MOD)\ powExpr)* \\
powExpr &\rightarrow atom\ (POW\ atom)? \\
atom &\rightarrow INT \mid expr
\end{aligned}
$$

Rules to handle arithmetic expressions are described below.

$$
\begin{aligned}
expr &\rightarrow (numExpr) \rightarrow numExpr \\
&\mid (ID\ OF\ ID) \rightarrow evalMeth
\end{aligned}
$$

Mathematical evaluation is done as outlined in [Parr and Quong, 1995] by using a ParseTree to store the operations. All non-mathematical tokens are

excluded from this tree, through the use of a special ANTLR token. Each operation is added to the tree, with each operator becoming a parent with two children, the operands. This tree is then parsed using a simple TreeParser, which evaluates each operation in order to resolve a final output, which will be displayed within the SCOOT environment. The order of operations is guaranteed through ANTLR's look-ahead(k) mechanism.

To give an example, `Evaluate (3 * (7-2));` would be tokenized as `EVALUATE LPAREN a:INT MUL LPAREN b:INT MINUS c:INT RPAREN RPAREN SEMI,` where a=3, b=7 and c=2. This would then be processed as follows:

`EVALUATE LPAREN a:atom MUL LPAREN b:atom MINUS c:atom RPAREN RPAREN SEMI`

`EVALUATE LPAREN a:powExpr MUL LPAREN b:powExpr MINUS c:powExpr RPAREN RPAREN SEMI`

`EVALUATE LPAREN a:powExpr MUL LPAREN b:prodExpr MINUS c:prodExpr RPAREN RPAREN SEMI`

`EVALUATE LPAREN a:powExpr MUL LPAREN d:sumExpr RPAREN RPAREN SEMI` where d = 7-2

`EVALUATE LPAREN a:powExpr MUL d:numExpr RPAREN SEMI`

`EVALUATE LPAREN a:powExpr MUL d:powExpr RPAREN SEMI`

`EVALUATE LPAREN e:prodExpr RPAREN SEMI` where e = 3 + (7-2)

`EVALUTE LPAREN e:sumExpr RPAREN SEMI`

`EVALUATE e:numExpr SEMI`

This numExp will then be evaluated using a parseTree to calculate the result as 8.

Evaluation of a literal mathematical expression is the only place in SCOOT where the asterisk is used for multiplication, in all other cases is is handled by the keywords described in section 3.5.9. The ability to evaluate mathematical statements will be removed in future versions of SCOOT, as it is not tied to the core purpose of teaching OO principles.

3.5.15 Cancel

When programmers are entering a block of commands (either a method, a loop or a conditional) it is possible that they will either change their minds, or become aware of some error they have already made. In this case programmers can cancel the block entirely (methods will not be created, loops and conditionals will be erased) simply by entering the command:

```
Cancel;
```

This will be handled by the grammar:

```
CANCEL SEMI
```

When this grammar is triggered the value of *doingParse*, *looping*, *conditional* and *Input* are all reset, and control returned to the user. All input entered since beginning the current block is lost, and must be re-entered.

3.5.16 Quit

To quit SCOOT, and leave the environment the programmer simply enters:

```
Quit;
```

Which will be handled by:

```
QUIT SEMI
```

This will cause SCOOT to close.

3.6 Interface Design

In Section 2.4.2 Cognitive Load Theory (CLT) as it applies to a novice programming language was discussed. Based on this discussion a minimal interface for SCOOT was designed, including as few menus and buttons as possible, in order to minimise the extrinsic load of using the tool. This allows

more mental energy to be focused on the task of learning computer programming principles. As has been previously stated, this is in alignment with Papert's Continuity Principle.

Chapter 4

Experimental Design

4.1 Introduction

As discussed in Chapter Two the three core hypotheses being tested are :

H1 - SCOOT is seen by novice programmers as easier to use than C++, written in a customised version of Notepad++

H2 - SCOOT is seen by novice programmers as more useful than C++, written in a customised version of Notepad++

H3 - SCOOT is seen by novice programmers as more enjoyable than C++, written in a customised version of Notepad++

In order to test this the three distinct constructs of perceived ease-of-use, perceived usefulness and perceived enjoyment needed to be measured. These construct were chosen because SCOOT was designed with a focus on usability. By increasing usability it was our goal to reduce extraneous load so users can focus on learning programming, and not the tool. Minimising unnecessary learning lines up with Papert's Continuity Principle, that new material must build on old in predictable and expected ways. Confounding this with additional complexity is not ideal.

Testing of H1-H3 was conducted using a questionnaire which participants completed after they had completed a series of tasks using either C++ or

SCOOT. The questions were derived from constructs found in [Davis, 1989] and [Davis et al., 1992] which measure perceived usability, perceived usefulness and perceived ease-of-use. User responses were measured using a Visual Analogue Scale (VAS), which provides a rapid way to measure attitudes [Hoare, 1986], and has been used for over 80 years to measure subjective phenomena [Hayes, 1921]. A VAS is a 100mm long line, drawn on the page. In the questionnaire used for this study the line was horizontal, but vertical variations exist. Each line is associated with a statement, for example, "I find it easy to get C++ to do what I want it to do". The line is marked with "Strongly Disagree" at the left-most end and "Strongly Agree" at the right-most end. The centre point of the line is also identified, but not labelled. Participants then mark along the line to indicate their response, which is then measured to an accuracy of one millimetre, rounding down.

Three hypotheses related to learning outcomes are also being tested, but due to limitations of the experiment they are not central to the project. These three hypotheses are :

H4 - When given a simple programming task, users of SCOOT are able to write a solution more easily than users of C++.

H5 - When given a simple piece of code, users of SCOOT are able to understand it more easily than users of C++.

H6 - When given a piece of code with errors in it, users of SCOOT are able to identify and correct the errors than users of C++.

Each of these hypotheses are an attempt to gauge the users understanding of computer programming principles.

For H4 the phrase "more easily" means :

1. take less time to arrive at a correct result;

2. more users arrive at a correct result.

H5 was tested by asking questions about the code, so it is expected that SCOOT users will answer more questions correctly than C++ users.

H6 was tested by counting the number of errors correctly identified, and how many were incorrectly identified.

4.2 Design Decisions

4.2.1 Learning Programming

The first objective when testing the tool is to determine if students have learned anything about programming from their experience. It needs to be stated here that there are limitations on the ability to test this with real accuracy, due to the limited time available to conduct the research. This study was also limited due to the educational requirements of the department, as it is not possible to teach some students with the new tool, and observe their progress over several years compared to others who did not use it. However, it can be seen if a short experience with the tool instils some understanding of programming in a simple test environment.

To assess the degree to which a user has "learned" to program it was decided that they should complete three tasks :

- Code Writing;
- Code Reading;
- Debugging.

These three tasks we commonly used in written examinations of programming courses at the researchers university, and so the same strategies were used to measure comprehension of coding concepts here. As discussed in Section 2.4.1 the fact that a programmer can use a code construct may not indicate that they understand it. In an attempt to more accurately gauge their understanding participants also completed reading and debugging tasks, since it is in those tasks that weak understandings may be exposed [Rader et al., 1997, Lister et al., 2006a, Brown, 2008, Robins et al., 2003].

Due to the lack of run-time error handling is SCOOT there is no examination of run-time errors in this experiment. This weakness in SCOOT will be corrected in a future version, and an examination on the impact an introductory tool has on the management of run-time errors will be conducted.

4.2.2 The Experience

As described in Section 4.1, this dissertation is focusing on measuring the user's perceptions of both SCOOT and C++, specifically with regards to usability, ease-of-use and usefulness. A questionnaire modified from [Davis, 1989] and [Davis et al., 1992] was completed by all participants at the end of their respective sessions. In order to gather data to test H4-H6 data needed to be gathered during each task the participants completed. Time to complete each task was measured for all participants, and individual measures of correctness were taken for each task.

Writing Tasks For each writing task it was noted whether students had successfully completed the task or not. For those that had not their errors were examined, but no useful data could be gathered. To get useful data here it would be necessary to understand what the programmer is thinking, and trying to do, which is impossible when simply reading their code. A more formal forum for code discussion would be needed to try to learn anything from their errors, though recent work in [Kunkle and Allen, 2016] may change this in future work. It was finally decided to simply note which participants completed the tasks, and which did not.

Reading Task Measuring understanding in this task was done by presenting users with a series of questions about the value of variables at various points during execution. Assessing understanding with exam-style questions has been used elsewhere [Lister et al., 2006b] and this approach was followed here. We believe that this was an objective measure.

Debugging Task Three measures were taken for this task.

1. Number of errors found by each participant(False negatives);

2. Number of errors incorrectly found by each participant (False positives).

The second measure was needed since during the pilot study it was noted that participants would identify syntax errors in poorly written, but valid pieces of code, and this reflected a misunderstanding of what a syntax error actually was.

4.2.3 Compare to C++

In order to have a frame of reference for the data gathered from SCOOT users, it was necessary to be able to compare those results with users of a more commonly used language. Since the first wave of participants were recruited from the student population of the first year programming course, it was decided to use C++ as the language in the base group, since that is what all participants would be familiar with. C++ is an object-oriented language, and had been used as the introductory language in the first year courses at James Cook University for several years.

4.3 Participants

Participants were initially recruited from the first year university programming course at JCU Cairns. Testing a novice programming language on first-year university students is not uncommon [Dougherty, 2007, Lorenzen and Sattar, 2008, Powers et al., 2007, Garner, 2004]. It was decided to recruit at the start of the second semester, since students would have been introduced to the basics of programming at this point, but would not have learned any OO concepts.

However the number of participants recruited from this course was far less than expected, and so participants were recruited from the broader university population, and eventually from the general public. Even with this group of

participants numbers were far lower than desired, due (most likely) to the time required of participants.

Since most participants recruited had no computer programming experience it was necessary to give all participants an introduction to all aspects of OOP. This, along with a tutorial session to ensure all participants could use the programming environment, meant that a session took five or six hours, rather than the initially planned two.

There were two separate waves of testing. One with the IT students, and the second about six months later with students from the university at large, and some members of the general population. Within each wave, participants were broken into two groups: a test group, using SCOOT, and a control group using C++. Due to the time taken by each session, and the fact that all sessions were run by the principal researcher, each of these sessions ran on separate days.

Splitting each wave into test/control group was done first by availability. Any participant who was only available for one session was placed in that group. Remaining participants were then randomly assigned to the groups, in such a way as to keep numbers as close to equal as possible.

4.4 Environment

Within each wave of testing, both the C++ and SCOOT sessions were run in the same room, with the same facilities. Each user was on a separate computer, and was randomly assigned a number, which was used to identify all their work. Access to SCOOT was done over the Internet, using Java Web Start technology. Student's work in SCOOT was saved to local files on their computer, which were then collected by the primary researcher, and named with their number.

In order to reduce the impact of the difference between interfaces (with C++ being far more complex) a customised version of Notepad++ was used by C++ participants. This meant that both groups entered code, and then pressed a button to compile/execute. The modification to Notepad++ also

made backups of the users code every time they compiled, which were again collected and renamed by the primary researcher.

All recruitment, sessions and tutorials were conducted by the primary researcher.

4.5 Procedure

4.5.1 Tasks

While the individual sessions had differences, due to participant familiarity with programming, all groups completed the same set of tasks. There were two writing tasks, one building on the other, as well as a reading and debugging task. All tasks given to the participants are provided in Appendix A.

Writing Task One The first writing task called for participants to design and create two objects, each containing two or three attributes. Instructions about the names of the classes and attributes were given, as were their initial values. This writing task covered fundamental OOP ideas - Object creation and construction.

Writing Task Two The second writing task required participants to add methods to the classes created above. One method simply modified the value of an attribute, the second required an *if* statement before modifying a different attribute, and the third was required to use a loop to repeatedly execute the first method until a condition was met. Any participant who was unable to complete Task One was given a working solution at the beginning of Task Two. They were also given an explanation of how the provided solution worked, which elements went where and why. This task covered more complex OOP concepts, specifically methods and their construction. It also covered the core procedural skills necessary to build more useful objects - conditionals and iterations.

Code Reading Task The code reading task provided participants with a short program in the appropriate language, and a series of questions about the execution of the code. Participants were required to identify values of certain attributes at certain points during the program.

Code Debugging Task The final task presented participants with a piece of code, in the appropriate language, which contained syntax errors. Participants were required to identify lines with errors, state what the error was and suggest how to correct it by writing a correct line of code. The code given to both the SCOOT groups and the C++ groups contained the same number of errors (six).

4.5.2 Wave One - IT Students

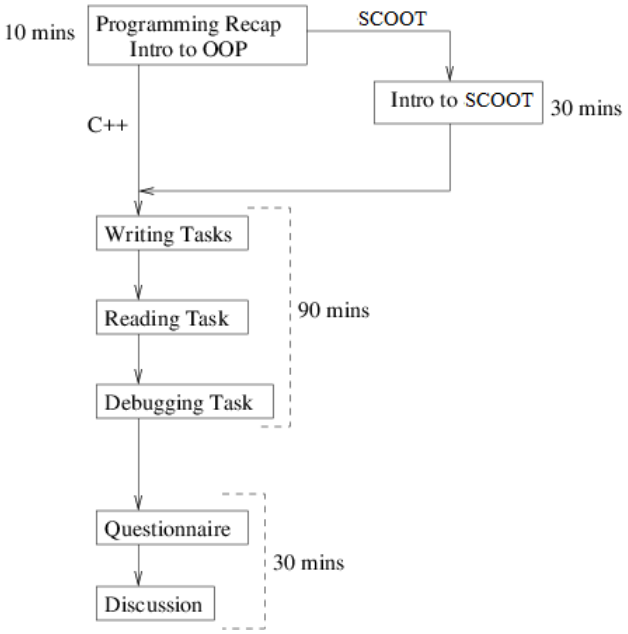


Figure 4.1: Wave One Session Flow.

Since all participants had a familiarity with programming concepts (including variables, functions, conditionals and iterations in C++), both sessions began with a quick (ten minute) recap of what they knew, and an introduction to OOP (30 minutes). Since students already knew C++ there was no need for

the base group to be introduced to the language, but the SCOOT group had never seen SCOOT before. The SCOOT group was therefore given a 30 minute introduction to SCOOT, being shown how to use the interface and how to perform the necessary tasks (object creation, definition, methods, loops and conditionals).

After this both groups completed the writing, reading and debugging tasks. Participants worked alone for these tasks. Once a participant had completed all tasks they completed a short questionnaire, and once all participants had completed the questionnaire a discussion group was held to discuss students experiences and thoughts. Task completion took around an hour and a half, and the questionnaire and discussion group was another 30 minutes.

4.5.3 Wave Two - General Population

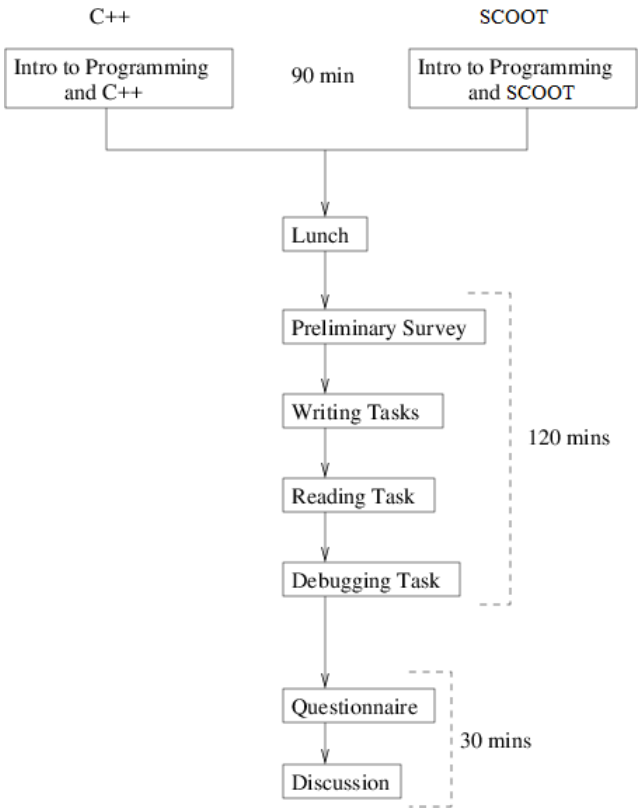


Figure 4.2: Wave Two Session Flow.

Since participants in this wave had little to no known programming knowledge a longer introduction session was needed in both the C++ and SCOOT groups. This session introduced the appropriate language/tool, as well as the basic concepts of computer programming. Each session lasted around an hour and a half. After that there was a break for lunch, which was provided by the school.

After lunch participants completed a short (three question) survey about their expectations before the tasks. They then completed the tasks as in wave one, including the final questionnaire and discussion group. Task completion took closer to two hours in this wave, with another 30 minutes or so for the final questionnaire and discussion group.

Chapter 5

Results

5.1 Introduction

Here the results of the testing process are presented. Firstly the results from the programming tasks are shown. This includes the writing, reading and debugging tasks. Secondly the responses to the questionnaire are given, including a reliability analysis and t-tests for each construct. Finally the correlations that were found between the tasks measures and the constructs are given, as well as the correlations between the results and the preliminary questionnaire. Within each of these sections the two waves of testing are presented separately, with Wave One being participants with programming experience, and Wave Two being those without. This is done as the two groups have very different levels of prior programming practice, and so they cannot be compared to each other.

5.2 Quantitative

As described in Chapter 4 the time taken to complete each task was recorded, as were measures of correctness. Due to the nature of the task some of these were entirely objective (for example, whether a participant completed the task) and some were highly subjective (for example, how close to "correct" a non-working

solution was). The more subjective observations will be discussed in Section 5.3, here only the objective measurements are discussed.

5.2.1 Writing Tasks

The time to complete (TTC) in minutes each of the two writing tasks was recorded, along with a count of how many people completed the task.

Wave One

Table 5.1: Mean Time In Minutes For Writing Tasks (Wave One)

Language		Writing Task 1	Writing Task 2
C++	Mean	54.11	24.56
	N	9	9
	Std. Deviation	8.536	10.737
SCOOT	Mean	11.13	27.53
	N	8	8
	Std. Deviation	5.083	8.279

Table 5.2: T-test Comparison For Writing Tasks (Wave One)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Writing Task 1	Equal variances assumed	12.398	15	0.000
	Equal variances not assumed	12.773	13.247	0.000
Writing Task 2	Equal variances assumed	-0.633	15	0.536
	Equal variances not assumed	-0.644	14.743	0.530

As can be seen from Tables 5.1 and 5.2, there is a significant difference between the C++ group and the SCOOT group for the time to complete task one, and the success rate for task one. A dramatic difference is seen in TTC for task one, with the C++ group having a mean time of 54 minutes, compared to the SCOOT group's 11 minutes. The t-test shown in Table 5.2 shows that this difference in groups is significant at the 99% confidence level. It should be noted that due to the design of SCOOT, there is far less overhead when creating objects when compared with C++. To create an object in SCOOT with three attributes,

Table 5.3: Mean Success Rate For Writing Tasks (Wave One)

Language		Completed Task 1	Completed Task 2
C++	Mean	0.22	0.44
	N	9	9
	Std. Deviation	0.441	0.527
SCOOT	Mean	1.00	0.50
	N	8	8
	Std. Deviation	0.000	0.535

Table 5.4: t-test Comparison for Writing Task Success Rate (Wave One)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Completed Task 1	Equal variances assumed	-4.971	15	0.000
	Equal variances not assumed	-5.292	8.000	0.001
Completed Task 2	Equal variances assumed	-0.215	15	0.832
	Equal variances not assumed	-0.215	14.712	0.832

accessors and mutators for each and assign values would take 7 lines of code, while the C++ code would be closer to 30 lines. It is unsurprising, therefore, that such a difference exists. It is also unsurprising that no such difference is evident for task two's TTC, since this task took pre-written objects and added behaviours. This task is similar in terms of number of lines needed for both C++ and SCOOT.

Of the 9 C++ participants, 2 completed the first writing task within the time given. The SCOOT group fared better, with all 8 participants completing the task. For the second writing task, 4 participants from each group completed in time. Assigning a binary measure of one for success, and zero for failure, the results shown in Tables 5.3 were produced. Table 5.4 again shows that the difference between groups for the first writing task is significant at the 99% confidence level.

It is interesting that a significant difference exists between C++ and SCOOT for the number of people to complete the first programming task, but not the second. This indicates that SCOOT enabled participants to create objects more easily, adding attributes and methods, while not significantly impacting their

ability to use more complex ideas such as conditionals or loop syntax.

Wave Two

Table 5.5: Mean Time In Minutes For Writing Tasks (Wave Two)

Language		Writing Task 1	Writing Task 2
C++	Mean	60.40	33.60
	N	5	5
	Std. Deviation	11.104	9.711
SCOOT	Mean	12.71	28.86
	N	7	7
	Std. Deviation	4.821	10.527

Table 5.6: T-test Comparison For Writing Tasks (Wave Two)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Writing Task 1	Equal variances assumed	10.239	10	0.000
	Equal variances not assumed	9.015	5.088	0.000
Writing Task 2	Equal variances assumed	0.793	10	0.446
	Equal variances not assumed	0.805	9.208	0.441

Tables 5.5 and 5.6 show that there is again a significant difference between the C++ group and the SCOOT group for both the time to complete task one, and the success rate for task one. The C++ group had a mean time of 60 minutes, while the SCOOT group took around 12. The t-test shown in Table 5.6 shows that this difference is significant at the 99% confidence level.

Of the 5 C++ participants, 2 completed the first writing task within the time given, and again all 7 SCOOT participants completed the task. For the second writing task, no C++ participants were successful, and 2 SCOOT participants completed in time. Assigning a binary measure of one for success, and zero for failure, the results shown in Tables 5.7 were produced. Table 5.8 again shows that the difference between groups for the first writing task is significant at the 99% confidence level.

It is interesting that a significant difference exists between C++ and SCOOT for the number of people to complete the first programming task, but not the second. This indicates that SCOOT enabled participants to create objects more

Table 5.7: Mean Success Rate For Writing Tasks (Wave Two)

Language		Completed Task 1	Completed Task 2
C++	Mean	0.40	0.00
	N	5	5
	Std. Deviation	0.548	0.000
SCOOT	Mean	1.00	0.33
	N	7	6
	Std. Deviation	0.000	0.516

Table 5.8: t-test Comparison for Writing Task Success Rate (Wave Two)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Completed Task 1	Equal variances assumed	-2.958	10	0.014
	Equal variances not assumed	-2.449	4.000	0.070
Completed Task 2	Equal variances assumed	-1.430	9	0.186
	Equal variances not assumed	-1.581	5.000	0.175

easily, adding attributes and methods, while not significantly impacting their ability to use more complex ideas such as conditionals or loop syntax.

5.2.2 Reading Task

The objective measure taken for all participants of the reading task was the time to complete.

Wave One

Table 5.9: Mean Time In Minutes For Reading Task (Wave One)

Language	Mean	N	Std. Deviation
C++	23.00	9	3.354
SCOOT	16.13	8	6.896

Table 5.10: T-test Comparison For Reading Task (Wave One)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Reading Task	Equal variances assumed	2.665	15	0.018
	Equal variances not assumed	2.563	9.872	0.028

As can be easily seen from Table 5.9 there is a difference of around 7 minutes between groups for the TTC the reading task. Table 5.10 shows that this difference is significant at the 95% confidence level. This indicates that SCOOT is more easily read and understood than C++. Measurement of success for the reading task were conducted differently within the two waves of testing, and were more subjective than desired for Wave One. For this reason the measures of participants understanding of the reading task for Wave One are discussed later in Section 5.3.2 as part of the qualitative results discussion.

Wave Two

Table 5.11: Mean Time In Minutes For Reading Task (Wave Two)

Language	Mean	N	Std. Deviation
C++	14.60	5	4.980
SCOOT	6.29	7	2.430

Table 5.12: T-test Comparison For Reading Task (Wave Two)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Reading Task	Equal variances assumed	2.665	15	0.018
	Equal variances not assumed	2.563	9.872	0.028

Table 5.11 indicates a difference of around 8 minutes between groups to complete the reading task. Table 5.12 shows that this difference is significant at the 95% confidence level. This again indicates that SCOOT is more easily read and understood than C++.

The second wave of participants were given a series of short questions, regarding the names or values of attributes at certain points during code execution. Assessing this was a simple matter of right/wrong for each question, with no room for subjective marking. Table 5.13 shows the mean number of questions correctly answered in each group, and Table 5.14 shows the t-test between languages. As can be seen, there was no significant difference found in participant's understanding of the code.

Table 5.13: Mean Number Of Questions Correctly Answered (Wave Two)

	Language	N	Mean	Std. Deviation	Std. Error Mean
CorrectReading	0	5	3.2000	1.64317	.73485
	1	7	4.1429	1.46385	.55328

Table 5.14: T-test Between Languages (Wave Two)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
CorrectReading	Equal variances assumed	-1.047	10	.320
	Equal variances not assumed	-1.025	8.088	.335

5.2.3 Debugging Task

In addition to TTC for the debugging task, the number of false negatives (errors missed) was recorded, as was the number of false positives (lines which participants flagged as incorrect, which in fact contained no errors).

Wave One

Table 5.15: Mean Time In Minutes And Measures For Debugging Task (Wave One)

Language		Debugging Task	False Negatives	False Positives
C++	Mean	20.44	5.67	1.67
	N	9	9	9
	Std. Deviation	3.909	2.345	1.658
SCOOT	Mean	8.00	1.13	0.63
	N	8	8	8
	Std. Deviation	2.000	1.126	1.061

Table 5.16: T-test Comparison For Debugging Task (Wave One)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Debugging Task	Equal variances assumed	8.093	15	0.000
	Equal variances not assumed	8.395	12.197	0.000
False Negatives	Equal variances assumed	4.978	15	0.000
	Equal variances not assumed	5.177	11.782	0.000
False Positives	Equal variances assumed	1.519	15	0.150
	Equal variances not assumed	1.559	13.734	0.142

Table 5.15 shows that the TTC was lower for the SCOOT group, who also had less false negatives and false positives. Table 5.16 shows that the difference in TTC and false negatives was significant at the 99% confidence level, however different languages do not seem to have significantly affected the number of false positives.

Wave Two

Table 5.17: Mean Time In Minutes And Measures For Debugging Task (Wave Two)

Language		Debugging Task Time	False Negatives	False Positives
C++	Mean	13.00	4.00	1.80
	N	5	5	5
	Std. Deviation	3.391	2.000	1.924
SCOOT	Mean	10.29	1.43	1.14
	N	7	7	7
	Std. Deviation	3.988	1.134	1.574

Table 5.18: T-test Comparison For Debugging Task (Wave Two)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Debugging Task Time	Equal variances assumed	1.233	10	0.246
	Equal variances not assumed	1.269	9.576	0.234
False Negatives	Equal variances assumed	2.852	10	0.017
	Equal variances not assumed	2.593	5.842	0.042
False Positives	Equal variances assumed	0.652	10	0.529
	Equal variances not assumed	0.628	7.583	0.548

Table 5.19: Reliability Analysis (Wave One)

Construct	Cronbach's Alpha
Perceived Usefulness	0.897
Perceived Ease of Use	0.886
Perceived Enjoyment	0.904

Table 5.20: Reliability Analysis (Wave Two)

Construct	Cronbach's Alpha
Perceived Usefulness	0.930
Perceived Ease of Use	0.862
Perceived Enjoyment	0.945

Table 5.17 shows that the TTC was slightly lower for the SCOOT group, who also had less false negatives and false positives. Table 5.18 shows that the difference in TTC and false positives was not significant, however the difference in false negatives was significant at the 95% confidence level.

5.2.4 Questionnaire

The questionnaire measures three constructs, as described in Chapter 4. Each was measured with three questions, each scored between 0 and 10 and Table 5.19 and 5.20 shows the reliability analysis for each construct measured in both waves.

The results of the reliability analysis indicate that all questions for a given construct are measuring the same thing, and this is consistent across both waves of participants.

Wave One

Table 5.21: Mean Results For Constructs (Wave One)

Language		Perceived Usefulness	Perceived Ease Of Use	Perceived Enjoyment
C++	Mean	7.0926	4.7667	5.9444
	N	9	9	9
	Std. Deviation	1.26435	2.15329	1.60158
SCOOT	Mean	7.0167	6.9584	5.6500
	N	8	8	8
	Std. Deviation	1.91219	1.60205	2.19884

Table 5.22: T-test Comparison For Constructs (Wave One)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Perceived Usefulness	Equal variances assumed	0.098	15	0.923
	Equal variances not assumed	0.095	11.922	0.926
Perceived Ease Of Use	Equal variances assumed	-2.354	15	0.033
	Equal variances not assumed	-2.397	14.597	0.030
Perceived Enjoyment	Equal variances assumed	0.318	15	0.755
	Equal variances not assumed	0.312	12.690	0.760

As shown in Table 5.21, only one of the three constructs measured (Perceived Ease of Use) was significantly higher for SCOOT than for C++. Both languages were rated similarly in the other two constructs. The difference in Perceived Ease of Use is significant at the 95% confidence level, as shown in table 5.22. This indicates that the usefulness of SCOOT and C++ are seen as roughly equivalent by participants, and that neither SCOOT not C++ was more "fun" to use. It does suggest that SCOOT was easier to use than C++, which was a major design goal during the development of SCOOT. A correlation analysis indicates some interesting links, as shown in Table 5.23.

Table 5.23: Correlation Between Task Performance And Constructs (Wave One)

		Perceived Usefulness	Perceived Ease Of Use	Perceived Enjoyment
Time for Writing Task1	Pearson Correlation	0.021	-.598*	0.013
	Sig. (2-tailed)	0.936	0.011	0.959
	N	17	17	17
Completed Task 1	Pearson Correlation	-0.019	.641**	0.096
	Sig. (2-tailed)	0.943	0.006	0.714
	N	17	17	17
Time for Writing Task 2	Pearson Correlation	-0.302	-.550*	-.550*
	Sig. (2-tailed)	0.238	0.022	0.022
	N	17	17	17
Completed Task 2	Pearson Correlation	-0.049	.570*	0.461
	Sig. (2-tailed)	0.853	0.017	0.063
	N	17	17	17
Reading Task	Pearson Correlation	0.456	0.140	0.302
	Sig. (2-tailed)	0.066	0.592	0.239
	N	17	17	17
Debugging Task	Pearson Correlation	0.182	-0.307	0.186
	Sig. (2-tailed)	0.484	0.231	0.475
	N	17	17	17
False Negatives	Pearson Correlation	-0.068	-.672**	-0.042
	Sig. (2-tailed)	0.795	0.003	0.874
	N	17	17	17
False Positives	Pearson Correlation	0.288	-0.115	-0.071
	Sig. (2-tailed)	0.261	0.660	0.787
	N	17	17	17
**. Correlation is significant at the 0.01 level (2-tailed).				
*. Correlation is significant at the 0.05 level (2-tailed).				

As can be seen from Table 5.23, there appears to be significant correlations between Perceived Ease of Use and all code writing activities. The idea that people could learn to write code more easily if a tool is easy to use was a significant part of the motivation behind the design of SCOOT. These results support that motivation.

Wave Two

Table 5.24: Mean Results For Constructs (Wave Two)

Language		Perceived Usefulness	Perceived Ease of Use	Perceived Enjoyment
C++	Mean	6.6000	4.2400	6.3533
	N	5	5	5
	Std. Deviation	2.83892	1.92624	2.96607
SCOOT	Mean	5.6944	6.4971	5.7762
	N	6	7	7
	Std. Deviation	0.95415	1.57440	1.98529

Table 5.25: T-test Comparison For Constructs (Wave Two)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Perceived Usefulness	Equal variances assumed	0.740	9	0.478
	Equal variances not assumed	0.682	4.755	0.527
Perceived Ease of Use	Equal variances assumed	-2.236	10	0.049
	Equal variances not assumed	-2.156	7.578	0.065
Perceived Enjoyment	Equal variances assumed	0.406	10	0.693
	Equal variances not assumed	0.379	6.524	0.717

As shown in Table 5.24, only one of the three constructs measured (Perceived Ease of Use) was significantly higher for SCOOT than for C++. Both languages were rated similarly in the other two constructs. The difference in Perceived Ease of Use is significant at the 95% confidence level, as shown in table 5.25. These results are consistent with those of wave one.

Table 5.27: Preliminary Questionnaire Correlations (Wave Two)

		Time for Writing Task 1	Time for Writing Task 2	Reading Task	Debugging Task	Perceived Usefulness	Perceived Ease Of Use	Perceived Enjoyment
ExpectedEnjoyment	Pearson Correlation	.097	.014	.051	.768**	.343	.285	.225
	Sig. (2-tailed)	.777	.967	.882	.006	.332	.396	.506
	N	11	11	11	11	10	11	11
ExpectedToCompleteQuickly	Pearson Correlation	-.435	.049	-.192	.301	.630	.743**	.202
	Sig. (2-tailed)	.181	.885	.571	.368	.051	.009	.551
	N	11	11	11	11	10	11	11
ExpectedInterest	Pearson Correlation	.107	-.089	.052	.557	.589	.523	.663*
	Sig. (2-tailed)	.753	.795	.880	.075	.073	.099	.026
	N	11	11	11	11	10	11	11

**. Correlation is significant at the 0.01 level (2-tailed).*. Correlation is significant at the 0.05 level (2-tailed).

Table 5.26: Correlation Between Task Performance And Constructs (Wave Two)

		Perceived Usefulness	Perceived Ease of Use	Perceived Enjoyment
Writing Task 1 Time	Pearson Correlation	0.002	-.735**	-0.087
	Sig. (2-tailed)	0.996	0.006	0.788
	N	11	12	12
Completed Task 1	Pearson Correlation	0.482	.776**	0.432
	Sig. (2-tailed)	0.133	0.003	0.160
	N	11	12	12
Writing Task 2 Time	Pearson Correlation	0.128	-0.488	-0.073
	Sig. (2-tailed)	0.707	0.108	0.821
	N	11	12	12
Completed Task 2	Pearson Correlation	-0.079	.729*	0.293
	Sig. (2-tailed)	0.829	0.011	0.381
	N	10	11	11
Reading Task Time	Pearson Correlation	0.252	-0.463	-0.139
	Sig. (2-tailed)	0.455	0.130	0.668
	N	11	12	12
Debugging Task Time	Pearson Correlation	.638*	-0.079	0.263
	Sig. (2-tailed)	0.035	0.807	0.410
	N	11	12	12
False Negatives	Pearson Correlation	-0.325	-.732**	-0.327
	Sig. (2-tailed)	0.329	0.007	0.300
	N	11	12	12
False Positives	Pearson Correlation	-0.284	-0.486	-0.068
	Sig. (2-tailed)	0.398	0.109	0.834
	N	11	12	12
**. Correlation is significant at the 0.01 level (2-tailed).				
*. Correlation is significant at the 0.05 level (2-tailed).				

Table 5.26 show that, as was found in wave one, there appears to be significant correlations between Perceived Ease of Use and all code writing activities.

In wave two a short (three question) questionnaire was administered to

participants before the programming tasks. This was done to find a link between expectations and experience. As can be seen in Table 5.27 there were significant correlations found between a participant's expectation that they would complete the task quickly, and the success rate for the first programming task. The same expectation was also linked to their Perceived Ease of Use, which has already been linked to the success rate for all code writing tasks. It should also be noted that Expected Interest is linked to Perceived Enjoyment. Though this has not been linked to performance, it does improve the users experience. While it lies out of the scope of this project, it seems worth investigating the link between the user expectations of a language or tool, and their learning experience. If a link is found then a core part of all early programming curricula should be designed to improve student expectations.

It should be noted here that Expected Enjoyment was positively correlated to Debugging Time, which indicated that participants who expected to enjoy the task, generally took longer to complete the debugging task.

5.3 Qualitative

5.3.1 Writing Tasks

In the first round of testing it was intended to assess the correctness of code written during the writing tasks. This proved to be an essentially subjective task, however, and so this went no further than the initial round of testing. The fundamental problem encountered was one of determining the "cause" of an error. It was not possible to objectively measure why a participant had made a particular error, and so the results of this analysis were not useful. In the end it was decided to simply record the number of participants who completed the tasks, and those who did not.

Table 5.28: Demographic Data (Wave One)

		Gender		English Experience		Computer Use		
		Male	Female	First Language	Additional Language	Occasional	Moderate	Frequent
Language	C++	7	2	8	1	0	1	8
	SCOOT	4	4	5	3	1	2	5

Table 5.29: Mean Intent to Continue Using Or Recommend (Wave One)

Language		Continue Using	Recommend for Uni	Recommend for High School
C++	Mean	6.644	7.900	6.456
	N	9	9	9
	Std. Deviation	2.3060	1.9248	3.3295
SCOOT	Mean	6.425	7.588	8.788
	N	8	8	8
	Std. Deviation	3.5176	3.1266	1.1332

5.3.2 Reading Task

The first wave of participants were given a short piece of code, and asked to describe its function. This proved insufficient for any meaningful comparison, since some participants answered in such a way as to suggest they understood the function of the code, but without being explicit. It was also difficult to reliably measure the "correctness" of their response in some cases, and to determine whether some errors of understanding were more significant than others.

5.3.3 Demographic Trends

In addition to the survey questions focusing on the constructs that were measured, background demographic data on the participants was also collected.

Wave One

As can be clearly seen in Table 5.28, there were no significant differences in the general make up of the two groups used in the testing process. Both groups had similar breakdowns by gender, familiarity with English and with computers. No correlations of any kind were found between these results and any other collected data. The final stage of the questionnaire asked participants if they planned to continue using the language/tool they had used, and if they would recommend it to high school or university students.

Table 5.30: T-test Comparison For Intent Or Recommendation (Wave One)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Continue Using	Equal variances assumed	0.154	15	0.880
	Equal variances not assumed	0.150	11.856	0.883
Recommend for Uni	Equal variances assumed	0.252	15	0.805
	Equal variances not assumed	0.244	11.381	0.811
Recommend for High School	Equal variances assumed	-1.881	15	0.080
	Equal variances not assumed	-1.976	10.026	0.076

Table 5.31: Demographic Data (Wave Two)

		Gender		English Experience		Computer Use		
		Male	Female	First Language	Additional Language	Occasional	Moderate	Frequent
Language	C++	2	3	5	0	4	1	0
	SCOOT	2	5	7	0	5	2	0

Table 5.32: Mean Intent to Continue Using Or Recommend (Wave Two)

Language		Continue Using	Recommend for Uni	Recommend for High School
C++	Mean	7.020	6.140	5.020
	N	5	5	5
	Std. Deviation	2.6762	3.2708	3.4369
SCOOT	Mean	6.657	7.957	8.457
	N	7	7	7
	Std. Deviation	2.6133	1.7491	1.1646

As can be seen in Table 5.29 there were more participants who would recommend SCOOT for High School students, than C++. This difference is significant at the 95% confidence level. No other significant differences in recommendations were found.

Wave Two

As can be clearly seen in Table 5.16, there were no significant differences in the general make up of the two groups used in the testing process. Both groups had similar breakdowns by gender, familiarity with English and with computers. No correlations of any kind were found between these results and any other collected data. The final stage of the questionnaire asked participants if they planned to continue using the language/tool they had used, and if they would recommend it to high school or university students.

Table 5.33: T-test Comparison For Intent Or Recommendation (Wave Two)

		t-test for Equality of Means		
		t	df	Sig. (2-tailed)
Continue Using	Equal variances assumed	0.235	10	0.819
	Equal variances not assumed	0.234	8.634	0.821
Recommend for Uni	Equal variances assumed	-1.255	10	0.238
	Equal variances not assumed	-1.132	5.644	0.303
Recommend for High School	Equal variances assumed	-2.494	10	0.032
	Equal variances not assumed	-2.150	4.662	0.088

As can be seen in Table 5.17 there were more participants who would recommend SCOOT for High School students, than C++. This difference is significant at the 95% confidence level. No other significant differences in recommendations were found.

5.3.4 Focus Groups

After each session there was a short discussion time, during which participants were free to offer any opinions, suggestions or thoughts about their experiences with the languages. Eight main questions were asked of each group, with room at the end for any other thoughts, and they are discussed here.

Wave One

What was interesting? Participants in wave one found nothing in particular interesting with either language. Some of the SCOOT participants did find the syntax of SCOOT a helpful change from a more formal language, but otherwise not much was mentioned here.

What was fun? Participants familiar with C++ who were using SCOOT found it fun to see another language, particularly one with such a different syntax. Those participants using C++ who were already familiar with it did not find any of the tasks fun. Most expressed the view that they felt that they were being tested, not the language, despite care taken by the researchers to

clarify this point. This may be linked to another point raised in the last part of this section.

What was easy? Most participants in both groups found the reading task easier than writing, saying that it was easier not to have to think of what to do, but just to read code and think it through. Participants in the SCOOT group said that the first writing task was quite easy, but that the difficulty jumped too high for the second task. Participants in most groups reported that the handouts given (showing simple examples of code) were very useful, and made writing code easier. These findings are as expected, given similar results in [Lister et al., 2006a].

What was difficult? Without a doubt, this question was the most discussed question across both groups of participants. The first round of C++ participants found that they couldn't remember basic programming concepts, even with their programming experience. Even though they had access to an outline of syntax examples (like all groups) they felt, for some reason, that they were being tested, and that they needed to complete the tasks without any help. In addition to this mentality, they also felt that the task should be harder than it was (thinking "create a class... surely it's not that simple") and so they added complexity that only confused themselves.

There was also a desire amongst this group to ensure that what they created was "useful", which none of the simple tasks really were. This meant that a participant asked to create and test an employee class would not create a simple test program (instantiate some employees, add data, test functions) but would instead try and create a fully functional employee management system, including pay systems, promotions and so on, which was far outside the scope of what was asked.

Participants with C++ experience who used SCOOT, on the other hand, found syntactic rules (like which keywords are capitalised) the trickiest thing to learn. Participants also reported some difficulty in remembering which blocks to

close and in what order. SCOOT requires programmers to end different blocks (method, conditional, loop) with different commands. In C++ they simply put a close brace, and the compiler closes the appropriate block. They were not used to having to explicitly think about the block structure of their programs. They also found the inability to edit code frustrating, which is a limitation of the SCOOT editor.

Was there a "light bulb moment", and what was it? Neither the C++ nor the SCOOT participants in the first round of testing reported anything like this.

What was frustrating? C++ participants found the whole task frustrating, specifically citing classes as being "too hard" and error messages as "useless". Participants using SCOOT found the inability to edit a block of code (such as a method or a loop) frustrating, since being able to fix a minor typo would save rewriting several lines of code. SCOOT participants also reported some difficulty identifying the relationship between inputs and outputs in the code history.

Any other suggestions/thoughts? C++ participants seemed to find the idea of "fixing" or "improving" C++ confusing. There was a prevailing attitude amongst them that C++ was "good" and that any difficulty they had in using it was their failing. They also said that "programming should be hard". While this certainly provides a sense of achievement once a person can program, it may in fact be a hurdle to novice programmers. This issue lies in the field of programming curriculum design, however, and is far outside the scope of this project.

SCOOT participants suggested that inputs be more clearly linked to outputs in the code history, and that they be able to save, load and edit their code.

Wave Two

What was interesting? Participants experiencing C++ for the first time found the whole idea of programming interesting, as it gave them the chance to see "behind the screen". Participants seeing SCOOT with no prior programming experience also found it interesting to see how computers worked, and to find that programming was not as difficult as they had thought.

What was fun? Students seeing C++ for the first time found the process of writing code which worked very satisfying, though they weren't sure it could be called fun. New programmers using SCOOT found the whole process enjoyable. They found the act of writing programs exciting, and found it fun to see how programs could be written. They also said it was good to see that programming was not "just for geeks" and that computers were not "very clever boxes". Some students pointed out that seeing a tool like SCOOT helped remove the "magic" around the computer.

What was easy? Most participants in both groups found the reading task easier than writing, saying that it was easier not to have to think of what to do, but just to read code and think it through. As with Wave One, participants using SCOOT said that the first writing task was quite easy, but that the difficulty jumped too high for the second task. Participants in both groups reported that the handouts given (showing simple examples of code) were very useful, and made writing code easier, just as was reported by Wave one participants.

What was difficult? Again, this question was the most discussed question across both groups of participants.

The C++ participants, who had no prior programming experience, reported very different issues to those in wave one. This group found it hard to complete tasks in the time allocated, and found the syntax of C++ very difficult to

understand. The most confusing things for this group were reported to be identifying keywords, and remembering when to use semi-colons.

The second round of SCOOT participants reported the most difficulty in knowing which commands to use in which circumstances, and in completing more complex tasks, such as having a conditional block inside a method being added to an object.

It is worth remembering that both groups (C++ and SCOOT) in this wave of testing had no prior programming experience, and were shown object creation/manipulation, methods, loops and conditionals in a single 5-6 hour session. A similar amount of material would be covered in 5-6 weeks of an introductory programming course.

Was there a "light bulb moment", and what was it? The C++ participants said they felt like they were getting there, but no-one felt that they had "got it". SCOOT participants reported that they thought they had it early on, during the first task, but then lost confidence during the second task which got much harder much faster than task one.

What was frustrating? C++ participants found that they did not have the time to learn what they needed to in order to feel confident completing the tasks. Participants using SCOOT found the inability to edit a block of code (such as a method or a loop) frustrating, as was found in Wave One. SCOOT participants also reported some difficulty identifying the relationship between inputs and outputs in the code history.

Any other suggestions/thoughts? The C++ participants suggested that case-sensitivity be removed, and that compiler error messages be more useful to beginners. just as with Wave One, SCOOT users suggested that inputs be more clearly linked to outputs in the code history, and that they be able to save, load and edit their code.

Chapter 6

Discussion

6.1 Summary and Outcomes

Learning to program is a difficult task, and while this task cannot necessarily be made easy with a well designed language or IDE, it can certainly be made harder with a bad one. Cognitive Load Theory (CLT) [Chalmers, 2003] suggests that it is necessary to remove as much extrinsic load as possible, freeing novice programmers to focus on learning principles of programming rather than an interface and a language. While there are many languages designed to be easy to use, and many OO languages, a need for an OO programming language, explicitly designed to be a stepping stone to other languages, was identified.

SCOOT was designed as a teaching tool to meet this need. SCOOT commands are English-like in syntax, while still requiring users to follow simple syntactic rules, similar to those found in other languages. It is interpreted, similar to Python, which allows users to build classes dynamically, adding and removing elements on the fly. In order to test the ways in which SCOOT users learned to program it was compared to C++ in a series of tasks including code reading, debugging and writing. Additionally a short questionnaire and focus group was conducted, to assess the experience of using the language or tool.

This first step in testing the hypotheses was to develop a working prototype

of SCOOT, and in order to do that the design principles needed to be firmly established. A survey of the literature indicated that while many languages had been developed to teach programming, most seemed to either redefine programming to remove code altogether (as in Alice 2.0, SCC) or to create a language that, whilst easy to learn, attempted to be a full language in its own right, rather than a stepping stone to other languages. Only one language found (GRAIL) followed a similar design approach to that developed for SCOOT. GRAIL's developers focused on creating a language that would maintain a textual interface and focus on being (in their words) a "zeroth" language, as a stepping stone to other languages.

When discussing the design of a new tool, however, it is not enough to simply discuss the commands. A key component of any teaching tool is the environment used by the learners. Some languages, such as C++ and Java, have multiple environments, which can be tailored to suit the needs of their users. Other languages, like Alice and SCC have a single interface, with less customisability. Since the goal of SCOOT was to merely be a teaching tool acting as a stepping stone to other languages it was decided to create an interface with minimal clutter, and very limited menus. There would be a single field for code entry, a panel showing the history of entered code, and a second panel showing the results of those commands. A single button would be used to execute code which had been entered. By keeping the interface as simple and minimalistic as possible, the extrinsic load on the task of programming would be significantly reduced.

The programming language GRAIL has a very similar approach in terms of language design. However, there is one significant area in which SCOOT differs from the design of GRAIL, and that is the introduction of objects. GRAIL's developers do not support the early introduction of objects, but based on Papert's Continuity Principle it is the argument of this dissertation that moving objects to the beginning of a programmers learning makes later transitions easier. For this reason SCOOT differs significantly from GRAIL in its approach

to programming and language structures. However, in terms of language design principles SCOOT's approach was very similar. Those principles, discussed in detail in Section 2.5, are :

- Predictability;
- Familiarity;
- Simplicity.

Once the design principles were in place a working prototype of SCOOT was developed for testing. It was then necessary to design an experiment which could test the hypotheses. An introductory seminar was designed to teach participants the basics of object-oriented programming, using either SCOOT or C++ to demonstrate the concepts. After this session participants worked through two programming tasks, as well as a reading and debugging task. Their performance, and the time to complete the tasks were measured. After this the participants completed a short questionnaire which used three constructs to measure their experience. These three constructs align with the three core hypotheses introduced in Chapter Two. The times to complete each task and the performance of the individual tasks were used to measure the three secondary hypotheses.

6.2 General Conclusions

6.2.1 Quantitative Observations

Wave One

On average participants took significantly less time to complete the first programming task, as well as the reading and debugging task, when using SCOOT rather than C++. Once participants moved on to the second programming task, however, this difference faded. For the first programming task the SCOOT group averaged eleven to twelve minutes, while the C++

group averaged over fifty-five minutes. Additionally only four of the fourteen C++ participants were able to complete the first writing task, while all fifteen SCOOT participants were able to complete it. As mentioned already, the time difference for the second programming task was not significant, with both groups taking around twenty-eight minutes to complete. It should be noted, however, that only four of fourteen C++ users completed task two, while seven of fifteen SCOOT users were able to complete this task.

SCOOT users also took significantly less time to complete the reading task, with the average participant taking eleven to twelve minutes, compared to twenty minutes for the C++ group. As was discussed in Chapter 4, only the last group of participants were discreetly measured during the reading task. Each group answered five questions, but no significant differences were found in the correctness of the answers, with the C++ group averaging 3.2 and the SCOOT group averaging 4.1.

Finally there was also a significant time reduction for SCOOT users in the debugging task, as well as a significant increase in success at finding errors. The SCOOT participants averaged just over nine minutes to complete the task, typically missing one or two errors. C++ users, on the other hand, took almost eighteen minutes (on average) to complete the task, and typically missed around five errors. In addition to measuring time to complete and the number of errors found, the number of errors incorrectly identified (that is valid code which participants identified as invalid) was also measured. There was no significant difference between the two groups when it came to incorrectly identifying errors.

Wave Two

On average participants took significantly less time to complete the first programming task, as well as the reading and debugging task, when using SCOOT rather than C++. Once participants moved on to the second programming task, however, this difference faded. For the first programming task the SCOOT group averaged eleven to twelve minutes, while the C++

group averaged over fifty-five minutes. Additionally only four of the fourteen C++ participants were able to complete the first writing task, while all fifteen SCOOT participants were able to complete it. As mentioned already, the time difference for the second programming task was not significant, with both groups taking around twenty-eight minutes to complete. It should be noted, however, that only four of fourteen C++ users completed task two, while seven of fifteen SCOOT users were able to complete this task.

SCOOT users also took significantly less time to complete the reading task, with the average participant taking eleven to twelve minutes, compared to twenty minutes for the C++ group. As was discussed in Chapter 4, only the last group of participants were discreetly measured during the reading task. Each group answered five questions, but no significant differences were found in the correctness of the answers, with the C++ group averaging 3.2 and the SCOOT group averaging 4.1.

Finally there was also a significant time reduction for SCOOT users in the debugging task, as well as a significant increase in success at finding errors. The SCOOT participants averaged just over nine minutes to complete the task, typically missing one or two errors. C++ users, on the other hand, took almost eighteen minutes (on average) to complete the task, and typically missed around five errors. In addition to measuring time to complete and the number of errors found, the number of errors incorrectly identified (that is valid code which participants identified as invalid) was also measured. There was no significant difference between the two groups when it came to incorrectly identifying errors.

6.2.2 Qualitative Observations

Observation during these sessions suggested that people found SCOOT much less frustrating and complex than C++, an observation supported by participant responses in the survey. A common issue observed in C++ users, and not in SCOOT users, was a tendency to overcomplicate a task. In one notable example participants were to write a class method which increased a class attribute by

500. One particular C++ participant wrote a function which prompted the user for the amount to increase by, and read that value from the keyboard. This input value was then compared to 500, and if it was different an error message appeared and the program asked again. If the user entered 500 then that value was added to the internal class attribute. In this example a function which should have been one line long, became nine lines of code, including a loop and a conditional.

6.2.3 Discussion

Wave One

SCOOT Participants

The first round of participants for both SCOOT and C++ were drawn from the first year programming course, which taught C++. In discussion with this group there was a prevailing attitude that SCOOT didn't really feel like programming, because "programming should be difficult". This attitude was unanimous within the group, as was the attitude that any problems they had using C++ were their fault, because C++ was good, and therefore couldn't be improved.

Participants found that the connection between what they wanted to do and how to do it seemed simpler with SCOOT than with C++. Feedback during the survey and the discussion after the session also supported this observation. Several participants reported that the command structures of SCOOT made it easy to know when to use certain constructs, while others noted the opposite issue with C++.

Some participants in the SCOOT group found the syntax requirements of SCOOT frustrating in the early stages of the workshop. Case sensitivity and the semi-colon at the end of a command were singled out as particularly irritating. Each line in SCOOT is entered and evaluated individually, so participants had to correct each line as they went. This was singled out as exceptionally irritating in the early tasks. Interestingly, this irritation led to participants self-checking

their code before entering it, and not relying on the SCOOT environment to fix their mistakes for them. This is the opposite of what was seen in the C++ groups, where people would simply compile first and then fix errors as they were told about them. This self-checking was mentioned during the discussion by several students as being a good habit, and those with some programming experience saw the benefits of reading their own code before running it.

A common request from SCOOT users was for a clearer visual link between inputs and outputs. A suggestion from the developers, which seemed to please most participants, was to replace the current two-window display with a single window, showing input commands in bold, with the response output indented and in normal font below it. This design change will be implemented in a later version of SCOOT.

A more critical issue raised by many SCOOT users was the inability to edit a block of code, such as a class method or loop. In the current version of SCOOT, if an error is found in a class method, then the method must be removed, and re-entered entirely. The initial vision for SCOOT was only for small scale projects, so this was not seen as an issue of critical importance, however it is apparent from the testing process that reducing novice frustration is important to improving their learning experience. Therefore a future version of SCOOT will allow for blocks of code, including loops, conditionals and class methods, to be edited after they have been created. A related issue was the inability to save or load a program once it was written. This is due to the nature of the SCOOT environment, but inclusion of this feature will be looked at for a later revision of the environment.

A smaller request from participants, which will be implemented in a future revision of SCOOT, is to improve the window displaying the names of all objects currently in the system. In a future revision, clicking on one of these will display a list of its attributes and methods.

C++ Participants

As discussed above, those participants drawn from the programming course did not offer any critical feedback on C++ as a language. Most people also found the error messages unhelpful, since they were too technical and confusing. This is likely to be a common problem when using any commercial grade language as a novice language. Error messages in these environments are written for programmers, and not for novices. While a case can certainly be made that understanding those error messages improves ones understanding of a programming language's internal behaviour, it seems a lot to add to the burden of a novice programmer.

Wave Two

SCOOT Participants

Most participants reported that the command structures of SCOOT made it easy to know when to use certain constructs, however many found the syntax requirements of SCOOT frustrating in the early stages of the workshop. Case sensitivity and the semi-colon at the end of a command were singled out as particularly irritating.

As with Wave One, SCOOT users requested a clearer visual link between inputs and outputs, as well as the ability to save and load their programs. This has been discussed above.

C++ Participants

C++ users from non-programming backgrounds found several issues during their time with it. Firstly, people were frustrated that a simple error (such as missing a semi-colon, or spelling a variable name incorrectly) would not be picked up until they attempted to compile their code. This is an interesting reversal of the attitude displayed by SCOOT participants regarding the identification of simple errors immediately.

Participants also complained that simply getting started creating an object took a long time (around ten minutes) before they were even able to begin compiling and testing to see where they had gone wrong. Most people also found the error messages unhelpful, since they were too technical and confusing. This has already been discussed.

Common Findings

6.2.4 Hypotheses

As has been mentioned, the three primary hypotheses are :

- H1 - SCOOT is seen by novice programmers as easier to use than C++, written in a customised version of Notepad++ (supported).
- H2 - SCOOT is seen by novice programmers as more useful than C++, written in a customised version of Notepad++ (not-supported).
- H3 - SCOOT is seen by novice programmers as more enjoyable than C++, written in a customised version of Notepad++ (not-supported).

and the secondary hypotheses are :

- H4 - When given a simple programming task, users of SCOOT are able to write a correct solution more quickly than users of C++ (supported).
- H5 - When given a simple piece of code, users of SCOOT are able to correctly understand it more quickly than users of C++ (supported).
- H6 - When given a piece of code with errors in it, users of SCOOT are able to identify and correct the errors more quickly than users of C++ (supported).

As shown in Table 5.11 SCOOT was seen by students as significantly easier to use than C++, but neither construct relating to H2 or H3 showed a significant difference based on the language or tool used. This suggests that while SCOOT's design led to a set of commands which is easier to use, there is no difference

in novice perceptions of usefulness or enjoyment. As has been shown, however, SCOOT users performed significantly better than C++ users during the simpler programming task. This trend continued for the reading task, with a significant reduction in the time to complete the task for users of SCOOT. Similarly, users of SCOOT were able to find more errors, and find them faster, than users of C++ during the debugging task. However no significant difference in completion time was found for the more complex programming task.

Significant correlations were found between perceived ease of use and success in all code writing tasks. This suggests a correlation between improving ease of use in a novice programming tool and the ability of learners to write correct code. Further work must be done here to determine if such a correlation exits, and it is expected that this will be examined as part of a long term study on the impact of SCOOT.

6.3 Future Work

In this work a prototype of SCOOT was evaluated, and several critical improvements that need to be made were identified. There are some changes that need to be made to the structure of SCOOT commands, especially improvements to ending a method, and of more explicitly linking a comparison to the branch/iteration that it affects. The most critical change that must be made to SCOOT is to add the ability to edit code after it has been entered. This weakness is a remnant of the early concept of SCOOT, allowing for immediate feedback on code as it was entered. As the idea for SCOOT expanded to include branches and iterations, it has become clear that this is an essential feature that must be included in any future revision of SCOOT.

As a result of the inclusion of code editing capabilities SCOOT will also need to handle runtime errors, and provide meaningful feedback to users in this regard. Current SCOOT errors are directly related to the line of code being evaluated, and are limited to syntax errors only. Runtime errors must provide a

similar level of detail and clarity in order to aid users in finding and correcting these errors.

Other than modifications to SCOOT itself there are several avenues of research which should be investigated as a result of this work. A long term study would allow for the introduction of programming concepts at a more sedate pace, and thus would more accurately measure the impact of SCOOT on advanced programming concepts. It would also be interesting to evaluate participant understanding of programming using the work given in [Kunkle and Allen, 2016].

The possibility of a psychological link has already been suggested, resulting in complex code because the language seems complex. This may be an interesting direction for future research into the teaching and learning of computer programming languages.

Another interesting direction for future research would be into whether the attitude shown by the programming students (that C++ is innately good and that programming should be hard) is commonly held amongst people who enrol in programming courses, or if this attitude is instead encouraged by introductory programming courses.

The possibility of a positive correlation between student experience of a programming environment and their learning outcomes has already been mentioned, and further work should be done to determine if such a link exists.

Finally, this study suggests that a language designed to be easier to use for novices has a significant positive impact on the learning experience, with SCOOT users performing faster and with more accuracy on almost all tasks. More research into the link between ease-of-use of a learning tool, and learning outcomes may illuminate this connection for future work in the field of novice programming languages.

6.4 Final Conclusions

In searching the literature in the field of novice programming languages, the need for an OO novice programming teaching tool, which used a textual language for code entry, became apparent. To fulfil this need SCOOT, the Student Centric Object-Oriented Teaching-tool was developed. In testing this tool it became clear that users generally found it easier to use SCOOT than an alternative language, C++. Additionally SCOOT users were able to complete code reading, debugging and simple writing tasks faster, and with more success than C++ users. While a long term, large scale study will clarify the nature of the link between ease-of-use and programming success, it seems that a tool designed to be easier to use is more successful at teaching, and reinforcing, common programming concepts and practices. The intent of this research project was to examine the effectiveness of a textual OO language, with a simple, consistent syntax, designed to lead into other languages. This research has shown that textual languages are still essential for novice programmers, and that a simple syntax can help novices learn programming principles without unnecessary overheads.

Bibliography

[Adams, 2007] Adams, J. C. (2007). Alice, middle schoolers & the imaginary worlds camps. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 307–311, Covington, Kentucky, USA. ACM.

[Andersen et al., 2016] Andersen, S. A. W., Mikkelsen, P. T., Konge, L., Cayé-Thomasen, P., and Sørensen, M. S. (2016). The effect of implementing cognitive load theory-based design principles in virtual reality simulation training of surgical skills: a randomized controlled trial. *Advances in Simulation*, 1(1):20.

[Backus and Heising, 1964] Backus, J. W. and Heising, W. P. (1964). Fortran. *IEEE Transactions on Electronic Computers*, EC-13(4):382–385.

[Ben-Ari, 2011] Ben-Ari, M. M. (2011). Moocs on introductory programming: a travelogue. *ACM Inroads*, 4(2):58–61.

[Black et al., 2010] Black, A., Bruce, K. B., and Noble, J. (2010). Panel: designing the next educational programming language. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 201–204. ACM.

[Blackwell and Bilotta, 2000] Blackwell, A. F. and Bilotta, E., editors (2000). *The Effect of Programming Language on Error Rates of Novice Programmers*, number Corigliano Calabro, Italy.

[Bloch, 2003] Bloch, S. (2003). Teaching linked lists and recursion without conditionals or null. In *Journal of Computing in Small Colleges*, volume 18.

[Bloch, 2000] Bloch, S. A. (2000). Scheme and java in the first year. In *Journal of Computing in Small Colleges*, volume 15, pages 157–165.

[Braught and Wahls, 2008] Braught, G. and Wahls, T. (2008). Teaching objects in context. *Journal of Computing in Small Colleges*, 23(5):101–109.

[Brown, 2008] Brown, P. H. (2008). Some field experience with alice. *Journal of Computing in Small Colleges*, 24(2):213–219.

[Calisir and Calisir, 2004] Calisir, F. and Calisir, F. (2004). The relation of interface usability characteristics, perceived usefulness, and perceived ease of use to end-user satisfaction with enterprise resource planning (erp) systems. *Computers in Human Behavior*, 20(4):505 – 515.

[Caspersen and Kolling, 2009] Caspersen, M. E. and Kolling, M. (2009). Stream: A first programming process. *ACM Transactions on Computing Education (TOCE)*, 9(1):4.

[Chalmers, 2003] Chalmers, P. (2003). The role of cognitive theory in human-computer interface. *Computers in Human Behavior*, 19(5):593–607.

[Chao, 2016] Chao, P.-Y. (2016). Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. *Computers & Education*, 95:202 – 215.

[Cheung et al., 2009] Cheung, J. C., Ngai, G., Chan, S. C., and Lau, W. W. (2009). Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students. In *ACM SIGCSE Bulletin*, volume 41, pages 276–280. ACM.

[chuen Lin et al,] chuen Lin et al, J. M. Teaching computer programming in elementary schools: A pilot study.

[Cooper et al., 2003] Cooper, S., Dann, W., and Pausch, R. (2003). Teaching objects-first in introductory computer science. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 191–195, New York, NY, USA. ACM.

[Costa and Miranda, 2017] Costa, J. M. and Miranda, G. L. (2017). Relation between alice software and programming learning: A systematic review of the literature and meta-analysis. *British Journal of Educational Technology*, 48(6):1464–1474.

[Cypher and Smith, 1995] Cypher, A. and Smith, D. C. (1995). Kidsim: end user programming of simulations. In *Conference companion on Human factors in computing systems*, pages 35–36, Denver, Colorado, United States. ACM.

[Dann et al., 2008] Dann, W. P., Cooper, S., and Pausch, R. (2008). *Learning To Program with Alice*. Prentice Hall Press, Upper Saddle River, NJ, USA.

[Davis, 1989] Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):pp. 319–340.

[Davis et al., 1992] Davis, F. D., Bagozzi, R. P., and Warshaw, P. R. (1992). Extrinsic and intrinsic motivation to use computers in the workplace. *Journal of Applied Social Psychology*, 22(14):1111–1132.

[De Raadt et al., 2002] De Raadt, M., Watson, R., and Toleman, M. (2002). Language trends in introductory programming courses. In *Proceedings of the 2002 Informing Science+ Information Technology Education Joint Conference (InSITE 2002)*, pages 229–337. Informing Science Institute.

[Diwan et al., 2004] Diwan, A., Waite, W. M., Jackson, M. H., and Dickerson, J. (2004). Pl-detective: a system for teaching programming language concepts. *Journal on Educational Resources in Computing (JERIC)*, 4(4):1.

[Dougherty, 2007] Dougherty, J. P. (2007). Concept visualization in cs0 using alice. *Journal of Computing in Small Colleges*, 22(3):145–152.

[Feinberg and Murphy, 2000] Feinberg, S. and Murphy, M. (2000). Applying cognitive load theory to the design of web-based instruction. In *Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation: technology & teamwork*, IPCC/SIGDOC '00, pages 353–360, Piscataway, NJ, USA. IEEE Educational Activities Department.

[Felleisen et al.,] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. The teachscheme! project: Computing and programming for every student. Electronic version available from http://www.ccs.neu.edu/scheme/pubs/.

[Feurzeig et al., 1970] Feurzeig, W., Papert, S., Bloom, M., Grant, R., and Solomon, C. (1970). Programming-languages as a conceptual framework for teaching mathematics. *SIGCUE Outlook*, 4:13–17.

[Findler et al., 2002] Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. (2002). Drscheme: a programming environment for scheme. *J. Funct. Program.*, 12(2):159–182.

[Gal-Ezer et al., 2009] Gal-Ezer, J., Vilner, T., and Zur, E. (2009). Has the paradigm shift in cs1 a harmful effect on data structures courses: a case study. In *Proceedings of the 40th ACM technical symposium on Computer science education*, pages 126–130, Chattanooga, TN, USA. ACM.

[Garner, 2004] Garner, S. (2004). The use of a code restructuring tool in the learning of programming. In *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 277–277, New York, NY, USA. ACM.

[Goldwasser and Letscher, 2008] Goldwasser, M. H. and Letscher, D. (2008). Teaching an object-oriented cs1 with python. In *Proceedings of the*

13th annual conference on Innovation and technology in computer science education, pages 42–46, Madrid, Spain. ACM.

[Good and Howland, 2015] Good, J. and Howland, K. (2015). Natural language and programming: Designing effective environments for novices. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, pages 225–233.

[Grandell et al., 2006] Grandell, L., Peltomäki, M., Back, R.-J., and Salakoski, T. (2006). Why complicate things?: introducing programming in high school using python. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 71–80, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

[Haiman, 1985] Haiman, J. (1985). *Iconicity in syntax: proceedings of a Symposium on iconicity in syntax, Stanford, June 24-6, 1983*, volume 6. John Benjamins Publishing.

[Hayes, 1921] Hayes, M.H.S & Patterson, D. (1921). Experimental devolopment of the graphic rating scale. *Psychological Bulletin*, 18:98–99.

[Hoare, 1986] Hoare, P. (1986). Adults' attitudes to children with epilepsy: The use of a visual analogue scale questionnaire. *Journal of Psychosomatic Research*, 30(4):471 – 479.

[Holt, 1994] Holt, R. C. (1994). Introducing undergraduates to object orientation using the turing language. *Dept. of Computer Science, University of Toronto*, 25:324–328.

[Holt and Cordy, 1988] Holt, R. C. and Cordy, J. R. (1988). The turing programming language. *Commun. ACM*, 31(12):1410–1423.

[Holt et al., 1987] Holt, R. C., Matthews, P. A., Rosselet, J. A., and Cordy, J. R. (1987). *The Turing programming language: design and definition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Howe, 1982] Howe, J. e. a. (1982). *Teaching Mathematics Through Programming in the Classroom*. DAI research paper. Department of Artificial Intelligence, University of Edinburgh.

[Howland and Good, 2015] Howland, K. and Good, J. (2015). Learning to communicate computationally with flip: A bi-modal programming language for game creation. *Computers & Education*, 80:224 – 240.

[Ivanović and Pitnerl, 2011] Ivanović, M. and Pitnerl, T. (2011). Technology-enhanced learning for java programming: Duo cum faciunt idem, non est idem. *ACM Inroads*, 2(1):55–63.

[Janzen et al., 2013] Janzen, D. S., Clements, J., and Hilton, M. (2013). An evaluation of interactive test-driven labs with webide in cs0. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1090–1098. IEEE Press.

[Jonas, 2013] Jonas, M. (2013). Teaching introductory progamming using multiplayer board game strategies in greenfoot. *Journal of Computing Sciences in Colleges*, 28(6):19–25.

[Kelleher and Pausch, 2005] Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137.

[Kelleher et al., 2007] Kelleher, C., Pausch, R., and Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1455–1464, San Jose, California, USA. ACM.

[Kölling, 2010] Kölling, M. (2010). The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):14.

[Kölling et al., 2015] Kölling, M., Brown, N. C. C., and Altadmri, A. (2015). Frame-based editing: Easing the transition from blocks to text-based

programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15, pages 29–38, New York, NY, USA. ACM.

[Kruglyk and Lvov, 2012] Kruglyk, V. and Lvov, M. (2012). Choosing the first educational programming language. In *ICT in Education, Research, and Industrial Applications: 8th International Conference, ICTERI 2012*.

[Kunkle and Allen, 2016] Kunkle, W. M. and Allen, R. B. (2016). The impact of different teaching approaches and languages on student learning of introductory programming concepts. *Trans. Comput. Educ.*, 16(1):3:1–3:26.

[Laboratories and Kernighan, 1981] Laboratories, A. . T. B. and Kernighan, B. (1981). *Why Pascal is Not My Favorite Programming Language*. Bell Computing Science-TR–100. Bell Laboratories.

[Layman and Hall, 1988] Layman, J. and Hall, W. (1988). Logo: a cause for concern. *Comput. Educ.*, 12(1):107–112.

[Lister et al., 2006a] Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C., and Whalley, J. L. (2006a). Research perspectives on the objects-early debate. In *Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 146–165, Bologna, Italy. ACM.

[Lister et al., 2006b] Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006b). Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *SIGCSE Bull.*, 38(3):118–122.

[Lorenzen and Sattar, 2008] Lorenzen, T. and Sattar, A. (2008). Objects first using alice to introduce object constructs in cs1. *SIGCSE Bull.*, 40(2):62–64.

[Louca, 2004] Louca, L. (2004). Programming environments for young learners: a comparison of their characteristics and students' use. In *Proceedings of the*

2004 conference on Interaction design and children: building a community, pages 129–130, Maryland. ACM.

[Louca, 2005] Louca, L. (2005). The syntax or the story behind it?: a usability study of student work with computer-based programming environments in elementary science. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 849–858, Portland, Oregon, USA. ACM.

[Malan and Leitner, 2007] Malan, D. J. and Leitner, H. H. (2007). Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 223–227, Covington, Kentucky, USA. ACM.

[Maloney et al.,] Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. The scratch programming language and environment.

[Maloney et al., 2008] Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., and Rusk, N. (2008). Programming by choice: urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 367–371, Portland, OR, USA. ACM.

[Martin, 1996] Martin, J. L. (1996). Is turing a better language for teaching programming than pascal? Honours Dissertation.

[McIver and Conway, 1996] McIver, L. and Conway, D. (1996). Seven deadly sins of introductory programming language design. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, SEEP '96, pages 309–, Washington, DC, USA. IEEE Computer Society.

[McIver and Conway, 1999] McIver, L. and Conway, D. (1999). Grail: A zeroth programming language. In *International Conference on Computers in Education*.

[Mellon,] Mellon, C. Alice 2.0 programming language.

[Microsoft,] Microsoft. Vs.net. Product website: https://www.visualstudio.com/ Last accessed 29/6/2016, 3.00pm.

[Miller and Ranum, 2005] Miller, B. N. and Ranum, D. L. (2005). Teaching an introductory computer science sequence with python.

[Mitchel Resnick and Eric Rosenbaum, 2009] Mitchel Resnick, J. M. and Eric Rosenbaum, J. S. (2009). Scratch: Programming for everyone. *ACM Communications*.

[Moons and De Backer, 2013] Moons, J. and De Backer, C. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Comput. Educ.*, 60(1):368–384.

[Mullins et al., 2009] Mullins, P., Whitfield, D., and Conlon, M. (2009). Using alice 2.0 as a first language. *J. Comput. Small Coll.*, 24(3):136–143.

[Pane, 1998] Pane, J. F. (1998). Designing a programming system for children with a focus on usability. In *CHI 98 conference summary on Human factors in computing systems*, pages 62–63. ACM Press.

[Pane and Myers, 2002] Pane, J. F. and Myers, B. A. (2002). The impact of human-centered features on the usability of a programming system for children. In *CHI '02 extended abstracts on Human factors in computing systems*, pages 684–685, Minneapolis, Minnesota, USA. ACM.

[Papert, 1980] Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. The Harvester Press Limited.

[Parr, 1993] Parr, T. J. (1993). *Obtaining Practical Variants Of LL(k) And LR(k) For k>1 By Splitting The Atomic k-Tuple*. PhD thesis, School of Electrical Engineering, Purdue University.

[Parr, 1997] Parr, T. J. (1997). *Language Translation Using PCCTS and C++*. Automata Publishing Company.

[Parr and Quong, 1995] Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll(k) parser generator. *Software–Practice and Experience*, 25(7):789–810.

[Parsons, 1992] Parsons, T. W. (1992). *Introduction to compiler construction*. Computer Science Press, Inc.

[Powers et al., 2007] Powers, K., Ecott, S., and Hirshfield, L. M. (2007). Through the looking glass: teaching cs0 with alice. *SIGCSE Bull.*, 39(1):213–217.

[Rader et al., 1997] Rader, C., Brand, C., and Lewis, C. (1997). Degrees of comprehension: children's understanding of a visual programming environment. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 351–358, New York, NY, USA. ACM.

[Raina Mason et al., 2015] Raina Mason, S. C. U., Graham Cooper, S. C. U., Barry Wilks, S. C. U., and Authors (2015). Using cognitive load theory to select an environment for teaching mobile apps development. School of Business and Tourism, page 47. Australain Computer Society.

[Reges, 2006] Reges, S. (2006). Back to basics in cs1 and cs2. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 293–297, Houston, Texas, USA. ACM.

[Robins et al., 2003] Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137.

[Rodger et al., 2009] Rodger, S. H., Hayes, J., Lezin, G., Qin, H., Nelson, D., Tucker, R., Lopez, M., Cooper, S., Dann, W., and Slater, D. (2009). Engaging middle school teachers and students with alice in a diverse set of subjects. In *Proceedings of the 40th ACM technical symposium on Computer science education*, pages 271–275, Chattanooga, TN, USA. ACM.

[Rubin, 2013] Rubin, M. J. (2013). The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 651–656. ACM.

[Sáez-Lòpez et al., 2016] Sáez-Lòpez, J.-M., Romàn-Gonzàlez, M., and Vàzquez-Cano, E. (2016). Visual programming languages integrated across the curriculum in elementary school: A two year case study using scratch in five schools. *Computers & Education*, 97:129 – 141.

[Sapounidis et al., 2015] Sapounidis, T., Demetriadis, S., and Stamelos, I. (2015). Evaluating children performance with graphical and tangible robot programming tools. *Personal and Ubiquitous Computing*, 19(1):225–237.

[Scripps and Sanner, 1999] Scripps, S. T. and Sanner, M. F. (1999). Python: A programming language for software integration and development. *J. Mol. Graphics Mod*, 17:57–61.

[Sim et al., 2006] Sim, G., MacFarlane, S., and Read, J. (2006). All work and no play: Measuring fun, usability, and learning in software for children. *Computers & Education*, 46(3):235 – 248.

[Smith et al., 1996] Smith, D. C., Cypher, A., and Schmucker, K. (1996). Making programming easier for children. *Interactions*, 3(5):58–67.

[Smith et al., 1994] Smith, D. C., Cypher, A., and Spohrer, J. (1994). Kidsim: programming agents without a programming language. *Commun. ACM*, 37(7):54–67.

[Smith et al., 2000] Smith, D. C., Cypher, A., and Tesler, L. (2000). Programming by example: novice programming comes of age. *Commun. ACM*, 43(3):75–81.

[Squires and Preece, 1996] Squires, D. and Preece, J. (1996). Usability and learning: Evaluating the potential of educational software. *Computers & Education*, 27(1):15 – 22.

[Stolee and Fristoe, 2011] Stolee, K. T. and Fristoe, T. (2011). Expressing computer science concepts through kodu game lab. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 99–104. ACM.

[Tedford, 2008] Tedford, P. (2008). Teach with alice - and have fun!: pre-conference workshop. *J. Comput. Small Coll.*, 23(6):6–6.

[Teitelbaum and Reps, 1981] Teitelbaum, T. and Reps, T. (1981). The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573.

[Trigwell and Prosser, 1991] Trigwell, K. and Prosser, M. (1991). Improving the quality of student learning: the influence of learning context and student approaches to learning on learning outcomes. *Higher Education*, 22:251–266. 10.1007/BF00132290.

[Uysal, 2016] Uysal, M. P. (2016). Evaluation of learning environments for object-oriented programming: measuring cognitive load with a novel measurement technique. *Interactive Learning Environments*, 24(7):1590–1609.

[Weintrop and Wilensky,] Weintrop, D. and Wilensky, U. Bringing blocks-based programming into high school computer science classrooms.

[Winslow, 1996] Winslow, L. E. (1996). Programming pedagogy - a psychological overview. *SIGCSE Bull.*, 28(3):17–22.

Appendix A

Testing Activities

A.1 Writing Task One

SCOOT

For this task, you will be writing code for two classes, a Car and an Employee. Both of these classes will contain several attributes.

A Car will contain information about the model, production year, colour and number of doors. You can choose any variable names you like, but note that variable names cannot contain spaces

An Employee will contain a name, an employee ID, a department number and a pay rate. These attributes can also be named as you choose, but should be logical, and contain no spaces.

Once you've created these two classes, give the attributes some values and display all class information to the screen. Note that SCOOT does not support spaces inside strings, so all values must be a single word or number.

C++

For this task, you will be writing code for two classes, a Car and an Employee. Both of these classes will contain several attributes.

A Car will contain information about the model, production year, colour and number of doors. You can choose any variable names you like, but remember that variable names cannot contain spaces.

An Employee will contain a name, an employee ID, a department number and a pay rate. These attributes can also be named as you choose, but should be logical, and contain no spaces.

Write a main program that creates an instance of each of these classes, and prints their contents to the screen. The class constructors should set default values for each of these classes. Add accessor functions to both objects to retrieve the data, and print it to the screen from your main program, not inside the class.

A.2 Writing Task Two

SCOOT

Now we will add some useful behaviours to our Car and Employee classes.

First of all, we will add a method called Repaint to the Car object. Repaint will be a simple method that checks the car's colour. If the colour is Red, then the car will be repainted White.

Next we will add a method called PayRaise to the Employee object. The PayRaise method will increase the PayRate by 500. Once this has been done, see if you can add a method called MaxRaise which repeatedly calls the PayRaise method while the PayRate is less than 15000.

Once you have added these methods to the Car and Employee objects, test them all.

C++

Now we will add some useful behaviours to our Car and Employee classes.

First of all, we will add a method called Repaint to the Car object. Repaint will be a simple method that checks the car's colour. If the colour is Red, then the car will be repainted White.

Next we will add a method called PayRaise to the Employee object. The PayRaise method will increase the PayRate by 500. Once this has been done, see if you can add a method called MaxRaise which repeatedly calls the PayRaise method while the PayRate is less than 15000.

Once you have added these methods to the Car and Employee objects, add code to your main program which tests them all.

A.3 Reading Task

The reading task was presented to participants as shown below, along with the questions that follow. Lines were numbered in the SCOOT code, and so were lines and pages in the C++ code. The page and line numbers in the questions were different between SCOOT and C++ users but the questions were otherwise identical.

SCOOT

```
A Person is an Object;
A Person has a Name;
A Person has an Age;

An Employee is a Person;
An Employee has an ID;
An Employee has a Salary;

Set Name of Person to Tom;
Set Age of Person to 58;

Set Name of Employee to Bob;
Set Age of Employee to 42;
Set ID of Employee to 112312;
Set Salary of Employee to 23440;

Get Person;
Get Employee;

Set Name of Person to Peter;
Set Name of Employee to John;
```

```
Set ID of Employee to 3212212;
Set Salary of Employee to 28230;

Get Person;
Get Employee;

An Employee performs PayRise;
Add 1000 to Salary of Employee;
End Recipe;

Evaluate PayRise of Employee;
Get Employee;
```

C++

main.cpp

```
#include <iostream>
#include "person.h"
#include "employee.h"
using namespace std;
int main()
{
person myPerson("Tom", 58);
person otherPerson;
employee emp1("Bob", 58, "112312", 23440);
employee emp2;
otherPerson.setName("John");
emp2.setName("Peter");
emp2.setID("3212212");
emp2.setSalary(28230);

cout << "First Person:" << endl;
myPerson.print();
cout << endl << "Second Person:" << endl;
otherPerson.print();

cout << endl << "First Employee:" << endl;
emp1.print();
cout << endl << "Second Employee:" << endl;
emp2.print(Code Reading Questions);

return 0;
}
```

person.h

```
#include <string>
using std::string;
#ifndef __PERSON__
#define __PERSON__
class person
{
public:
person(string n = "", int a = 18);
void setName(string name);
void setAge(int age);
string getName();
int getAge();
void print();
private:
string name;
int age
;};
#endif
```

person.cpp

```
#include <iostream>
#include <string>
#include "person.h"

using std::cout;
using std::endl;
using std::string;

person::person(string n, int a)
{
name = n;
if (a > 0)
age = a;
else
age = 0;
}

void person::setName(string n)
{
name = n;
}

void person::setAge(int a)
{
if (a > 0)
{
age = a;
}
```

```
}

string person::getName()
{
return name;
}

int person::getAge()
{
return age;
}

void person::print()
{
cout << "Name:\t" << name << endl;
cout << "Age:\t" << age << endl;
}
```

employee.h

```
#include "person.h"
#include <string>
using std::string;

class employee : public person
{
public:
employee(string n = "", int a = 0, string id = "", double s = 0);
void setID(string id);
void setSalary(double s);
string getID();
double getSalary();
void payRise();
void print();
private:
string empID;
double salary;
};
```

employee.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "employee.h"

using std::cout;
using std::endl;
using std::string;
using std::fixed;
using std::setprecision;

employee::employee(string n, int a, string id, double s)
        : person(n, a)
{
empID = id;
if (s > 0)
salary = s;
}

void employee::setID(string id)
{
empID = id;
}

void employee::setSalary(double s)
{
if (s > 0)
salary = s;
```

```
}

string employee::getID()
{
return empID;
}

double employee::getSalary()
{
return salary;
}

void employee::payRise()
{
salary = salary + 1000
}

void employee::print()
{
person::print();
cout << "ID:\t" << empID << endl;
cout << fixed << setprecision(2);
cout << "Salary:\t$" << salary << endl;
}
```

Reading Task Questions

SCOOT Code Reading Questions

1. Please give the names of all the attributes of the Employee object.

_ _

_ _

2. Please give the names of all behaviours of the Employee object.

_ _

_ _

_ _

3. Please give the names and values of all attributes of the Employee class when "Get Employee;" is first used (line 10)

_ _

_ _

4. Please give the names and values of all attributes of the Employee class when "Get Employee;" is next used (line 18)

_ _

_ _

5. Please give the names and values of all attributes of the Employee class when "Get Employee;" is last used (line 20)

_ _

_ _

C++ Code Reading Questions

1. Please give the names of all the attributes of the Employee object.

_ _

_ _

2. Please give the names of all behaviours of the Employee object.

_ _

_ _

_ _

3. Please give the names and values of all attributes of the Employee class when the 'print' method is first used (line 10, page 3)

_ _

_ _

4. Please give the names and values of all attributes of the Employee class when the 'print' method is next used (line 13, page 3)

_ _

_ _

5. Please give the names and values of all attributes of the Employee class when the 'print' method is last used (line 16, page 3)

_ _

_ _

A.4 Debugging Task

Participants were given copies of the code shown below, along with the task description:

> Identify programming errors in this code. Locate errors using the page and line number, and describe the problem. Suggest a replacement line that would correct the problem.

Both SCOOT and C++ code contained 5 errors, and lines were numbered in the SCOOT code, and so were lines and pages in the C++ code, just like the Reading Task.

SCOOT

```
A Car is an Object;
A Car has a Model;
A Car has a ProductionYear;
A Car has a Colour;
A Car has a Price;

Set Model of Car to Camry;
Set ProductionYear of Car to 2001;
Set Colour of Car to Orange;
Set price of Car to 14500;

A Car performs addFees;
Set Price of Car to Price of Car + 1000
End Recipe;

A Car can repaint;
Compare Colour of Car to Red;
If equal;
Set Colour of Car to Black;
End If;
End Recipe;

Car.addFees();
Evaluate repaint of Car;
Get Car;
```

C++

main.cpp

```
#include <iostream>
#include "person.h"
#include "employee.h"

using namespace std;

int main()
{
        person myPerson("Tom", 58);
        person otherPerson;
        employee emp1("Bob", 58, "112312", 23440);
        employee emp2;
        otherPerson.setName("John");
        emp2.setName(Peter);
        emp2.setID("3212212");
        emp2.setSalary(28230);
        cout << "First Person:" << endl;
        myPerson.print();
        cout << endl << "Second Person:" << endl;
        otherPerson.print();
        cout << endl << "First Employee:" << endl;
        emp1.print;
        cout << endl << "Second Employee:" << endl;
        emp2.print();

        return 0;
}
```

person.h

```
#include <string>

using std::string;

#ifndef __PERSON__
#define __PERSON__

class person
{
        public:
                person(string n = "", int a = 18);
                void setName(string name);
                void setAge(int age);
                string getName();
                int getAge();
                void print();
        private:
                string name;
                int age;
}

#endif
```

person.cpp

```
#include <iostream>
#include <string>
#include "person.h"

using std::cout;
using std::endl;
using std::string;

person::person(string n, int a)
{
        name = n;
        if (a > 0)
                age = a;
        else
                age = 0;
}

void setName(string n)
{
        name = n;
}

void person::setAge(int a)
{
        if (a > 0)
        {
                age = a;
        }
```

```
}

string person::getName()
{
        return name;
}

int person::getAge()
{
        return age;
}

void person::print()
{
        cout << "Name:\t" << name << endl;
        cout << "Age:\t" << age << endl;
}
```

employee.h

```
#include "person.h"
#include <string>

using std::string;

class employee : person
{
        public:
                employee(string id = "", double s = 0);
                void setID(string id);
                void setSalary(double s);
                string getID();
                double getSalary();
                void print();
        private:
                string empID;
                double salary;
};
```

employee.cpp

```
#include <iostream>
#include <string>
#include <iomanip>
#include "employee.cpp"

using std::cout;
using std::endl;
using std::string;
using std::fixed;
using std::setprecision;

employee::employee(string id, double s)
        : person(n, a)
{
        empID = id;
        if (s > 0)
                salary = s;
}

void employee::setID(string id)
{
        empID = id;
}

void employee::setSalary(double s)
{
        if (s > 0)
                salary = s;
}
```

```
string employee::getID()
{
        return empID;
}

double employee::getSalary()
{
        return salary;
}

void employee::print()
{
        person::print();
        cout << "ID:\t" empID << endl;
        cout << fixed << setprecision(2);
        cout << "Salary:\t$" << salary << endl;
}
```

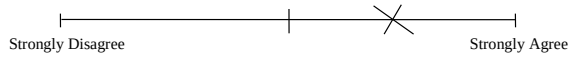
A.5 Questionnaires

C++ Language Questionnaire

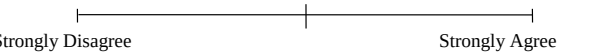
Today's Date: _____

Your Programmer number: ___

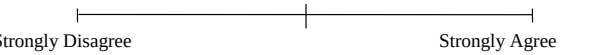
In the following statements, indicate by marking on the scale how much you agree e.g:



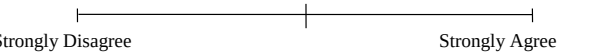
Statement 1. When programming, using C++ increases my programming productivity



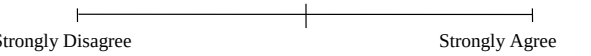
Statement 2. Using C++ makes it easier to accomplish programming tasks more quickly



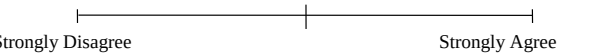
Statement 3. Overall, I find C++ useful when programming



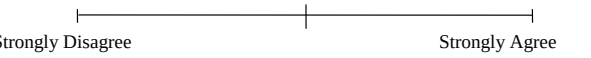
Statement 4. I often become confused when using C++



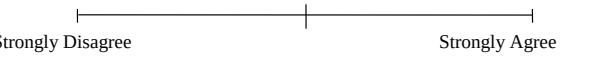
Statement 5. I find it easy to get C++ to do what I want it to do



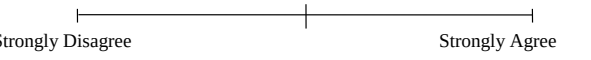
Statement 6. I found completing the tasks took longer than I expected



Statement 7. It is easy for me to remember how to perform tasks using C++



Statement 8. Overall, I find C++ easy to use

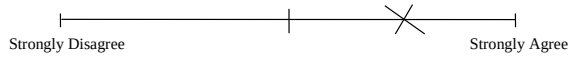


SCOOT Language Questionnaire

Today's Date: _____

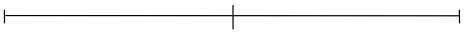
Your Programmer number: ___

In the following statements, indicate by marking on the scale how much you agree e.g:



Strongly Disagree Strongly Agree

Statement 1. When programming, using SCOOT increases my programming productivity

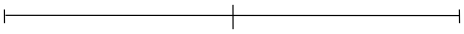


Strongly Disagree Strongly Agree

Statement 2. Using SCOOT makes it easier to accomplish programming tasks more quickly

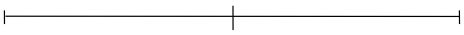
Strongly Disagree Strongly Agree

Statement 3. Overall, I find SCOOT useful when programming



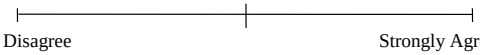
Strongly Disagree Strongly Agree

Statement 4. I often become confused when using SCOOT



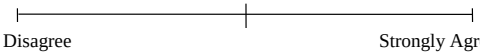
Strongly Disagree Strongly Agree

Statement 5. I find it easy to get SCOOT to do what I want it to do



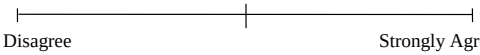
Strongly Disagree Strongly Agree

Statement 6. I found completing the tasks took longer than I expected



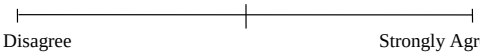
Strongly Disagree Strongly Agree

Statement 7. It is easy for me to remember how to perform tasks using SCOOT



Strongly Disagree Strongly Agree

Statement 8. Overall, I find SCOOT easy to use



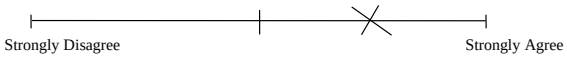
Strongly Disagree Strongly Agree

Preliminary Questionnaire

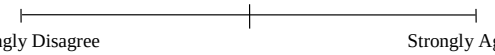
Today's Date: _____

Your Programmer number: ___

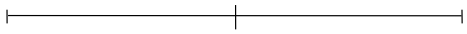
In the following statements, indicate by marking on the scale how much you agree
e.g:



Statement 1. I expect that I will enjoy completing the programming tasks



Statement 2. I expect that I will be able to complete the programming tasks quickly



Statement 3. I expect that completing the programming tasks will be interesting