PICTURES TO OBJECTS:
TRANSITIONING TO OBJECT ORIENTED PROGRAMMING

Sarah Heimlich

Bachelor of Engineering
Software Engineering

Department of Engineering
Macquarie University

June 10, 2016

Supervisor: Prof. Michael Johnson

ACKNOWLEDGMENTS

I would like to acknowledge a number of people who helped me throughout the course of my undergraduate degree and this thesis.

First of all my parents who gave me my love of learning and showed me the sky is the limit. Thank you for breaking down stereotypes and giving me the courage to pursue what I'm passionate about. Also thank you to my sister Jaye for putting up with all my crazy questions and late night working.

Thank you to Prof Michael Johnson for giving huge amounts of flexibility in this thesis—from the topic to the timing and location. I truly appreciate how much time and attention you have given this project.

STATEMENT OF CANDIDATE

I, Sarah Heimlich, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any academic institution.

Student's Name: Sarah Heimlich

Student's Signature: Sarah Heimlich (electronic)

Date: June 10, 2016

ABSTRACT

Teaching programming is considered hard, but transitioning from procedural to object oriented programming is even harder. Yet, this is what the new National Digital Technologies curriculum expects of every Australian student. An examination of the literature reveals that transitioning to object oriented programming is difficult, but for unknown reasons. Here, we examine and study this transition through the literature and a survey. The survey results help us identify programming concepts which are easier, and other concepts which are more difficult, to learn. The data gathered suggests that fourteen years old is the optimal age for learning object oriented programming. Based on the information gathered through the literature review and survey, a new visual programming language for the LEGO MINDSTORMS EV3 that can assist in the transition to object oriented programming is created. We propose a new measure to determine how hard an object oriented programming language is to learn and discover five potential threshold object oriented concepts. It is hoped this project will provide insight into the transition to object orientation and ease the difficult transition to object oriented programming.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The world requires engineers to provide the necessary and expected utilities of a modern nation [5]. However, there is a significant lack of trained personnel in engineering and Information and Communication Technology (ICT) related careers. Many computer science employees are in industries other than ICT and an estimated 100,000 new ICT positions will be available by 2020 [33]. To fill all job vacancies, Australian companies are recruiting from outside the country. In 2015, four of the top fifteen primary occupations of skilled worker (subclass 457) visa recipients were in ICT related careers with ICT professionals accounting for 12% of subclass 457 visas [22] [33].

With this setting as the backdrop, it is easy to understand why computer science education has become an increasing priority in Australia and around the world. In particular, this may be seen through the introduction of mandated national computer science curricula in countries including Australia [2] and the United Kingdom [18]. As students progress from primary school to secondary and tertiary education, these requirements take pupils from visual procedural to textual object oriented programming languages.

It is widely recognized that students struggle with transitioning from procedural to object oriented programming. There is research to support the claim that learning object oriented programming first enables students to easily transition "back" to procedural structures [31]. Despite this, with an increasing priority on computer science education, national curricula often mandate students begin programming in early primary school with a procedural, visual language. This is before students can fully comprehend abstraction [53] which is a critical concept for computing and object oriented programming [43].

In addition to curriculum requirements, most robotics competitions for primary and lower-secondary students have a requirement to use a visual procedural language. One reason for this is most of these competitions utilize the LEGO robotics platform called the "EV3 MINDSTORMS". The programming language LEGO provides for the EV3 MINDSTORMS is a visual procedural language.

As a result of the requirements from national curricula and robotics competitions students are often learning to program with a visual procedural language in early primary school. This creates a disconnect between what the research suggests and what is practiced. Ultimately, this makes learning object oriented programming more difficult for

students.

With this disconnect, one would expect a large amount of academic research into how to bridge the gap between procedural and object oriented programming. Yet, there is very little information available. The literature abruptly stops with the problem, offering no solutions.

1.1 Project Goal

The goal of this project is to create a programming language for the LEGO EV3 MINDSTORMS to assist the transition from visual procedural to object oriented programming languages.

It should be noted that the goal of this project is not to create an object oriented language. While many object oriented concepts will be included in the language, it does not need to implement all features of an object oriented language. It is critical this point is understood as many decisions throughout the project will be based on this point.

To achieve this goal, the project is divided into two main components: understanding the transition to object oriented programming and creating the language.

The literature available on the transition to object oriented programming was insufficient for understanding the problem. As a result, the first component of this project was to find information regarding the subject area. To accomplish this, in addition to a full literature review a survey was conducted to better understand which object oriented concepts were perceived by learners to be easy or difficult to learn, why object oriented concepts are easy or difficult to comprehend, and if there is an ideal age to learn object oriented programming.

With this background, the language was then defined, designed, implemented, and validated.

1.2 Project Planning

As with any project, it is critical to begin by setting a plan. Here we consider the project's scope, time frame, and cost.

1.2.1 Scope

The project's initial scope is set through the goal stated in Section 1.1. From this goal, it is clear that there are two main stages to the project—requirements elicitation and system implementation. Many projects would continue into acceptance and maintenance stages. Due to the time frame of the project, this is infeasible. As a result, the scope of the project has been limited to the requirements and implementation. The project is validated by ensuring the final product meets the requirements.

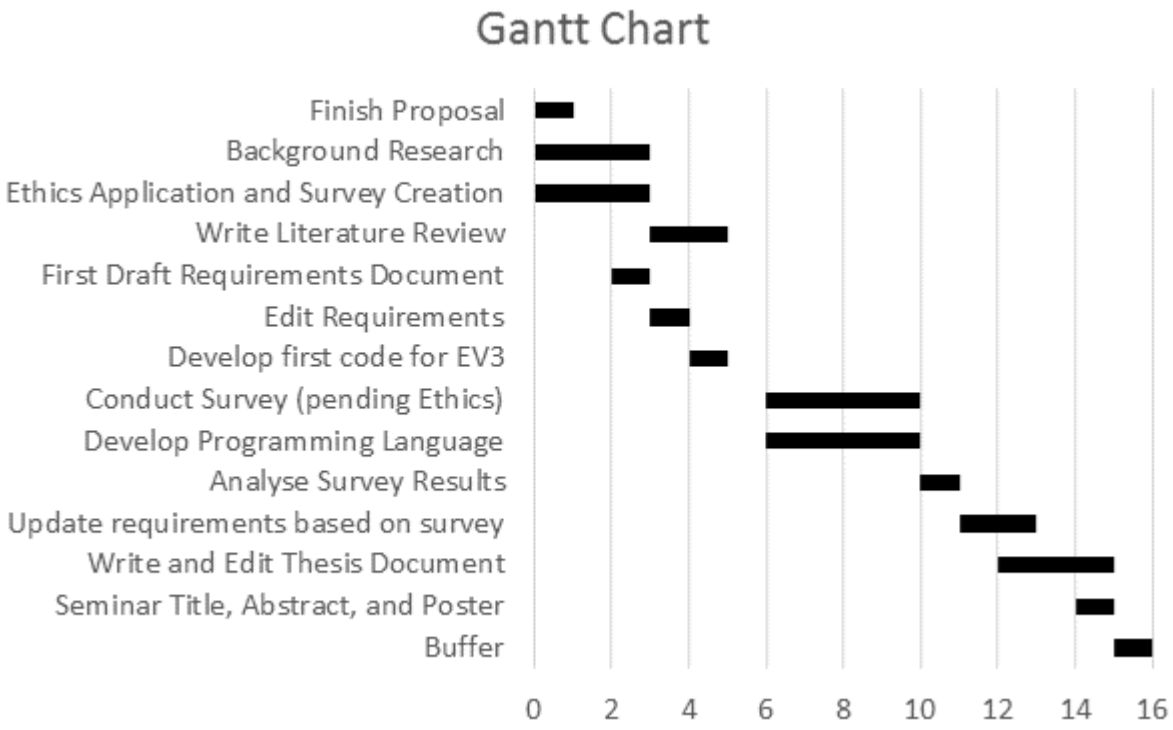

Figure 1.1: Gantt chart representing the project's schedule.

1.2.2 Time

To ensure the project was completed on time, a schedule was developed at the beginning. This schedule was approved by the project supervisor. The original project plan can be seen in the schedule in Table 1.1 and Gantt chart in Figure 1.1.

As with any project, there was some schedule slippage and unforeseen difficulties. In particular, ethics approval was not received until May, about a month after the survey was supposed to be started. Despite these set backs, the project was still completed on time.

1.2.3 Cost

The project was allocated budget of $300. Due to the nature of the project, and the equipment already owned by Macquarie University, it was originally estimated the project would cost $0 leaving a $300 surplus. This has occurred with the project having no financial cost.

Table 1.1: Schedule for the project.

Week	Requirements Elicitation	Thesis Document	Programming Language	Deliverables
26/02	Finish Proposal	Start Collating Papers		
4/3	Draft Ethics Application and Survey	Read Papers		
11/3	Edit and Submit Ethics Application	Read Papers	First draft Requirements Doc	Project Specifications and Plan Agreement to Mike
18/03		Start writing lit review section	Edit Requirements Doc	
25/03		Finish "Prior research" section	Develop code for the EV3	
1/4	Send out Survey (Pending Ethics Approval)		Start developing language	Progress Report
8/4			Continue developing language	
15/04			Continue developing language	
22/04			Continue developing language	
29/04	Analyze Survey Results		First draft of language	
6/5		Determine thesis "Story"	Update requirements based on Survey	
13/05		Begin writing	Update Language based on feedback and survey	
20/05		First Draft Complete	Finalize Language based on feedback and survey	
27/05		Edit Thesis Create Poster/Presentation		
3/6	Week of buffer	Practice Presentation	Week of buffer	Final Report
10/6		Practice Presentation		
17/06				Seminar Title, Abstract, and Poster

Chapter 2

Background and Related Work

Before we can begin developing *the system*, it is critical to understand the subject area of the project. This extends to computer science education, the LEGO MINDSTORMS EV3, and the psychology of abstraction. Additionally, it is important to have a thorough understanding of computing concepts including object oriented programming and creating a visual programming language. Here, we present a brief overview of these topics based on publications from a variety of journals, authors, and subjects. This background knowledge is used in later sections to justify decisions and to provide a basis for further arguments.

2.1 Computer Science Education

2.1.1 Curricula Requirements

As computer science becomes an increasing priority around the world, digital technologies education has received more academic attention. In particular, researchers have discovered that it takes ten years of experience to progress from a novice to an expert programmer [52]. Despite this, most undergraduate degrees vary in length from three to four years. As a result, to graduate expert programmers, we must introduce programming in years 5-6. This is reflected in the Australian and UK national computing curricula as outlined below.

The transition from visual to object oriented programming languages is mandated in curricula in some countries. In the Australian Digital Technologies Curriculum, students are required to begin learning a graphic programming language in year 3, a general purpose text language in year 7, and an object oriented language in year 9 [2]. From years 1-2, students begin understanding algorithms, but they are not required to learn a programming language.

This sequence of learning differs from the United Kingdom where the computing curriculum begins in year 1, two years before the Australian curriculum. Students start with a visual programming language and transition to a textual language in year 7 at the same time as their Australian peers [4, 18]. Despite starting to program sooner, there is no requirement for students in the UK to learn an object oriented language.

The transition between programming languages occurs outside of the curriculum as well. The FIRST Robotics family of competitions for students in years k to 12 also has students transitioning from visual to object oriented programming. In general, FIRST students use visual programming languages in years 1 to 6 and object oriented programming languages in years 7 to 12.

A graph of these different programs and a curriculum mapping to programming paradigms is shown in Figure 2.1. By examining this figure, we note the steep learning curve expected of students from general purpose text languages to object oriented languages.

It should also be observed that we can expect students to transition to object oriented programming in years 7 to 9. This is important to note as it will help define *the system's* target demographic in a later section.

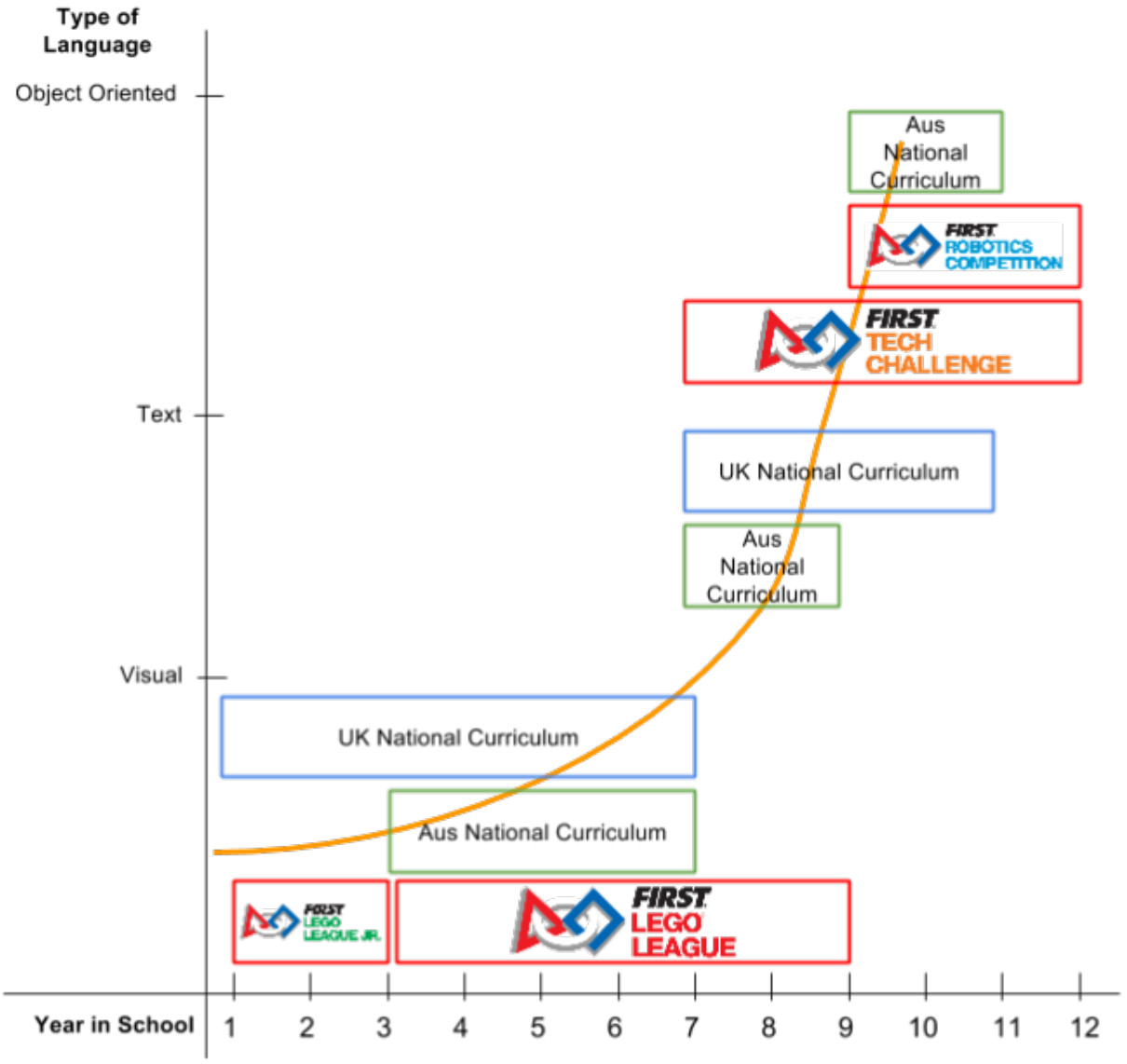

Figure 2.1: Mapping different robotics programs' and countries' programming language requirements to years in school [1, 2, 4, 6, 7, 18, 21]. The orange line depicts the average learning curve expected of students.

2.1.2 Pedagogy

In the words of a University of Helsinki research group, "Teaching programming is hard" [57]. As of 2011, The University of Helsink's computer science long term drop out rate was approximately 45%. As a result, it is easy to understand why so many different languages and approaches have been tried in teaching programming. Debates in teaching programming include depth versus breadth, procedural versus object oriented, and

many more [24]. Here, we provide an overview of some proposed methods for teaching programming.

Apprenticeship

One approach advocates for an apprenticeship model to be used in teaching programming. In contrast to a traditional apprenticeship which is based on manual skills, computer science falls under the cognitive apprenticeship category. Cognitive apprenticeships can be divided into three stages—modeling, scaffolding, and fading [57].

In the modeling stage, the teacher provides a description of how a problem would be solved [57]. This could occur through worked examples in a lecture or teaching programming theory.

From modeling, students transition to scaffolding. In this stage, the teacher provides structured exercises [57]. This could occur through assignments or other types of exercises.

The final stage is fading. Here, the teacher provides less contrived solutions and allows the apprentice to solve full problems [57]. This slowly moves the apprentice towards self-sufficiency. At this point, the apprentice should be ready to tackle challenges on their own [24].

Low Floor, High Ceiling, Wide Walls

The principle of creating a teaching environment with a low floor, high ceiling, and wide walls, is prevalent in the literature [34, 36, 51, 60]. This does not refer to the location in which students are taught, but rather to the limitations placed upon students. These limitations may be placed by the language or by the concepts taught.

The low floor and high ceiling are discussed more extensively . The low floor indicates an easy entry point for beginners [34]. For example, in COMP115 at Macquarie University, students are taught Processing instead of Java. This is done because Processing has fewer requirements to begin writing a program. This provides a lower entry point for students who are taking their first computing class. The high ceiling speaks to the language's ability to support complex projects [36].

The Lifelong Kindergarten group at MIT also coined the term "wide walls" which reflects the breadth of projects that can be completed using the language [51, 60].

Student Engagement

It is also critical to find ways to fully engage students in programming classes. This is noted in numerous papers supporting different pedagogies [24, 30, 45, 51, 57]. Interactive languages that enable visual feedback are highly recommended as the student can see the direct impact of their code [30, 45]. By seeing the impact, students are better able to understand what their program is doing. This in turn enables a deeper understanding of the impact of each statement. By using a robotics platform, the programming language will enable this direct feedback.

2.1.3 Languages to teach programming

As with all debates over programming languages, there are a number of differing opinions about what style of programming language is best to teach programming. Text, visual, programming by example, and tiered languages have all been suggested as potential ways to teach programming [47, 49]. To better understand programming languages designed for teaching, we explore the Scratch programming language in depth below.

Scratch

Scratch is one of the most popular languages for introducing students to computing with nearly 14 million projects shared on the Scratch website [13]. Developed at MIT's Media Lab by Professor Mitchel Resnik, Scratch is a visual language that allows users to program characters, called **Sprites**, in a scene, called the **Stage**. While multiple **Sprites** normally occur in a Scratch project, there is only **Stage** Scratch was built on the theory of having a low floor, high ceiling, and wide walls as discussed in Section 2.1.2. There are a number of Scratch's characteristics that make it easy for users to learn.

Firstly, the language components act like puzzle pieces, only the correct parts can fit together [45]. For example, an integer cannot fit where a function or boolean is required. This allows the user to intuitively see where each part can, and cannot, be placed. Following on from this, each program created is automatically syntactically correct [45].

Scratch is an event driven language. Each section of the code begins when a different trigger occurs. Triggers range from the keyboard, to **Sprite** interaction, and many more options. Scratch provides three first-class data types—boolean, number, and string—and automatic conversion between numbers and text as required [45].

Sprites are the major component of any scratch program. Users create "costumes" and "scripts" for each **Sprite** and the **Stage**. **Sprites** and the **Stage** can contain variables and behaviors making the language object-based [45]. Despite this, Scratch cannot be considered Object Oriented as it does not support encapsulation, inheritance, or polymorphism. A **Sprite** can communicate with other **Sprites** using the broadcast system. The broadcast system is one-to-many [45], a **Sprite** will broadcast a message which can trigger behaviors in other **Sprites**.

2.1.4 Programming Paradigms

There is much debate over what paradigm to first teach students. Some argue procedural languages are the best place to start as they offer the simplest option [26]. However, this view appears to be in the minority with most researchers agreeing it is easier to learn object oriented programming first and then later transition to this paradigm [31, 42, 52].

What the literature does not address is what makes this transition difficult for students. This gap in the literature is very significant. It is impossible to create new methodologies to help students transition to object oriented programming without first understanding why they are struggling.

This knowledge gap implies *the system* cannot be created by simply examining the literature. As a result, the gap in knowledge drives our requirements elicitation process to the survey described in later sections.

2.1.5 Threshold Concepts

Regardless of object area, a threshold concept is a difficult conceptual "gateway" or "portal" that once crossed enables a deeper understanding of a subject [46]. There are five indicators of a threshold concept:

1. A threshold concept is transformative. Once understood, it changes a person's perspective [28].
2. A threshold concept is integrative. The concept brings together other concepts that were previously unknown [46].
3. A threshold concept is irreversible. Once grasped, the concept will not be easily forgotten [28].
4. A threshold concept is potentially troublesome. The concept is normally difficult to understand and may appear counter-intuitive [46].
5. A threshold concept is often a boundary marker. A threshold concept is often at the boundary of a subject [28].

With this definition of a threshold concept, and the information presented in Section 2.1.4, it is not surprising that object oriented programming has been identified as a threshold concept [28]. As a result, it is logical that there are most likely threshold concepts within object oriented programming as will be discussed in later sections.

2.2 LEGO MINDSTORMS EV3

The EV3 is the hardware the language will control. As a result, it is important to fully understand this platform. The EV3 was released in 2013 as the third evolution of the LEGO MINDSTORMS platform. Its predecessors were the RCX and the NXT. Known at LEGO as the PBrick, or PBR for short [15], the EV3 has been used by primary and secondary schools, universities, and even NASA [35].

From a terminology perspective, the EV3's hardware and software are both referred to as "EV3". To reduce confusion for the reader, we will in general refer to the hardware as the EV3 or EV3 brick and the software as EV3 MINDSTORMS. This convention will be dropped when it is obvious which part of the EV3 we are referencing.

2.2.1 EV3 Hardware

The EV3 uses a 32-bit ARM9 processor, Texas Instrument AM1808 at 300MHz and includes 64MB of DDR RAM [17]. The device has inbuilt Bluetooth and USB communication abilities. Additionally, there is a SD card slot for off-device storage [17]. The on-board USB port can be used for off device storage or to house a NetGear WNA110 USB dongle, thus enabling WiFi connectivity.

There are 4 input and 4 output ports for sensors and motors [17]. LEGO sells two types of motors (large and mini) and numerous sensors (light/colour, touch, ultrasonic, gyro, etc.) for the device [10, 11]. Additionally, other vendors have created sensors that are compatible with the EV3.

2.2.2 EV3 Programming Languages

There are a number of programming languages available for the EV3 Brick, from LabVIEW based languages such as EV3 MINDSTORMS [9], to text based versions such as Not Quite C (better known as NQC) [41].

Several of the languages were designed to be used from primary to graduate school. While this may seem an infeasible goal, several of these languages succeeded. One such language, RoboLab, has been used by kindergarten and graduate students [35]. Other languages focus on a particular demographic such as an artificial intelligence class [41].

Many EV3 programming languages are designed to be extensible—users are able to add new elements. In particular, we examine two extensible EV3 visual programming languages: EV3 MINDSTORMS and ScratchX.

EV3 MINDSTORMS by LEGO

EV3 MINDSTORMS is the programming language that LEGO distributes alongside the EV3 hardware. As a result, it is one of the most widely used platforms for programming the EV3. Versions of the software are available for Windows, OS X, iOS and Android. EV3 MINDSTORMS is built on National Instruments LabVIEW Web UI Builder and is a procedural language that supports multi-tasking. EV3 MINDSTORMS communicates with the EV3 via a USB cable or a wireless Bluetooth connection.

EV3 MINDSTORMS supports methods as “MyBlocks”. Users create MyBlocks by combining existing blocks into a single block. In addition to this, through the LEGO MINDSTORMS EV3 Block Developer Kit for Windows, LEGO enables advanced users to create new blocks using the NI LabVIEW Web UI Builder [15].

ScratchX

ScratchX is a version of Scratch that can be extended by third parties. Scratch is built on a Flash core as shown in Figure 2.2. ScratchX uses this same Flash core, but has a different shell that enables interaction with third party code as shown in Figure 2.3.

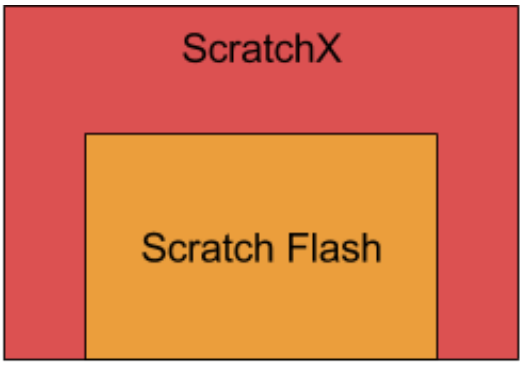

Figure 2.2: Basic Architecture of Scratch.

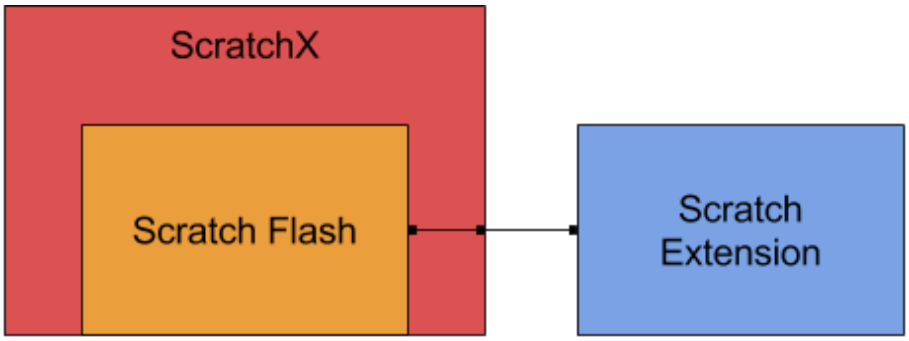

Figure 2.3: Basic Architecture of ScratchX.

Third party extensions are able to create Scratch blocks with new functionality. This new functionality can include interfacing with another platform. For example, extensions are available for Arduino microcontrollers, Twitter, and the EV3.

Aside from the creation of new blocks, ScratchX can be modified in other ways. This is because the three components of the architecture—the Scratch Flash core, ScratchX website, and Scratch EV3 Extension, are all open source. As a result, it is possible to modify any component in *the system*.

As a result of this multi-component architecture, a variety of languages are used to create ScratchX for EV3. In particular, the Flash core is written in ActionScript while the Scratch Extensions are written in JavaScript.

2.3 Object Oriented Programming

As the subject of this project is to create a programming language that helps transition students to object oriented programming, it is important to fully understand object oriented programming. Here we discuss the three fundamental components of object orientation—encapsulation, inheritance, and polymorphism [58]. This section provides a basic overview of these core object oriented concepts. Readers already familiar with object oriented programming may elect to skip this section.

2.3.1 Encapsulation

Encapsulation allows objects to hide their data members and as a result is sometimes referred to as data-hiding. By hiding information, programmers are encouraged to create modular designs [48]. Interestingly, it can be argued that inheritance hurts encapsulation since child classes are allowed to access information from their parents in many programming languages [54].

2.3.2 Inheritance

Inheritance creates an "is-a" relationship between different classes [42]. A class, called the child, may claim inheritance from a second class, called the parent. Once a parent class is noted, the child receives the data members and functions associated with the parent. Depending on the language's implementation of encapsulation, the child may not receive items marked private.

2.3.3 Polymorphism

Polymorphism, meaning many forms, enables different classes with the same parent to be used in the same location. This occurs when a function requires a parent class as a parameter. A purely object oriented language recognizes the "is-a" relationship through inheritance and will therefore accept any of the child classes in the parent's place. This can be allowed because we know the child includes the parent's interface [29]. As a result, a child may always be used in the same way as a parent.

While often referred to as a core concept of object orientation [56], polymorphism can be viewed as an extension of inheritance [58]. This is because polymorphism is a more complex implication of the "is-a" inheritance relationship.

2.4 Visual Programming Languages

Visual programming languages use pictures instead of words to communicate meaning [37]. As a result, visual languages have unique properties. Some of these properties enable visual languages to be easier to learn [27, 39]. For example visual languages are better at expressing structure, enable pre-literate children to program, and can provide a superior resemblance to the surrounding world [27]. Additionally, several articles point to the fact that "a picture is worth a thousand words" [27, 37], and is thus able to convey more meaning. Because the brain considers a picture as a single unit, it is therefore easier to understand multiple items about a command in a visual language rather than a text based language [27]. These properties of visual programming languages will be useful for *the system* as shown in later chapters.

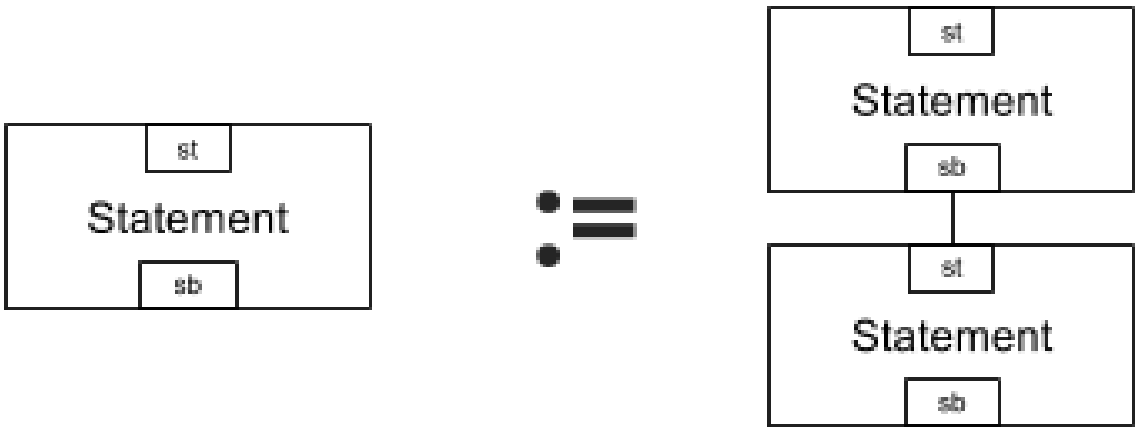

Figure 2.4: Example Graph Grammar

2.4.1 Syntax of Visual Programming

As with any programing language, the creation of a visual language begins with defining the syntax of the language. For textual languages, this is done using context free grammars. These grammars define how each part of the language interacts and combines to form the overall program. The language is built up from λ which represents the empty set. Using a set of rules, called productions, the language can be built up by substituting the left hand side of the production for the right hand side. The grammars are considered context-free as they map a single entity, instead of a group of entities, to a number of different options. Context free grammars are an important tool in creating a new programming language, but they are inadequate for this project as they are based on text instead of graphics.

Instead of using only text, visual languages are defined with shapes, lines, and text [38]. It is still critical to ensure visual languages have well defined specifications [23]. Due to the graphic nature of visual programming, we can use graphs to represent the most basic form of a visual language [37]. Therefore, in visual programming languages graph grammars are used. While numerous flavors of graph grammars are available, their underlying principles are the same [37, 38].

Graph grammars define different nodes and show or describe how they can be combined to form the overall program. Put another way, graph grammars define the only allowable transformations on the graph. By defining these valid transformations, graph grammars ensure proper syntax in the same manner as context-free grammars. Just like context-free grammars, graph grammars create allowable transformations through production rules. A production rule contains two graphs as shown in Figure 2.4. The left-hand side of the production can be transformed into the right-hand side. Figure 2.4 effectively says a statement can be comprised of multiple statements.

Graph grammars are often not context-free as this restriction has proven too difficult

to create many visual languages [50]. As a result, we are not limited to having a single node on the left hand side of the production as we would be in a context free grammar. Instead, the general rule is to ensure the left-hand side of each production has fewer elements than the right-hand side. This allows the language to be built up through the graph grammar. Examples of graph grammars abound in the literature [25, 45].

The theory behind creating graph grammars presented here will be used in later chapters in the design of the language.

2.5 Abstraction

As humans our ability to abstract starts at the beginning of our lives and develops as we mature [53]. Abstraction is our "mental process in which some attribute or characteristic is observed independently of other characteristics of an experience as a whole." [59]. Put more simply, abstraction is simplifying and finding commonalities between different elements [43]. These elements could be objects, experiences, or other items. From Picasso's cubism and Louis Armstrong's jazz music to maps of train networks and classes in object oriented programming [43], abstraction is used in numerous and diverse fields.

In computing, abstraction is a key concept. Abstraction is obviously used in object oriented programming through inheritance and encapsulation. Beyond this, abstraction is used to create models and this is a critical tool for engineers regardless of their area of expertise [43].

Studies have shown numerical abstraction occurs for infants as young as 6 months old [55]. However, these abilities are nowhere near the level needed to understand larger abstract concepts such as object oriented programming. Additional studies indicate that math is a critical tool to helping students develop their abstraction abilities [43]. It is important to note the importance of mathematics in developing abstraction abilities. In future sections this point will be critical to understanding some of the arguments presented.

In this chapter we have provided the reader with an understanding of a number of concepts that are critical for the overall project. By understanding computer science education, we can see where this project fits into the current landscape. As the project aims to develop a language for the EV3, it is obvious that the hardware and software associated with the platform is understood. To assist the transition from visual procedural to textual object oriented programming, it is critical to understand both of these paradigms and language styles. Because abstraction is a critical ability to learn object oriented programming, it is important to understand the psychology behind abstraction abilities development in children. Perhaps the most important part of this literature review is the knowledge gap in the struggles students face when transitioning to object oriented programming. This gap drives the survey that will be discussed in the next chapter.

Chapter 3

Survey Methodology

While the literature review provided the background information required for the project, there was insufficient information for understanding why it is difficult to transition to object oriented programming. As a result, a survey was created to fill the knowledge gap. Here, we describe the methodology behind creating the survey and recruiting participants.

To understand how to create a programming language to assist students to transition to object oriented programming, it is critical that we understand what makes this conversion so difficult. As discussed in Section 2.1.4, we know that this transition is difficult but there is very little research on what makes it a challenge. This makes it more difficult to define *the system* as there is limited information to direct the project. To make up for this lack of information, a detailed requirements elicitation approach is required.

As a member of the target audience, the author has her experiences as a starting point, but this is purely anecdotal evidence and is thus not suitable for a rigorous academic approach. Therefore, it was decided that a survey should be created and conducted to better understand the issues that the language needs to address.

All questions that involved participants responding on a scale were done on a 1-6 scale. This was chosen to give a significant range of options. An even number of options were provided to participants to remove a neutral answer.

3.1 Areas of Inquiry

The survey was created to find answers the following five questions:

- Is there an optimal age to learn object oriented programming?
- What object oriented concepts do students find easy to understand?
- What object oriented concepts do students struggle to learn?
- What factors influence the difficulty of learning an object oriented concept?
- How difficult is it to learn the following object oriented concepts?

- File Interaction
- Method Calls
- Accessing Variables
- Public versus Private
- Inheritance
- *Abstract* Methods, Variables, and Classes
- Polymorphism
- Encapsulation

Through the survey, we aim to discover what object oriented concepts cause students to struggle along with which concepts are easy to grasp. To accomplish this, participants were asked open ended questions about what concept they found the easiest and what concept they found the most difficult. The goal of these questions was to find concepts which were not originally considered. After both of these short answer questions, respondents were asked what made the concept easy or difficult. The hope was that from the survey results enough common areas of difficulty would emerge to determine attributes of *the system*.

In addition to the qualitative data these questions will provide, quantifiable data was collected. Because we know students need abstraction ability and this is not fully developed in young children, the age at which respondents learned object oriented programming was recorded. This age will be compared to how difficult on a 1-6 scale they found object oriented programming. The hope was to find an age at which object oriented programming is easiest to learn.

In addition, participants were asked how difficult on a 1-6 scale they found the predetermined concepts. The concepts were chosen based on the author's experience and core object oriented concepts. By having participants score each concept, it became possible to directly compare concepts and see which ones are more difficult.

To create an unbiased survey, short answer questions occurred before the ranking questions.

3.2 Recruitment Methods

The target audience is those whose first programming language was procedural and have since learned an object oriented programming language. Participants of the survey were required to be over sixteen for consent purposes.

Survey recruitment was focused on FIRST Robotics Competition and FIRST Tech Challenge students and alumni as this demographic meets the requirements for survey participation. The survey was promoted through posts to the Australian FIRST Robotics Competition e-mail list, social media groups for FIRST Robotics students and alumni, and Chief Delphi which is the major online forum for FIRST Robotics. In addition three

social media pages and several colleagues of the author publicized the survey through social media channels. A copy of the survey recruitment brochure is in Appendix B.2.

Because the survey recruitment focused on a specific audience, it is possible for the data to be skewed. This has been considered and where appropriate it enters into our discussion of the survey results.

Chapter 4

Survey Results

The survey was open from Tuesday May 10th 2016 and the data studied here was collected on May 24th 2016. In that time frame, 146 unique visitors started the survey with 56 participants completing all questions.

Before the data was processed, the information given was checked to be certain survey respondents met the criteria. Six survey responses were found to be invalid as the participants learned object oriented programming before a visual-procedural language. These responses were discarded and have not been included in the following analysis.

With the information gathered validated, the survey was analyzed in respect to each area of inquiry outlined in Chapter 3.

4.1 Age to Learn Object Oriented Programming

The age object oriented programming was learned was considered in comparison to the difficulty of learning object oriented programming. The data suggest that those who learned object oriented programming at fourteen found it easier than any other age group. To illustrate this, consider the box plots of difficulty for each age group as shown in Figure 4.1. The lack of one, or both, whiskers for some age brackets occurs from the relatively few possible answers (only six distinct answers are possible) that participants could provide.

Using a t-test, with a threshold of $p = 0.05$ the difference in reported difficulty between fourteen year olds and the general survey respondent was found to be statistically insignificant. Considering the statistically low number of participants, the threshold was increased to $p = 0.1$. At the $p = 0.1$ level, the difference was significant.

On a positive note, fourteen years old was the most common age to learn object oriented programming. Nearly 20% of respondents learned object oriented programming at this age.

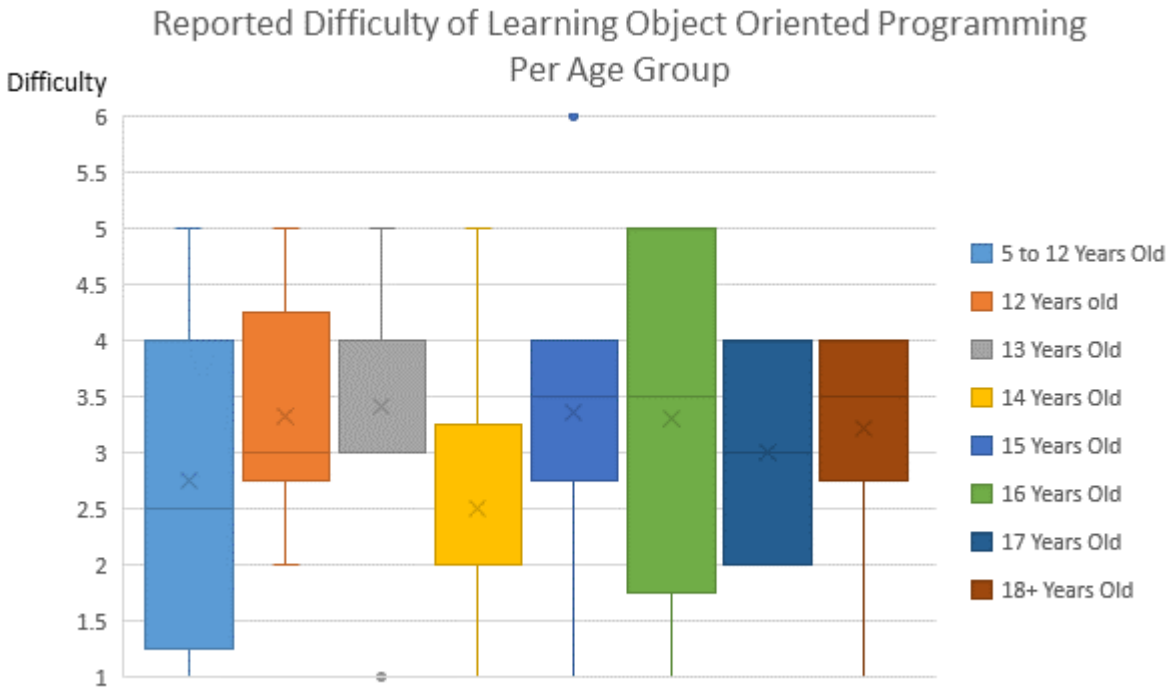

Figure 4.1: A box plot of reported object oriented programming difficulty for each age group. The 'x' marks the average and the box shows the upper and lower quartile range. The whiskers begin at the box and extend to the last data point that occurs within one and a half times the inter-quartile range. Any data points outside this range are considered outliers (in common with usual statistical practice) and are represented by a dot.

4.2 Easy Object Oriented Concepts

While this project focuses on the difficulties associated with learning object oriented programming, it is important to also understand which concepts are easy. This enables a better understanding of the entire problem. To accomplish this, an analysis of object oriented concepts respondents reported as easy is conducted.

The analysis of easy to understand object oriented concepts began by grouping responses. In total, nine categories were found: classes, encapsulation, inheritance, not applicable, objects, other, polymorphism, procedural concepts, and syntax. The "other" category was reserved for singleton answers. To view the complete responses and their categorization, please refer to Appendix C.1. A visual breakdown of the responses per category is shown in Figure 4.2.

Of particular note is the large number of respondents who did not provide object oriented concepts. Between the "procedural concepts", "not applicable", and "syntax" categories, 27% of respondents thought a component outside of object orientation was the easiest to understand. This underscores the difficulty of learning object oriented concepts

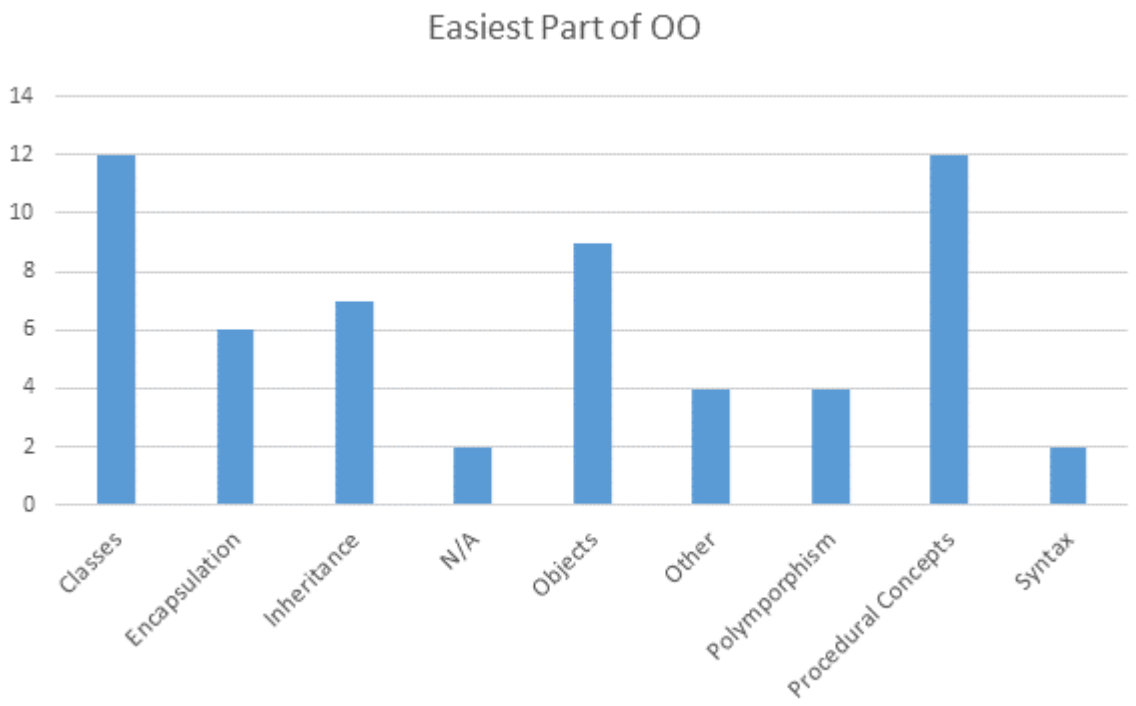

Figure 4.2: Number of responses per category of easy object oriented concepts.

and the importance of this project.

Aside from procedural concepts, the most common answer was classes followed by objects. It is suggested that this occurs because these are some of the first concepts taught in object oriented programming classes. As a result, students are given a significant amount of time to understand these concepts.

4.3 Difficult Object Oriented Concepts

Considering the easy to learn object oriented concepts only provides half of the information required to understand the problem space. The other half comes from the difficult parts of learning object oriented programming. Here, an analysis of object oriented concepts respondents reported as difficult is conducted.

As with the analysis of easy to learn object oriented concepts, this analysis began by grouping responses into appropriate categories. In total, ten categories were discovered: *abstract*, class versus object, classes, encapsulation, inheritance, objects, other, overloading, polymorphism, procedural concepts. The "other" category was reserved for answers with no similarities to other responses. To view all the responses and their categorization, please refer to Appendix C.2. A graph of the responses per category is shown in Figure 4.3.

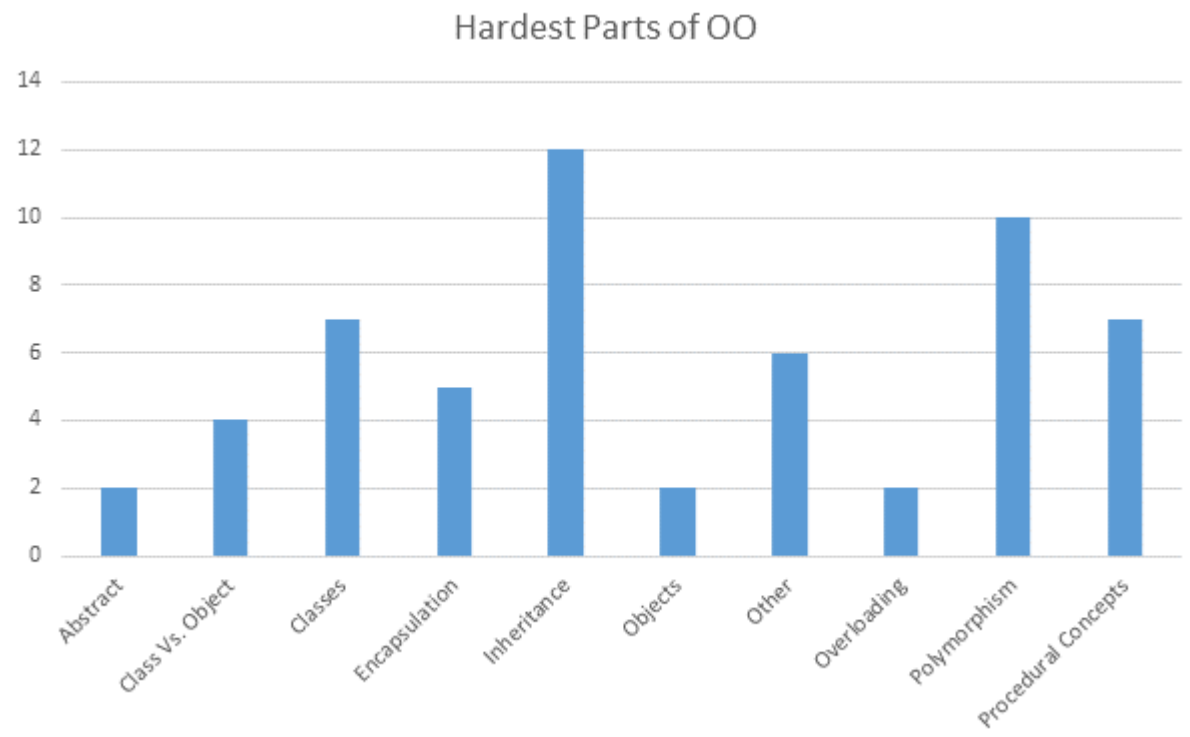

Figure 4.3: Number of responses per category of difficult object oriented concepts.

The most common difficult to understand concepts in object oriented programming that respondents gave were inheritance and polymorphism. Together, inheritance and polymorphism accounted for 44% of the responses. Interestingly, both inheritance and polymorphism also appeared on the easiest object oriented programming list. This overlap between the lists is not unique. Six concepts (classes, encapsulation, inheritance, objects, polymorphism, and procedural concepts) appear on both lists.

Only three concepts were unique to the difficult to learn category: *abstract*, class versus object and overloading. This suggests that these concepts were extra difficult as no respondent considered them as the easiest concept to grasp.

4.4 Factors that Influence Difficulty

When asked why a particular concept was easy, respondents offered a wide range of opinions. A complete list can be found in Appendix C.3. A few patterns emerged upon examining these factors. In particular, many respondents were assisted in their learning by:

- An explanation a tutorial or teacher provided.
- The object oriented concept being intuitive.

Table 4.1: Average reported difficulty (to two decimal places) for pre-determined object oriented concepts.

Object Oriented Concept	File Interaction	Method Calls	Accessing Variables	Public versus Private	Inheritance	*Abstract* Methods, Variables, Classes	Polymorphism	Encapsulation
Average Reported Difficulty	3.13	2.38	2.09	2.41	3.41	3.89	4.00	3.80

- A component of the programming language.
- Visual feedback.

It is interesting to note the inclusion of visual feedback in this list. The literature supports this finding. In particular, it was noted in Section 2.1.2 that interactive languages with visual feedback would assist students as it enables an understanding of the impact of the code changes.

When respondents were asked what made a particular concept difficult, a similarly wide range of answers were given. A complete list can be found in Appendix C.4. Some of the most common reasons respondents provided were:

- An explanation a tutorial or teacher provided.
- Could not see usefulness or relevancy.
- Nothing to compare the concept to from previous languages.
- General confusion and difficulty.

It should be noted that some concepts appear on both the *easy* and *difficult* lists of factors that influence the difficulty of learning an object oriented programming concept. In particular, a tutorial's or teacher's explanation occurs in both lists. This suggests that how the student is taught is a markedly contributing factor to a student's ability to learn object oriented programming.

4.5 Scored Difficulty of Object Oriented Concepts

In addition to giving short answer responses, participants were asked to score the difficulty of a number of pre-determined object oriented related concepts on a six-point scale. The average of these responses is shown in Table 4.1.

Based on the responses, it is possible to divide the concepts into those that are hard and those that are easy to learn. In particular, file interaction, method calls, accessing variables and public versus private, are all suggested easy concepts. *Abstract* methods, variables and classes, polymorphism, and encapsulation appear to fall into the difficult category. This difficult category aligns with the short answer question discussed in Section 4.3.

Chapter 5

System Requirements

With the survey results analyzed, our focus shifts to setting the requirements of *the system*. These requirements are based on the literature review, the author's experience and judgment, and the survey results.

As a matter of terminology, *the system* refers to the new programming language, development environment, and associated supporting infrastructure. The requirements outlined below consider the entire *system* and are not limited to the new programming language.

This section is based on 10.1109/IEEESTD.1998.88286 recommendations for software requirements specification documents [40]. Here, *the system* is considered from a number of perspectives to elicit requirements. Based on this, the requirements are provided in Appendix A.

5.1 Interfaces

The system will need to interface with a variety of other components. These components are defined and the interfaces specified in sections below.

5.1.1 System Interfaces

The system needs to interface with a computing device and the EV3. In particular we will require that the programming should be completed on a Windows XP, Windows 7 or Windows 8/8.1 machine as these three operating systems have over 85% market share [20]. System interface requirements are found in Table A.1.

5.1.2 User Interfaces

In order to define the user interfaces, it is important to first know the target audience. Based on the national curricular requirements and robotics programs considered in Section 2.1.1, *the system* is targeted for students in years 5-8.

To help bridge the gap between the visual procedural languages learnt in years 1-4 and object oriented programming, *the system* will be a visual programming language. This will also decrease the literacy requirements for younger children learning *the system*.

The user interface requirements appear in Table A.2.

5.1.3 Hardware Interfaces

The hardware interfaces can be divided into two main categories; the computer and the EV3 ecosystem. On the computer side, there are no major requirements as the OS ensures a well-defined interface. For the EV3, *the system* must be able to properly interface with a variety of input/output hardware. For the purposes of this project, we have limited these inputs and outputs to the hardware that comes in the EV3 LEGO Education and LEGO Retail kits, including the input/output on the EV3 brick. The exact details are outlined in Table A.3.

5.1.4 Software Interfaces

Aside from the operating system interfaces outlined in Section 5.1.1, *the system* will also need to interface with the operating system on the EV3. In particular, the EV3 has a Linux kernel [16] which can be found on GitHub [12].

The software interface requirements appear in Table A.4.

5.1.5 Communications Interfaces

The EV3 supports two main communication protocols for downloads—Bluetooth and Micro USB. As a result, at least one of these communication protocols will be used. Given the complex requirements for adapting the EV3 for WiFi, it was decided this communications protocol would not be supported. The communication interface requirements appear in Table A.5.

5.2 Nonfunctional Requirements

The system's nonfunctional requirements relate to no single function of *the system*, but rather its overall implementation.

5.2.1 Reliability

In the author's ten years of competitive LEGO robotics experience, the majority of firmware for LEGO MINDSTORMS programming languages will corrupt itself within two to four weeks of extensive use. With this in mind, the firmware is expected to have a Mean Time Between Failure (MTBF) of 3 weeks under extensive usage.

The reliability requirements appear in Table A.6.

5.2.2 Availability

The availability requirements are straightforward and therefore are only shown in Table A.7.

5.2.3 Maintainability

Several LEGO MINDSTORMS languages support the ability to download updates and patches [3] [19]. Perhaps this is due in part to the extensive user feedback LEGO Education receives once the product is released. As a result, *the system* created must also enable updates to be sent after *the system* is released.

The maintainability requirements appear in Table A.8.

5.3 Functional Requirements

The main aim of the project is to create a programming language that helps students transition from procedural to object oriented programming. The key detail to recall is that the project goal is not to create an object oriented programming language. This distinction is critical to understanding the functional requirements as it dictates many decisions made.

Here, a further explanation of the choice of object oriented concepts selected is provided. A full list of the functional requirements appears in Table A.9.

5.3.1 Data Encapsulation

Data encapsulation was chosen for a number of reasons. Firstly, we believe the target audience can understand the concept. Data encapsulation is based on information hiding and secrecy. Children as young as eight have been shown to understand the concept of secrecy [44]. We believe this concept easily extends to objects thereby making it a good concept to include in the language.

Additionally, data encapsulation was identified in Section 4.5 as a potentially difficult concept for students. As a result, providing transitional assistance in this area can reduce the overall difficulty of moving to object oriented programming.

For the purposes of this language, data encapsulation will include allowing public and private attributes and methods, and showing the object's interface. This provides the functionality required for encapsulation and assists the student's understanding.

5.3.2 Inheritance

To help students transition to object oriented programming, some form of abstraction is needed. As described in Section 2.3.3, polymorphism can be considered an extension of inheritance. As a result, it is logical to introduce inheritance before polymorphism. This

allows students to understand the underlying concept of inheritance before extending this knowledge to inheritance.

In Section 4.3, we saw that inheritance was reported as the most difficult object oriented concept to learn by the largest number of participants. Yet, in Section 4.5, the data implies that inheritance is the easiest of the object oriented concepts that can be classified as difficult to learn. This dichotomy suggests that inheritance can be mastered without large amounts of difficulty if properly considered.

As a result, we have decided to implement inheritance in the programming language. This will help students master inheritance and prepare them for learning polymorphism.

5.3.3 Polymorphism

As discussed in Section 2.3.3, polymorphism is a more complex implication of inheritance. With this in mind, it makes sense for students to learn inheritance before polymorphism. In fact, it is possible to argue that polymorphism can be taught by inheritance. As a result, we have chosen to focus on implementing inheritance.

This decision is supported by the data collected through the survey. As shown in Section 4.5, polymorphism is potentially the most difficult object oriented concept to learn.

This decision does not violate the project's goals as we are are creating a programming language to assist in the transition to object oriented programming. This goal significantly differs from the creation of an object oriented programming language.

5.3.4 *Abstract* Classes

The survey data suggests that *abstract* items (methods, variables, and classes) are the second hardest to learn object oriented concept. As a result, *abstract* classes arose for consideration in the overall project. Unlike encapsulation and inheritance, *abstract* classes are not a core object oriented concept.

However, *abstract* classes are an often used object oriented concept. Since *abstract* items ranked the second hardest concept and multiple respondents said it was the hardest object oriented concept, *abstract* classes were included in the language's requirements.

Chapter 6

Language Design

With the language's requirements set, the project can continue to the design phase. We begin this process by examining two options for a platform on which to build the language. After that, we present a syntax for *the system* that meets all the requirements outlined in Chapter 5.

6.1 Language Platform

To enable quick prototyping and creation of *the system*, it was decided to build the language on top of an existing platform. This reduces the overhead of creating a visual language. Two languages were considered—ScratchX and EV3 MINDSTORMS. Both languages run on the EV3 with no extra effort required. Here, we discuss each language platform and how it would work as a platform for this project.

6.1.1 EV3 MINDSTORMS

EV3 MINDSTORMS is the default software that LEGO provides alongside the EV3 and was discussed in Section 2.2.2.

Because EV3 MINDSTORMS comes alongside the EV3 brick, the target audience is most likely already familiar with EV3 MINDSTORMS. This is probably the most familiar language for those learning to program the EV3. EV3 MINDSTORMS supports both major communication protocols—Bluetooth, and USB.

The Software Development Kit provided by LEGO allows anyone to create new blocks using National Instruments LabVIEW Web UI Builder. For example, a square function was added to the provided sample data icon. This was properly built into a block that was successfully tested on the EV3. The block is shown in Figure 6.1 and the underlying LabVIEW is shown in Figure 6.2.

Perhaps the biggest downside to using EV3 MINDSTORMS is that there is no underlying object orientation. LabVIEW is a dataflow language, making it more difficult to transform into an appropriate form. Additionally, because the software is not open source, it is more difficult to modify the language aside from the creation of new blocks.

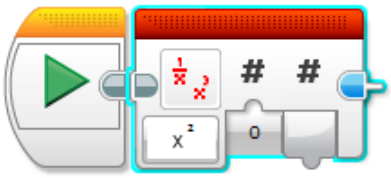

Figure 6.1: Square function added to the Sample Data Block.

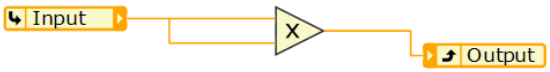

Figure 6.2: Underlying LabVIEW for the square function.

6.1.2 ScratchX

ScratchX is an extensible build of Scratch which was discussed in Section 2.1.3. ScratchX is unusual in that it only runs in a web-browser. Internet Explorer or Firefox are the recommended browsers *the system* supports. By including the EV3 extension for ScratchX, the language can be used to control the EV3.

ScratchX for EV3 connects to the EV3 via Bluetooth with no firmware download required. Programs are not stored on the EV3, but are communicated from computer via Bluetooth to the EV3 brick.

As of May 29, 2016, Scratch had over 12 million registered users [14], making the programming language familiar to many students in the target audience.

ScratchX is built on the same Flash core as Scratch. This is then combined with a different webshell to form ScratchX. Extensions can be developed by anyone and added into ScratchX. All components of this ecosystem are open source. This enables developers to extend the language in any way they chose. This includes changing a block's shape, creating new *data* items, and adding new provided functions. The ScratchX loaded with the EV3 extension can be seen in Figure 6.3.

Many of the downsides of using ScratchX come from the nature of open source projects, In particular, Scratch is poorly documented. The code is barely commented and there are several layers of the project to navigate through.

6.1.3 Platform Selection

Based on the above information, ScratchX was selected as the language's platform for a number of reasons. While both EV3 MINDSTORMS and ScratchX were able to meet the

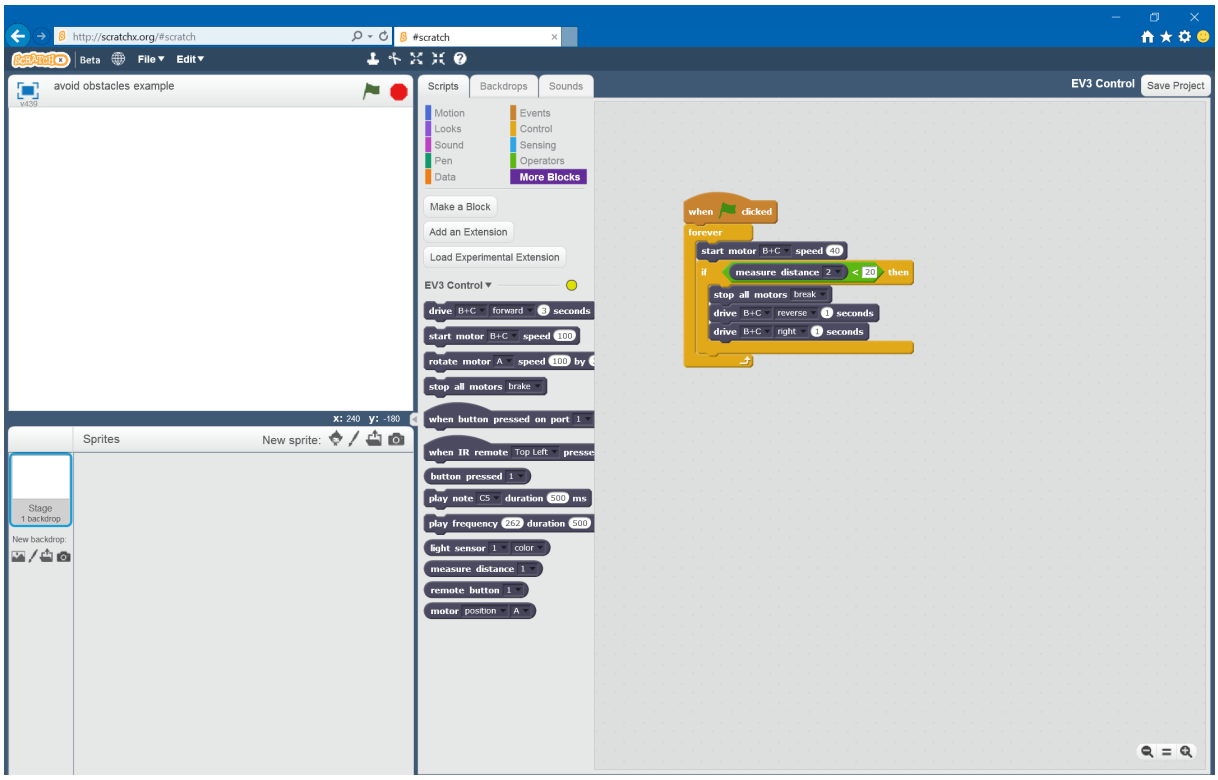

Figure 6.3: ScratchX with the EV3 extension and a sample sample program at the **Stage** level.

requirements, ScratchX being open source rendered it the easier to modify as desired. This increased flexibility allowed the requirements and design to dictate the language instead of the platform chosen. While Scratch lacks documentation, the code is well organized and numerous tools are available to assist in debugging ActionScript programs.

6.2 Defining Scratch

The language's syntax will be defined using graph grammars. Because the language will be built on ScratchX, we use this as the beginning of the syntax. As a result, before we can define the language's graph grammar, it is critical to understand the ScratchX programming language and environment.

It is important to note that the graph grammars of Scratch and ScratchX are both defined in the Scratch Flash core and are therefore the same. To define the Scratch programming language, one would assume we would use a graph grammar as discussed in section 2.4.1 as Scratch is a visual programming language. However, Scratch defines certain elements using a user interface. In particular, a number of elements are created using a button while their parameters are defined using a dialog box. As a result, we must examine the graph grammar and user interface in more detail. By ensuring this

firm foundation in the Scratch language, the new programming language can be similar to Scratch thus easing the transition to object oriented programming.

6.2.1 Scratch Sprites

To understand the Graph Grammar, it is critical to remember that in Scratch users program through characters called **Sprites** on a **Stage** as was discussed in Section 2.1.3. A Scratch project can contain many **Sprites**.

It is also important to remember that **Sprites** are similar to objects in many ways. For example, in Scratch, each **Sprite** has a unique location to build its programs, in a similar manner to how an object oriented language would have a file for each class. **Sprites** have appearances, *data* members, and functions.

6.2.2 The Stage

The Scratch **Stage** is very similar to a **Sprite** and can be thought of in the same manner. In particular, the **Stage** has appearances (costumes), *data* members, and functions.

One of the major differences between the **Stage** and **Sprites** is there is only one **Stage** per Scratch project. As we will see in Section 7.4, the properties this 1:1 relationship ensures have useful implications.

6.2.3 Scratch Costumes

A costume object contains an image and associate directional information (x-direction, y-direction, x-rotation, y-rotation, layer, etc.). Each **Sprite** and the **Stage** contain a list of costumes that represent the multiple forms the object can take.

6.2.4 Scratch Blocks

There are three major types of blocks in Scratch—hats, *procedures*, and reporters. The graph grammar defines how these three types of blocks can be combined to create the overall programming language. Each combination of blocks is referred to as a "script".

Each script begins with a hat node which can either be an event or a *procedure* definition. Under the hat, a number of *procedure* blocks can be placed to create a stack. When a hat is triggered, the *procedures* in the stack are executed procedurally. Event hats are triggered when the event occurs while *procedure* definition hats are triggered when the defined *procedure* is called. **Sprites** and the **Stage** can have multiple scripts, enabling different functionality based on which hats are triggered.

A *procedure* is the general purpose block of the Scratch environment. It is responsible for the majority of commands and actions. For example, if a **Sprite** moved across the **Stage**, this action would be caused by a *procedure*.

Reporters are the remaining block type in Scratch. These blocks return a value and are often included in *procedures*. Reports can return *data* (a string or integer), *list*, or boolean.

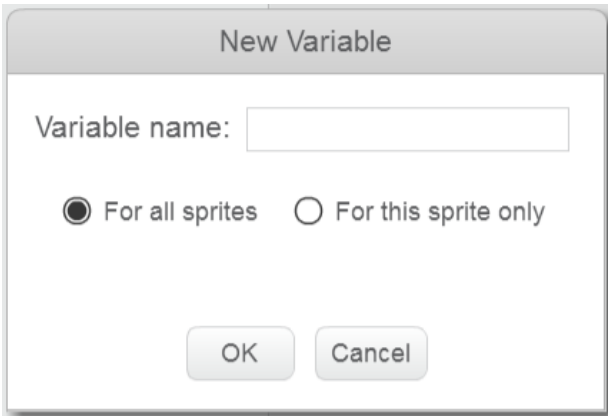

Figure 6.4: The dialog for defining the parameters of a *data* or *list* element in Scratch.

Data and *list* reporters have a different shape to booleans. This is part of Scratch's ability to create automatically syntactically correct programs. Because booleans are a different shape to *list* and *data*, they cannot go in the same location. This ensures the program is syntactically correct before compilation.

Blocks will be explained in more detail in Section 7.2.2.

6.2.5 User Interface

It is important to note that there are user interface elements within Scratch that provide additional functionality. This simplifies the graph grammar. For example, *data* and *lists* are created using a user interface button instead of a block. To define parameters of a *data* or *list* element, the dialog box shown in Figure 6.4 is used. This means the Scratch programming language is more complex than the graph grammar provided. As a result, while the graph grammar outlined below provides the underlying basis for Scratch, it is important to remember there are additional components that are not defined in this manner.

Elements are often created using a button while their parameters are defined in a dialog box. In particular, buttons and dialog boxes create and set parameters of the following Scratch components:

- *Data*
- *Lists*
- *Procedure* Definitions
- **Sprites**
- **Stage**
- Costumes

For example, *data* and *lists* are set as public or private in a dialog box. A **Sprite** is created with a user interface button. The inputs to a *procedure* are defined in the *procedure* definition dialog box.

Understanding how the user interface buttons and dialog boxes are currently used in Scratch is critical for later determining which components of the new language will be implemented via new user interface buttons and dialog box options instead of updates to graph grammar productions.

6.2.6 Scratch Graph Grammar

The parts of Scratch not controlled by the user interface are defined with the graph grammar productions. Each script begins with a hat block as shown in Figure D.1.

Procedures can be simple blocks as shown in Figure D.5. *Procedures* also include *if*, *if-else*, and *loop* statements as shown in Figure D.9, Figure D.10, and Figure D.11 respectively.

The graph grammar production rule for reporters can be seen in Figure D.8.

The full set of productions that define ScratchX's graph grammars are provided in Appendix D.

6.3 New Language Definition

To create the new programming language, the majority of changes will occur through new user interface buttons or dialog box option. An example of an updated dialog box is shown in Figure 6.4. Here, we describe how Scratch will be extended in each of the major areas.

6.3.1 Updated User Interface

To consider how the user interface buttons and dialog boxes will be updated, we remember their general purposes are to create and set parameters of elements.

It is logical to include the data encapsulation elements in a dialog box as this edits the parameters of the *procedure* definition. This decision is supported by recalling that for *data* and *lists*, the public versus private setting was included in the dialog box.

To create a cohesive language, *abstract* classes should be created in the same way as a **Sprite**. As **Sprites** are created with a buttons, this is how *abstract* classes will be created as well.

Finally, a **Sprite's** parent can be thought of as a parameter of the **Sprite**. As a result, we have chosen to include setting the **Sprite's** parent in a dialog box.

6.3.2 Updated Graph Grammar

With this large amount of functionality to be added via the user interface as defined above, it could appear that the graph grammar does not need to be updated. This is

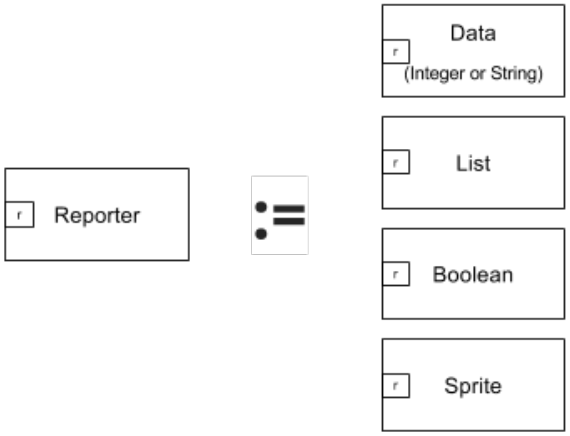

Figure 6.5: The changes required to the reporter block for the new programming language.

incorrect. In particular, to make **Sprites** passable entities, a new reporter will need to be created. This cannot simply be done through the user interface buttons and dialog boxes as the **Sprite** entity needs to be passed between blocks. As a result, this will change the graph grammar for reporters from that in Figure D.8 to that in Figure 6.5.

It should be noted that to maintain the iconic puzzle piece structure and automatic syntactic correctness of Scratch, a unique shape will be required for the **Sprite** reporter. This unique shape will ensure that a **Sprite** cannot be used where a boolean is required.

In this chapter we have designed the new language. We began by selecting ScratchX as the platform on which to build the language. With this set, we examined how ScratchX is defined to better understand the base of the new language. With this understanding, we have shown how the new language's features will be implemented to create a cohesive *system*.

Chapter 7

Language Implementation

In this chapter the implementation of the language is discussed. As part of the implementation, tools were selected for development and source code control. Next, the Scratch architecture is described to provide the reader the background required to understand the implementation. With this basis, we explain the implementation of the overall language.

7.1 Tool Selection

In the process of obtaining the Scratch Flash core, the code downloaded had to be compiled to an executable form. On the source code page, the creators of Scratch recommend the Intellij IDEA integrated development environment. Intellij IDEA provides support for ActionScript. This support includes features from debugging to text highlighting. Intellij IDEA is provided free for students and therefore was within the budget outlined in Section 1.2.3.

Before the language was implemented, a source code management tool was selected. Because Scratch, ScratchX, and the EV3 extension for Scratch are all available on GitHub, we began by considering this tool. GitHub is integrated into Intellij IDEA and enables branching, committing, reverting, and merging files. GitHub also enables the project to be open source, a requirement of the licensing agreement that all the relevant projects were released under. Additionally, by having the project on GitHub, any changes made to Scratch, ScratchX, or the EV3 extension for Scratch can be merged into the new language. As a result, we elected to stay with GitHub as the source code management tool for this project.

7.2 Scratch Flash Code Architecture

As discussed in Section 2.1.3, Scratch is an object-based language. As a result, it is unsurprising that the underlying architecture is object oriented. As a Flash application, Scratch is written in ActionScript. Here, some critical components of Scratch's architecture are discussed to give readers a background to enable an understanding of the new

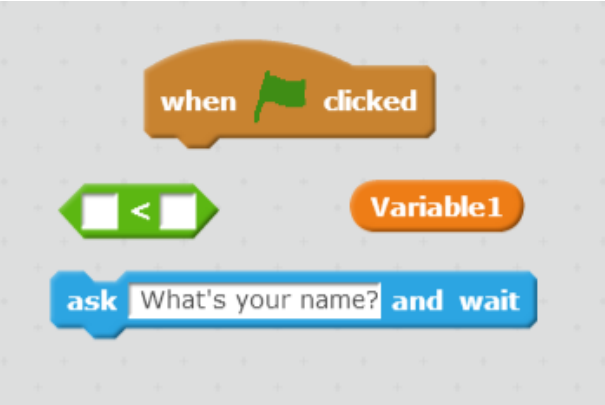

Figure 7.1: A hat (top), boolean reporter (middle left), *data* reporter (middle right), and *procedure* (bottom) blocks.

language's implementation.

7.2.1 Scratch Class

The Scratch class is the top-level of the application. It can be thought of as the "main" method. The Scratch class contains the Runtime, Interpreter, and user interface components. Additionally, the Scratch class keeps track of the currently *viewed-object* which is crucial for the user interface. Depending on what type of object is currently being used, the interface will adapt appropriately. This is critical for Scratch to function properly as we shall see in the following sections. The Scratch Object also allows for the creation of new Scratch **Sprites**.

7.2.2 Blocks Class

Blocks are the equivalent to most programming languages' statements. They have a form (appearance on the screen) and function. The blocks have their own graph grammar that maps to the different shapes. The blocks can be stacked and combined to form a program. This grammar was discussed in Section 6.2.

In general, these blocks fall into three shapes (hats, reporters, and *procedures*) which can be seen in Figure 7.1 and were described through graph grammar productions in Section 6.2. Hats are *procedure* declarations. This can arise through either the user defining a new *procedure* or declaring what should happen when an event occurs. Reporters can be given to *procedures* and used within the method. The shape of a reporter can vary slightly depending on the data type it returns. Finally, *procedures* are the most common block and have Scratch's iconic "puzzle piece" design.

As a matter of terminology, each stack of blocks that can be run is referred to as a script. It is possible, and expected, that each Scratch Object will have multiple scripts.

In Scratch, blocks execute on the *do-object* set by the Scratch Runtime and discussed in more detail in section 7.2.4.

7.2.3 Scratch Objects Class

Scratch Objects are the underlying support for the things users can program, as such they have a number of common properties. In the Scratch language, the key attribute of these objects is they can be programmed individually. The Scratch Objects class serves as a super class for the Scratch **Stage** and Scratch **Sprite** objects discussed below.

Stage Class

The Scratch **Stage** is the backdrop of the scene. In addition to this visual role, the **Stage** can be scripted in the Scratch environment. Scratch **Stage** objects do not have access to all of the *procedures* available to **Sprites**. For example, the "motion" set of blocks are only available to **Sprites**. This is managed by the palette as discussed in more detail in Section 7.2.6.

Sprite Class

Sprites are the major component of the Scratch programming language. From an object oriented perspective, **Sprites** are a combination of classes and objects. They can be cloned in a similar manner to instantiating a new object, but cloning is not required. **Sprites** are capable of executing all the Scratch blocks.

7.2.4 Scratch Runtime Class

The Scratch Runtime class contains all the logic to start hat blocks and find *data* or *lists*. To accomplish this, the Scratch Runtime class is able to search all Scratch Objects for the needed information.

The Scratch Runtime keeps track of the current *do-object* which is critical to the the Interpreter's execution. The *do-object* is usually the Scratch Object that contains the script being executed. When a user defined *procedure* is called, the object calling the *procedure* is the *do-object*.

7.2.5 Interpreter Class

The Interpreter class is self explanatory (by the name). It takes the scripts (stacks of blocks) and interprets them into the required actions.

7.2.6 User Interface Classes

The Scratch user interface is comprised of a number of different classes. To see how these objects combine, consider the mapping of the Scratch Screen to user interface components as shown in Figure 7.2.

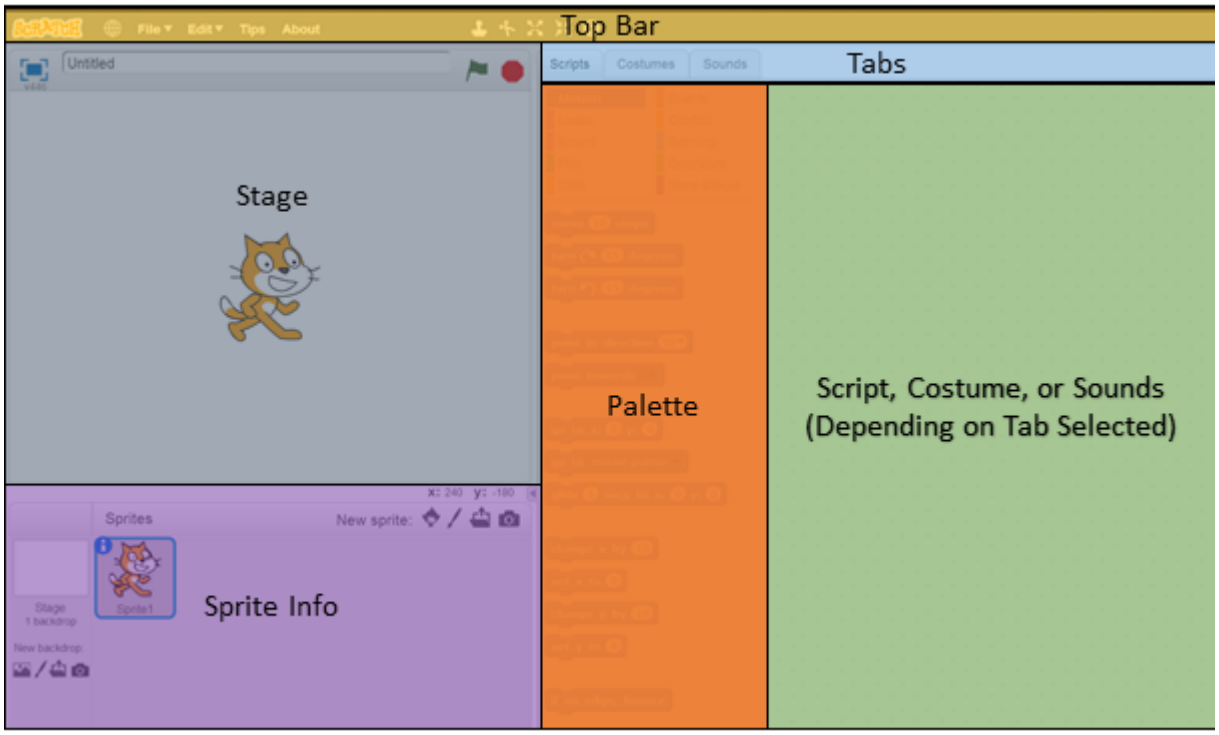

Figure 7.2: The Scratch programming environment user interface components.

Palette Classes

The Scratch palette is a subcomponent of the user interface that is composed of a number of additional classes. Overall, these classes determine which blocks should be displayed based on a number of factors including what type of Scratch Object is currently being viewed.

7.3 Scratch EV3 Extension

In addition to understanding the Scratch Flash source code, it is important to understand the ScratchX extension for EV3.

As with all ScratchX extensions, the EV3 extension is a JavaScript file that defines new Scratch Blocks and sets functions for the Interpreter to execute when the block is used. The EV3 ScratchX extension also defines a number of *data* elements that are EV3 specific.

When the user adds EV3 ScratchX extension to ScratchX, the blocks defined in the JavaScript appear in the "more blocks" palette. These are then usable in the same manner as any other Scratch block. The EV3 Scratch Extension includes thirteen blocks that enable users to program EV3 motors and the light/color, ultrasonic, and touch sensors.

Figure 7.3: A **Sprite** reporter block's shape is distinct, indicating it can only be used in specific locations.

7.4 Scratch with Components of Object Oriented Technology

To differentiate the language created by this project from others, we have called it Scratch with Components of Object Oriented Technology or SCOOT for short. Here, we discuss the additions and extensions made to ScratchX to transform it into SCOOT and to meet all the requirements outlined in Chapter 5.

7.4.1 Sprites as Attributes

The first update required for SCOOT was transforming **Sprites** into a passable attribute. As Scratch **Sprites** are already a class within Scratch, a reporter block was created that returned the appropriate Scratch **Sprite** object.

To maintain the user's ability to only create syntactically correct programs, a new shape was created for **Sprites**. This shape can be seen in Figure 7.3. SCOOT will only allow users to place **Sprites** into locations where a **Sprite** is expected.

Once this was accomplished, the already existing blocks that acted on **Sprites** were updated to require a **Sprite** as shown in Figure 7.4. This enabled the Interpreter to update the passed **Sprite** instead of the current *do-object*, giving the user greater control over his or her program's interpretation.

The **Sprite** reporter blocks were added to the palette under a new category "Creatures".

7.4.2 Encapsulation

There were two components for implementing encapsulation. To begin, public and private needed to be included for *data*, *lists*, and *procedures*. Additionally, an interface display was added to the programming environment. Here, the implementation of these features is described.

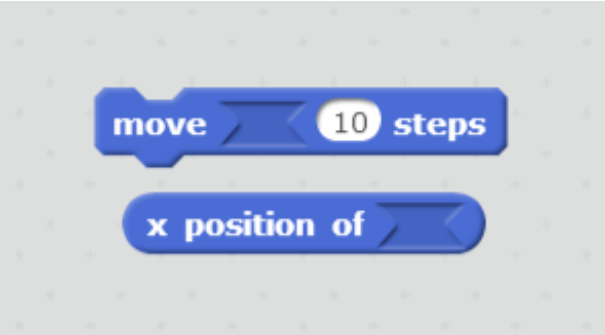

Figure 7.4: A reporter and *procedure* block that require a **Sprite** attribute to execute.

Public and Private

ScratchX already contained support for public and private *data* and *lists*. As a result, we only needed to implement public and private functions. To do this, we first consider how Scratch accomplishes public and private *lists* and *data*.

For *data* and *lists*, public items are stored in the Scratch **Stage** while private items are stored in the Scratch **Sprite**. This can be done because *data* and *lists* are stored within Scratch as objects. This same approach could not be used for *procedures* as there is no *procedure* object within Scratch.

Instead, we considered how Scratch Runtime looks up *procedures*. This is done by the Scratch Runtime iterating over all the Scratch Objects until the *procedure* definition is found. From examining the code, it became apparent *procedures* were only private because they could not be accessed via the palette.

As a result, to enable public and private *procedures*, the palette needs to display the appropriate blocks. This was accomplished by adding an isGlobal boolean to the Block class. When a *procedure* is defined, the dialog box that appears to set parameters includes a check box for making the *procedure* global as shown in Figure 7.5.

By knowing the isGlobal boolean, which **Sprite** defined the *procedure*, and the current *viewed-object*, the palette updates which *procedures* should be shown according to

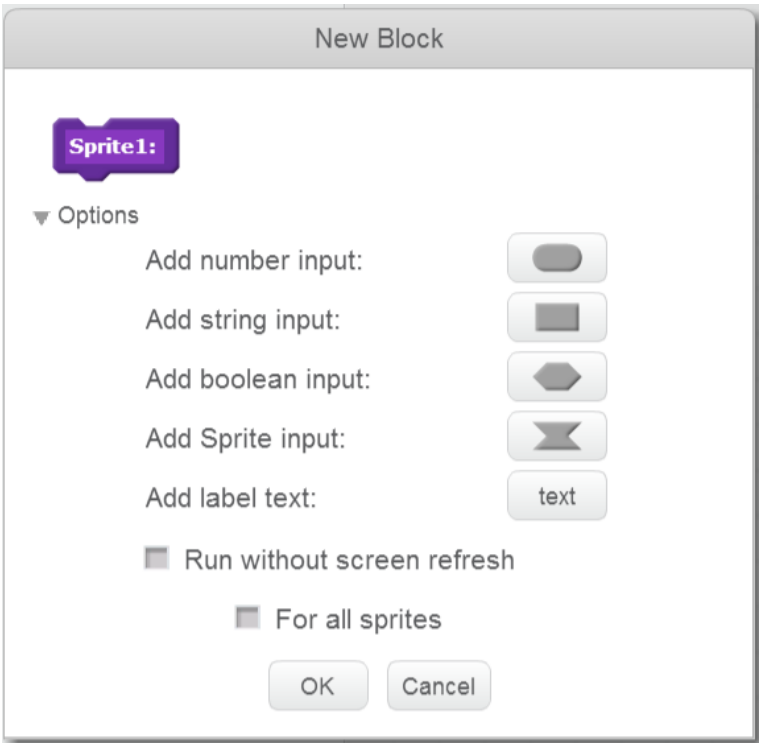

Figure 7.5: Dialog box for updating a *procedure's* parameters.

Algorithm 1.

Data: p is a procedure definition;
isGlobal is a boolean;
viewedObject is the currently view object set by the Scratch class
Result: p is added to the display if allowed by the isGlobal boolean
if *isGlobal* **then**
  Show p
else
  if *viewedObject defines p* **then**
    show p
  end
end

Algorithm 1: Algorithm for showing the appropriate *procedures* based on the isGlobal boolean.

Interface Display

The interface display shows the *procedures*, *data*, and *lists* that are publicly available from the current **Sprite** being viewed as shown in Figure 7.6. The interface display appears on the background of the scripting environment. This enables users to "see" the interface of the Scratch Object they are currently programming.

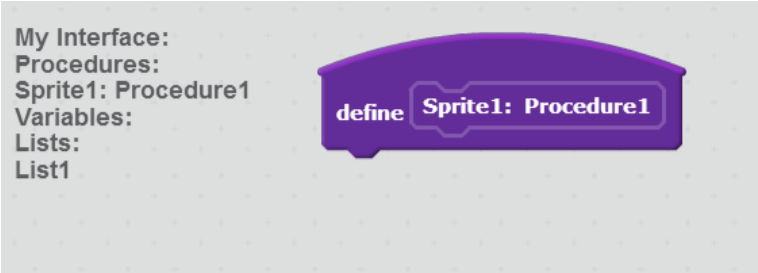

Figure 7.6: Interface display on the background of the SCOOT scripting environment.

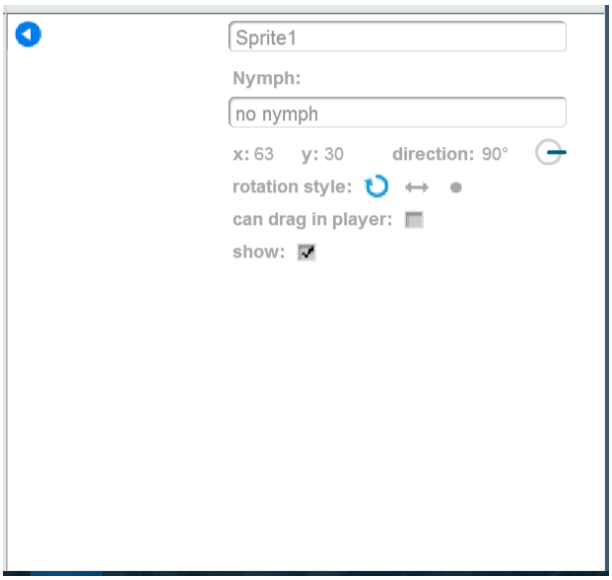

Figure 7.7: The **Sprite** info pane includes the ability to set a **Nymph**.

7.4.3 Nymphs

In mythology, nymphs are higher beings than sprites. By analogy, the parent class of a **Sprite** is a **Sprite**. The **Nymph** can be set in a **Sprite's** info pane as shown in Figure 7.7. Once a **Nymph** is set, the **Sprite** receives the **Nymph's** costumes and private attributes.

Costumes are stored within a **Sprite** as a list filled with Costume objects. For a **Sprite** to receive the **Nymph's** costumes, we added the **Nymph's** costumes to the **Sprite's** costume list. Because these reference the same costume object, any changes made to the costume will be reflected in both the **Nymph's** and **Sprite's** view of the costume.

For *data*, *lists* and *procedures*, changes were simply required to the palette. Here, SCOOT displays both the **Sprite** and its **Nymph's** attributes and methods.

7.4.4 Ghosts

It is a truth universally acknowledged that ghosts do not exist. Yet, the concept persists. With this in mind, the implementation of *abstract* classes in SCOOT was accomplished by introducing **Ghosts** to the language.

Ghosts are the most complex component in SCOOT. As there is no difference between classes and objects in Scratch, **Ghosts** must attempt to be an *abstract* class in a class-less language. As a result, we have focused on the ability to define an *abstract* class, but not create any objects from the class. In SCOOT, this is manifests as allowing **Ghosts** to exist, but not allowing them to execute without assistance from a **Sprite** or the **Stage**.

To accomplish this, **Ghosts** do not have any costumes and thus cannot appear as actors on the Scratch **Stage**. The hat blocks are also removed for **Ghosts** so their actions have no way of starting. This leaves **Ghosts** with the ability to declare *data*, *lists*, and *procedures*.

Ghosts are implemented within SCOOT as a Scratch Object. The palette does not show the hat blocks, aside from *procedure* definition hats, when a **Ghost** is the *viewed-object*. Additionally, the user interface objects work together to remove the costume tab.

It should be noted that **Ghosts** appear as reporters within the creatures section of the palette and can be set as a **Sprite's Nymph**.

7.4.5 EV3 Integration

To generate a product that feels well-integrated, the ScratchX EV3 extension was pre-loaded in the SCOOT environment. This allows users to see SCOOT as one product instead of the several underlying systems.

The ability to access the Gyro sensor's reading was added. This enables *the system* to meet the functional requirements.

7.5 Wide Walls

It should be noted that while the original functional specifications outlined the language's interaction with the EV3, SCOOT is able to work on a larger number of projects. This was caused by the decision to use Scratch as the underlying platform of SCOOT.

This is not a problem as the original specification has still been met. Indeed, it instead "widens the walls" of the language making it better suited for students as discussed in Section 2.1.2.

Chapter 8

Requirements Validation

To validate SCOOT, it is compared to the requirements set in Section 5. Here, each section of the requirements is considered to ensure nothing is forgotten.

8.1 Interfaces

8.1.1 System Interfaces

There were two major systems SCOOT needed to interface with—the EV3 brick and the computer running SCOOT. SCOOT is able to successfully integrate with and control the EV3 brick. On the computer side, SCOOT was required to work on computers running Windows XP, Windows 7 and Windows 8/8.1 as the operating system.

SCOOT runs in an Internet Explorer or Firefox web browser. Between these, SCOOT is able to work on computers running the required operating systems. This requirement has been found more narrow than needed as SCOOT is able to run on a range of operating systems larger than specified.

8.1.2 User Interfaces

SCOOT is without a doubt a visual programming language meeting requirement UI1. There is some reading and writing required for the language, but none above a 5th grade reading level.

Additionally, the names of new components of SCOOT—**Nymphs** and **Ghosts**—were chosen to reflect the student user's age.

8.1.3 Hardware Interfaces

The current *system* is able to interface with the EV3, the motors, and required sensors.

8.1.4 Software Interfaces

As SCOOT is able to run on the EV3, it is clearly able to properly interface with the brick's Linux kernel.

8.1.5 Communications Interfaces

SCOOT can communicate with the EV3 via Bluetooth. As only one communication protocol was required, this meets the requirements.

8.2 Non Functional Requirements

8.2.1 Reliability

Because SCOOT does not have firmware, the reliability requirement is unnecessary for *the system* developed.

8.2.2 Availability

The system's availability is Dependant on the web hosting. *The system* is currently hosted on GitHub which was available for 99.47% of the month of May 2016 [8]. *The system* therefore meets the availability requirement.

8.2.3 Maintainability

As a web hosted system, SCOOT can be easily updated at any point in time for all users.

8.3 Functional Requirements

All the functional requirements have been met by *the system*. Given their increased importance, here we examine data encapsulation, inheritance, and *abstract* classes.

8.3.1 Data Encapsulation

SCOOT implements data encapsulation. *Data*, *lists*, and *procedures* can all be made public or private. Additionally, an interface display was added to help students learn about *data* encapsulation.

8.3.2 Inheritance

Inheritance was implemented through **Nymphs**. Once a **Sprite** sets its **Nymph**, it receives the **Nymph's** costumes, *data*, *lists*, and *procedures*.

8.3.3 *Abstract* Classes

Abstract classes were implemented through **Ghosts**. **Ghosts** exist, but are incapable of doing anything on their own as they lack the "event" hats that start scripts. **Ghosts** also lack a physical appearance as costumes cannot be associated with a **Ghost**.

Chapter 9

Discussion

In the previous chapters we have presented the work completed as part of this project. Here, we use this work as a base to form new arguments in the field of teaching object oriented programming. We begin by discussing the ambiguity associated with SCOOT and present theories on why the survey conducted indicates that fourteen year olds find it easier to learn object oriented programming. Additionally, we introduce five potential threshold object oriented concepts and a new measurement system.

9.1 Object Oriented Concept Ambiguity in SCOOT

As discussed in Section 7.4, **Nymphs** are parents and **Ghosts** are *abstract* classes. However, this can be disputed. Because **Sprites** are objects and classes, there is ambiguity in these terms.

This ambiguity was designed into *the system*. SCOOT is not, and was never intended to be, object oriented. Instead, SCOOT should assist in the transition to object oriented programming. We argue these ambiguities will further assist students in the transition as they provide students with a base understanding of multiple object oriented concepts.

For example, **Ghosts** could be viewed as classes while **Sprites** are objects. This is caused as a **Sprite** receives all of the **Ghost's** attributes. However, it is important to note that **Sprites** can exist on their own, and are thereby still a mix of classes and objects.

9.2 Age to Learn Object Oriented Programming

Based on the information collected in the survey, it appears fourteen year olds find it easiest to learn object oriented programming. They found it easier than any other age group, including thirteen and fifteen year olds. It is unclear why the data suggests that fourteen year olds found learning object oriented programming easier than other age groups, especially in comparison with those one year older or younger. Here, we present some theories as to why this occurred.

To understand why it was easier for fourteen year olds to learn object oriented concepts compared with those one year younger, a few theories emerge. The first is that by fourteen, most students have begun algebra at school. The abstraction taught in algebra extends a student's abstraction abilities and thereby assists in the transition to object oriented programming. This theory is supported by the literature which states mathematical knowledge increases students' abstraction abilities as discussed in Section 2.5.

Another possible explanation arises from considering the participants recruited from the survey. As the recruitment focused on FIRST Robotics students and alumni, we know there is the potential to have skewed data. FIRST Robotics students normally transition to object oriented programming when they begin in the FIRST Robotics Competition at 14 years old. The data implies this is correct with 20% of survey participants learning object oriented programming at 14 years old. Perhaps the ease of learning object oriented programming comes from not doing it alone, but partaking in a group learning the language together.

9.3 Object Oriented Threshold Concepts

Object Oriented programming is recognized as a threshold concept within computing as discussed in Section 2.1.5. As a result, it can be surmised that there are object oriented threshold concepts. Here, we consider the survey results to try to find potential threshold concepts. We begin by remember the five attributes of a threshold concept as presented in Section 2.1.5: irreversible, troublesome, boundary marker, transformative, and integrative.

Because we did not ask the survey participants whether they unlearnt any concepts, we cannot comment on the irreversibility of potential object oriented threshold concepts. As a result, we cannot definitively declare any threshold concepts. Instead, we examine potential concepts in terms of the other four threshold concept attributes.

We next consider the "potentially troublesome to learn" attribute of a threshold concept. In particular, we remember the six concepts that appeared through the survey as both the *easiest* and most *difficult* concept: classes, encapsulation, inheritance, objects, polymorphism, and procedural concepts. Obviously, procedural concepts are not an object oriented threshold concept as they are not unique to object oriented programming. As a result, we will further consider classes, objects, encapsulation, inheritance, and polymorphism with the remaining threshold concept attributes.

A boundary concept should represent the where the subject begins or ends. Objects and classes are the underlying concept for object oriented programming, as a result they clearly mark the start of object oriented programming. In Section 2.3, encapsulation, inheritance, and polymorphism were all shown to be crucial to object oriented programming. As a result, they each mark a boundary of object orientation.

A transformative concept should change a person's perspective. To learn object oriented programming, a shift is required from procedural to object oriented thinking. This implies the very transition is transformative. As the beginning of object oriented program-

ming, objects and classes are clearly instrumental in this transition. As core concepts of object oriented programming, encapsulation, inheritance, and polymorphism should also change the student's perspective.

A threshold concept must integrate other previously unknown concepts. It is easy to see how objects and classes integrate each other. A class would not be useful if one could not instantiate an object. At the same point in time, an object could not exist without a class. As a result these two concepts are integrative with each other. Encapsulation includes many other object oriented concepts including data hiding and modularity. As we have argued several times before, inheritance and polymorphism are integrated concepts. Polymorphism is an extension of the "is-a" inheritance relationship. As a result, both inheritance and polymorphism are integrative.

Here, we have shown that objects, classes, encapsulation, inheritance and polymorphism all meet four of the five criteria of a threshold concept. The fifth concept, irreversibility, cannot be confirmed with the currently available information. As a result, we can say objects, classes, encapsulation, inheritance, and polymorphism are all potential object oriented threshold concepts.

9.4 Measurements

We know Scratch is object-based, yet students still report struggling in the transition from Scratch to an object oriented language like Java or C++. This raises the question of how many object oriented concepts does a programming language need to assist the transition from procedural to object oriented programming. The answer to this question is crucial to the overall project and *system* as it will dictate what features *the system* implements.

9.4.1 Measurement Definition

To better understand this, we need an objective measurement system that will allow programming languages to be ranked based on the number of difficult object oriented concepts the language implements. No such measurement system currently exists. As a result, we began by creating a measure of the difficulty of learning the object oriented concepts in a given programming language. We will call this a measure of Object Oriented Learning Difficulty (*OOLD*).

Based on the above definition, we know an object oriented programming language should have the maximum possible *OOLD*. With this in mind, we have chosen to measure *OOLD* as a percentage. Subsequently, we can say a programming language is a certain percentage *OOLD*. In addition, we expect an object oriented language to be 100% *OOLD* by definition.

Table 9.1: Mapping reported object oriented concept difficulty to percentage of overall difficulty.

Concept	**Average Difficulty**	**Percentage of Difficulty**
Accessing Variables	2.09	8.32%
Method Calls	2.38	9.46%
Public v. Private	2.41	9.60%
File Interaction	3.13	12.45%
Inheritance	3.41	13.58%
Encapsulation	3.80	15.15%
Abstract	3.89	15.50%
Polymorphism	4.00	15.93%
Total	*25.11*	*100%*

9.4.2 Measurement Creation

To measure *OOLD*, we need to consider the difficulty of a number of components of object oriented programming. With the results from the survey we have a number of attributes of object oriented programming scored on difficulty to learn. These values are mapped to percentages of the overall difficulty as shown in Table 9.1.

Each language is given a rating for each attribute with 0 meaning the language does not support the attribute, 0.5 meaning the language partially supports the attribute, or 1 meaning the the language fully supports the attribute. While this is a coarse approximation and does not fully reflect the spectrum of supporting an object oriented concept, the error introduced is minimal. We know the maximum error for a concept is 0.25. Given the hardest concept (polymorphism) provides approximately 16% of the difficulty, the maximum error introduced in a concept is 4%. We deem this an acceptable error.

With these mappings, we can use the following equation to find the *OOLD* of a language:

$$OOLD = \sum_{concepts} d\% \times r \tag{9.1}$$

Where d is the percentage of difficulty and r is the rating of 0, 0.5, or 1 assigned in the method described above.

9.4.3 Measurement Usage

To use the new measurement system, consider this example of rating Scratch's and Java's *OOLD*. This analysis is shown in Table 9.2. Each language is given a score of 0, 0.5, or 1 as outlined above. This is then multiplied by the percentage difficulty. These category difficulties are added together to determine the language's overall *OOLD*.

Table 9.2: An example of using the measurement system to evaluate Scratch and Java.

	Percentage of Difficulty	**Scratch**	**Java**
Accessing Variables	8.32%	1	1
Method Calls	9.46%	1	1
Public v. Private	9.60%	0.5	1
File Interaction	12.45%	1	1
Inheritance	13.58%	0	1
Encapsulation	15.15%	0.5	1
Abstract	15.50%	0	1
Polymorphism	15.93%	0	1
OOLD		*43%*	100%

Based on the information shown in Table 9.2, we can see Scratch was 43% *OOLD* while Java was 100% *OOLD*. This suggests that Scratch implements less than 50% of the difficulty associated with learning object oriented programming. Java is 100% *OOLD*. This is not surprising as Java is an object oriented language, and therefore should be 100% *OOLD* by definition.

9.4.4 Measurement Shortcomings

The major shortcoming of this measurement is the lack of components that contribute to *OOLD*. Only the attributes that were scored on the survey can be utilized as they have an associated quantifiable metric. This excludes a number of difficult concepts discovered in the course of the survey.

This exclusion occurs because the new attributes found were not scored by participants. As a result, there is no quantitative data on which attributes are harder than others. The number of participants who provided a concept as the most difficult to learn is independent from the average reported difficulty of the same concept.

This is because it is impossible to compare the hardest question data to the average ranking data. Consider inheritance which 12 people cited as the most difficult concept, yet it only received an average difficulty score of 3.41. In contrast, polymorphism was the hardest part of object oriented programming for 10 participants, yet it received a difficulty score of 4.00. As a result, it is apparent why concepts that should be included in the *OOLD* measure cannot currently be used due to a lack of data.

This is particularly frustrating for classes and objects. Theses two concepts, and the difference between them, were provided 13 times as the most difficult concept. Combined, this is more than any other area. Despite this, "class versus object" cannot be included in the measure as we lack the average response.

This shortcoming is important to note as it implies the measurement is not completely

Table 9.3: *OOLD* of Scratch, SCOOT, and Java.

	Percentage of Difficulty	**Scratch**	**SCOOT**	Java
Accessing Variables	8.32%	1	1	1
Method Calls	9.46%	1	1	1
Public v. Private	9.60%	0.5	1	1
File Interaction	12.45%	1	1	1
Inheritance	13.58%	0	1	1
Encapsulation	15.15%	0.5	1	1
Abstract	15.50%	0	0.5	1
Polymorphism	15.93%	0	0	1
OOLD		*43%*	76%	100%

accurate. However, the measure does provide a baseline quantifiable metric for evaluating programming languages. This shortcoming, and how to overcome it, are discussed in more detail in Section 11.3.

9.4.5 Learning Difficulty of SCOOT

Based on the analysis done in Section 9.4.3, we know Scratch is 43% *OOLD* while object oriented programming languages are 100% *OOLD*. As a result, SCOOT should be approximately 70% *OOLD* to be halfway between Scratch and object oriented languages.

SCOOT is currently 76% *OOLD*. In Table 9.3, the *OOLD* for Scratch, SCOOT, and Java are shown.

Chapter 10

Conclusions

The goal of this project was to create a programming language to assist in the transition to object oriented programming. Along the way we realized there was no way to determine how difficult an object based language was to learn. To satisfy this, we created a new measure to quantify the expected difficulty to learn a language based on the object oriented concepts the language implements. Additionally, in the process of determining what object oriented concepts students struggle to learn, five potential threshold object oriented concepts were discovered. Here, we conclude by examining everything the project has accomplished.

A literature review was conducted to understand the background areas of the project. The review incorporated subject areas ranging from teaching pedagogy to visual language design. It was found that while it is widely recognized that the transition to object oriented programming is difficult, there is no research into why this is a difficult conversion.

With little research to base *the system* on, it was clear additional information would need to be gathered. Through the survey, we gathered information relating to five areas of inquiry. These areas were: optimal age to learn object oriented programming, easy to learn object oriented concepts, difficult to learn object oriented concepts, factors contributing to difficulty of object oriented programming, and difficulty of pre-determined object oriented concepts. In total, the survey had over 140 responses.

These responses were analyzed and two interesting results appeared. In particular, it was discovered that many concepts were easy for some while difficult for others. The responses suggest that the optimal age to learn programming is fourteen. Additional data collected led to the creation of a new measurement system and potential threshold concepts as discussed in Chapter 9

Based on the literature review and survey, a set of requirements were created for *the system*. These requirements defined *the system's* interface, non-functional attributes, and functionality. In total, they presented a unified view on how *the system* should work.

With the requirements set, *the system* could be designed. It was decided to build the language on Scratch, a common visual programming language. The Scratch graph grammar was extended and new user interface buttons and dialog boxes were designed. In total, these changes transformed Scratch into a *system* that met the requirements and

goal of the project.

Once a design was created, implementation could begin on *the system* which was called SCOOT. SCOOT extended Scratch to include encapsulation, inheritance, and *abstract* classes. Inheritance was implemented through the introduction of **Nymphs** while *abstract* classes appear as **Ghosts**. In total, this allowed SCOOT to meet the requirements.

In the process of creating SCOOT, a number of discoveries were made as noted in Chapter 9. Notably, five potential threshold concepts were found—classes, objects, encapsulation, inheritance and polymorphism. Additional research should be conducted to confirm these attributes as threshold concepts within object oriented programming.

To determine how difficult a language is to learn in terms of object oriented concepts, a new measurement system, Object Oriented Learning Difficulty (*OOLD*), was created. This measure was described in Chapter 9. The measure is based on information from the survey and allows language developers to determine how difficult an object based language is to learn.

Chapter 11

Future Work

11.1 Further Survey

Based on the information found in the initial survey, several new lines of inquiry have emerged. Here, we discuss these lines of inquiry and present questions to be included in a second survey.

To refine the new measure defined above, more object oriented concepts should be scored on difficulty by participants. The lack of having quantifiable data for some difficult concepts discovered in the open-ended question has limited the development of the measure. For example, class versus object was a component that appeared several times as a problem in the open ended question, but participants were not asked to score this component. As a result, it was excluded from the measure. By increasing the number of attributes considered in *OOLD*, the measure can become more accurate.

It is believed that at least five threshold object oriented concepts were discovered in the survey. However, because respondents were not asked about the the ability to "unlearn" these concepts, it is impossible to ascertain if they are threshold concepts. As a result, we propose adding a question to a second survey after asking what object oriented concept was the most difficult: "Once you learned this concept, have you un-learnt it?" In addition, when asking participants to score object oriented concepts for difficulty, the question should be expanded to include whether the concept has ever been un-learnt.

11.2 Trial of System

In this project, *the system* has been defined and implemented. This has been based purely on theory and some of the author's experiences and presents the beginning of research into what makes learning object oriented programming difficult. Given the immaturity of this field of study, it is no surprise that there is very little beyond theory in the field.

A clear next step is to use *the system* created within this project to help students transition to object oriented programming. A full study should be conducted to compare students learning SCOOT to those learning a lower *OOLD* language such as Scratch on

their ability to transition to object oriented programming. This will serve as a way to test both SCOOT and the *OOLD* measurement system.

11.3 Refinement of Measurement

As discussed in the above sections, the *OOLD* measure needs further testing and refinement. This should occur through multiple avenues including the survey and trial. It is expected the measure will need to be refined as further research in this area occurs.

11.4 Final Words

As shown in Chapter 1, Australia and the world are in need of more qualified engineers and information technology employees. As a result, we must strive to assist students into these fields. This project aims to do this be easing the transition from visual procedural to object oriented programming.

Chapter 12

Abbreviations

CS	Computer Science
EV3	LEGO MINDSTORMS EV3
FIRST	For Inspiration and Recognition of Science and Technology
FLL	FIRST LEGO League
FLL Jr.	FIRST LEGO League Junior
FRC	FIRST Robotics Competition
FTC	FIRST Tech Challenge
ICT	Information and Communication Technology
OO	Object Oriented
OOLD	Object Oriented Learning Difficulty
OOP	Object Oriented Programming
SCOOT	Scratch with Components of Object Oriented Technology

Appendix A

Requirements

The requirements for *the system* are shown in Tables A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, and A.9.

Table A.1: System Interface requirements for *the system*

Ref Number	Requirement	Rationale
SI1	*The system* must interface with EV3.	*The system* must properly operate on the LEGO MINDSTORMS EV3.
SI2	The user must be able to use *the system* on a Windows XP, Windows 7 or Windows 8.8.1 machine.	These three operating systems hold 85.45% market share [20].

Table A.2: User Interface requirements for *the system*

Ref Number	Requirement	Rationale
UI1	*The system* must have a graphic programming interface.	To enable primary school students to use *the system*, a graphics-based approach is used. This enables students to focus on mastering programming skills instead of their still-developing spelling and language skills.
UI2	*The system* must require minimal reading/writing for the student user.	To enable primary school students to use *the system*, a graphics-based approach is used. This enables students to not worry about their still-developing spelling and language skills.

Table A.3: Hardware Interface requirements for *the system*

Ref Number	Requirement	Rationale
HI1	*The system* must interface with the EV3 servo motor, including the ability to move the motor at a particular speed and take readings from the built-in rotation sensor.	The EV3 servo motor is part of the LEGO Retail and LEGO Education EV3 kits [11] [10].
HI2	*The system* must interface with the EV3 mini-motor, including the ability to move the motor at a particular speed and take readings from the built-in rotation sensor.	The EV3 mini-motor is part of the LEGO Retail and LEGO Education EV3 kits [11] [10].
HI3	*The system* must interface with the EV3 ultrasonic sensor to read its output.	The EV3 ultrasonic sensor is part of the LEGO Retail and LEGO Education EV3 kits [11] [10].
HI4	*The system* must interface with the EV3 touch sensor to read its output.	The EV3 touch sensor is part of the LEGO Retail and LEGO Education EV3 kits [11] [10].
HI5	*The system* must interface with the EV3 color sensor to read its output.	The EV3 color sensor is part of the LEGO Retail and LEGO Education EV3 kits [11] [10].
HI6	*The system* must interface with the EV3 ultrasonic sensor to read its output.	The EV3 ultrasonic sensor is part of the LEGO Retail and LEGO Education EV3 kits [11] [10].
HI7	*The system* must interface with the EV3 gyro sensor to read its output.	The EV3 gyro sensor is part of the LEGO Education EV3 kit [11] [10].

Table A.4: Software Interface requirements for *the system*

Ref Number	Requirement	Rationale
SW1	*The system* must interface with the EV3 Linux kernel.	To properly run, *the system* must interface with the operating system on the EV3.

Table A.5: Communication Interface requirements for *the system*

Ref Number	Requirement	Rationale
CI1	*The system* shall support communication via Bluetooth or a USB to Micro USB cable.	Bluetooth and USB are the two major communication protocols the EV3 supports.

Table A.6: Reliability requirements for *the system*

Ref Number	Requirement	Rationale
R1	*The system*'s firmware shall have a MTBF of 3 weeks with extensive use.	The MTBF of most languages is 2-4 weeks.

Table A.7: Availability requirements for *the system*

Ref Number	Requirement	Rationale
A1	*The system* shall be available 99% of the time.	With a high MTBF, *the system* should be available nearly constantly.

Table A.8: Maintainability requirements for *the system*

Ref Number	Requirement	Rationale
M1	*The system* shall be able to be updated once released.	As with other LEGO MINDSTORMS languages, *the system* should be able to be updated after released.

Table A.9: Functional requirements for *the system*

Ref Number	Requirement	Rationale
F1	*The system* must be object-based.	The whole point of *the system* is to enable students to learn object oriented programming, thus *the system* must have many of these traits.
F1.1	*The system* must allow different files to interact as the program executes.	In object oriented programming, different files must work together in the program execution.
F2	*The system* must allow users to create objects.	Objects are the key concept of object oriented programming, thus they must be supported.
F2.1	Objects must be able to contain attributes.	Without the ability to store data within an object, it is meaningless.
F2.2	Objects must be able to contain functions.	Without the ability for objects to do things, they would be meaningless.
F3	*The system* must support data encapsulation.	Data encapsulation is a key concept of object oriented programming and thus needs to be included.
F3.1	Attributes and functions must be able to be public or private.	Public versus Private is a key component of data encapsulation.
F3.2	*The system* must show an object's interface.	Encapsulation is a concept students struggle with, by showing the interface *the system* assists with this concept.
F3.3	Private attributes and functions can be implemented as protected.	To assist with a student's understanding of inheritance.
F4	*The system* must support inheritance.	Inheritance is a key concept of object oriented programming, as a result this must be included.
F5	*The system* must support assignment, variable declaration, sequence, test, and *loop*.	Most programming languages are comprised of at least five constructs; assignment, variable declaration, sequence, test, and *loop*. These constructs form the language's imperative core [32].
F6	*The system* must support *abstract* classes.	*Abstract* classes was a commonly reported difficult concept for students.

Appendix B

Survey Information

B.1 Survey

Pictures to Objects

Pictures to Objects Understanding how students transition from graphic to object oriented programming

Participation in this survey is anonymous and purely voluntary. Information collected will be used in an undergraduate research thesis at Macquarie University and any future publications of this project.

Participants of this survey must:

- **Be at least 16 years of age.**
- **If under 18 years of age, have parental permission to complete this survey.**
- **Reside in Australia, Canada, or the United States of America.**
- **Have begun programming with a graphic (visual, icon-based) programming language.**
- **Have learned an object oriented programming language.**

By ticking this box you agree you meet the participation requirements and understand the terms outlined above.

I agree

How old were you when you began programming in a graphic programming language?

What was your first graphic programming language?

LEGO Mindstorms (RCX)

LEGO NXT-G (NXT)

RoboLab (RCX or NXT)

LEGO EV3

Scratch

Visual Basic

Other:

What was the first style of text-based language you learned?

Object Oriented

Procedural

Functional

Other:

How old were you when you began programming in an object-oriented language?

What was your first object oriented programming language?

Java

C++

Python

Processing

Other:

With what level of ease did you transition to these programming methodologies?

	Very Easily	Easily	A little Easily	A little difficultly	Difficultly	Very Difficultly	Not Applicable
Object Oriented	○	○	○	○	○	○	○
Functional	○	○	○	○	○	○	○

What object oriented concept did you find the easiest to understand?

What made this concept easier to understand?

What object oriented concept did you find the most difficult to understand?

What made this concept difficult to understand?

With what level of ease did you understand the following Object Oriented Concepts?

	Very Easily	Easily	A little Easily	A little difficultly	Difficultly	Very Difficultly	I'm not sure what that means
How the different files interacted.	○	○	○	○	○	○	○
Method Calls.	○	○	○	○	○	○	○
Accessing Variables.	○	○	○	○	○	○	○
Public vs. Private.	○	○	○	○	○	○	○
Inheritance.	○	○	○	○	○	○	○
Abstract Methods/Variables/Classes	○	○	○	○	○	○	○
Polymorphism	○	○	○	○	○	○	○
Encapsulation	○	○	○	○	○	○	○

To receive a copy of the study's results once complete, please enter your email address below.

B.2 Survey Recruitment

A copy of the survey recruitment flyer is shown in Figure B.1

Figure B.1: Survey Recruitment Brochure

Appendix C

Survey Responses

C.1 Easy Object Oriented Concepts

In total, 58 responses were received to the short answer question on the easiest object oriented concept. All these responses, and their categorization, are provided in Table C.1 and Table C.2.

Table C.1: First half of responses, and their categorization, to a short answer question on the easiest object oriented concept.

Raw Response	**Category**
Class	Classes
Class	Classes
Classes	Classes
Classes	Classes
Classes	Classes
Classes	Classes
Classes	Classes
Classes are like containers	Classes
Classes as being "blueprints" for new objects to be made	Classes
Classes as reusable pieces of code	Classes
Multiple classes	Classes
The idea that classes contained methods that I've been using before I learned object oriented programming	Classes
encapsulation	Encapsulation
Encapsulation	Encapsulation
Encapsulation	Encapsulation
Encapsulation	Encapsulation
Modular programming	Encapsulation
Modularity	Encapsulation
Classes and subclasses	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
Subclasses and interfaces	Inheritance
nill	N/A
not sure	N/A

Table C.2: Second half of responses, and their categorization, to a short answer question on the easiest object oriented concept.

Raw Response	**Category**
Dividing components into objects	Objects
General idea of what an object is	Objects
Objects	Objects
Objects	Objects
Objects	Objects
Objects modeling real world items/behaviors	Objects
The idea that objects can have properties	Objects
What to put into objects	Objects
interacting with other objects	Objects
Classification	Other
Composition	Other
Recursion	Other
rotations	Other
overloading/polymorphism	Polymorphism
Polymorphism	Polymorphism
Polymorphism	Polymorphism
polymorphism	Polymorphism
conditionals	Procedural Concepts
Data Types	Procedural Concepts
Flow? (Loops, function calls etc.)	Procedural Concepts
Functions	Procedural Concepts
functions	Procedural Concepts
Functions/tasks/variables	Procedural Concepts
If then	Procedural Concepts
Logic Statements	Procedural Concepts
Loops	Procedural Concepts
Loops, switches, cases etc	Procedural Concepts
Methods	Procedural Concepts
Method operations	Procedural Concepts
syntax is easy	Syntax
The syntax of object.method()	Syntax

C.2 Difficult Object Oriented Concepts

In total, 57 responses were received to the short answer question on the most difficult object oriented concept. All these responses, and their categorization, are provided in Table C.3 and Table C.4.

Table C.3: First half of responses, and their categorization, to a short answer question on the hardest object oriented concept.

Raw Response	**Category**
Abstraction	Abstract
Abstraction/ interfaces	Abstract
Diference between interface and class	Class versus Object
Difference between class, and an instantiated object	Class versus Object
instances of objects vs class definitions, and the differences with static objects.	Class versus Object
Objects and classes themselves	Class versus Object
[User-Defined] Classes (and why they are useful)	Classes
Class definitions	Classes
classes	Classes
Classes	Classes
Classes	Classes
Classes/objects	Classes
Relating Classes (Extends, Implements)	Classes
Data structures	Other
Confining functions to a single object	Encapsulation
Interfaces	Encapsulation
Interfaces in Java	Encapsulation
objects don't talk to each other esily	Encapsulation
Setters/Getters	Encapsulation
extends and implements relationship	Inheritance
Inheritance	Inheritance
inheritance	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
Inheritance	Inheritance
inheritance	Inheritance
Inheritance and delegation	Inheritance
inheritence	Inheritance
Polyinheritance	Inheritance

Table C.4: Second half of responses, and their categorization, to a short answer question on the hardest object oriented concept.

construction/deletion	Objects
Objects	Objects
How similar things differ (accessibility of things, functions vs methods, enums, structs, etc)	Other
Singletons	Other
Small nitpick stuff with brackets etc	Other
static methods	Other
v	Other
Multi-faceted functions (doing different things based on certain inputs, etc)	Overloading
Overloading	Overloading
Inheritance and polymorphism	Polymorphism
Inheritance/polymorphism	Polymorphism
Polymorphism	Polymorphism
Polymorphism	Polymorphism
Polymorphism	Polymorphism
Polymorphism	Polymorphism
Polymorphism	Polymorphism
Polymorphism	Polymorphism
Polymorphism	Polymorphism
Probably Polymorphism	Polymorphism
Methods	Procedural Concepts
methods	Procedural Concepts
Recursion	Procedural Concepts
Recursion	Procedural Concepts
Some aspects of methods	Procedural Concepts
Variables	Procedural Concepts
writing codes and referencing variables	Procedural Concepts

C.3 Reasons for Easy Object Oriented Concepts

The reasons participants provided for finding an object oriented concept *easy* to learn are outlined in Table C.5 and Table C.6.

Table C.5: First half of survey responses for why an object oriented concept was easy to learn.

A book about Flash programming which explained different classes of Cars, all inheriting the base methods of a Car such as driving but each having their own unique functions
A general understanding of templates and inheritance of attributes.
Because I enjoy keeping things organized
Detailed class diagrams
Division of code into sub-VIs
Flow Charts
Functions are similar to SubVIs in LabVIEW
Graphic programming experience
having a base
Having a physical manifestation of the '{}objects'{} that I was programming; in this case, a robot.
How its defined
I always thought of it as giving the object a "hat" so it could do other things.. I like organizing things so it this concept just made a lot of sense.
I could picture what an object was
i had used them before in scratch
I was able to relate this to a physical understanding of a robot. It made sense that each part of the robot would get its own class and set of methods.
Imagining them as a box of parts and values
Intuitive concept
It is intuitive
it is simple.
It makes sense, to have different ways to call the same method
It was fairly ease to visualize as just a function with a weird implementation.
It was intuitive to me, but Java tutorials made it easy.
it was really just like scripting languages, but with custom variable types.
It's a logical extension of OOP features.
It's fairly intuitive
It's often the starting point (and even the justification for) using OOP
Just the explanation of inheritance on where I learned JS, Codecademy.

Table C.6: Second half of survey responses for why an object oriented concept was easy to learn.

knew C
logic
Looking at things sequentially
looking inside classes to see the methods contained within
math
mentor
Metaphors, like Animal ->Pet ->Dog
Mindstorms is pretty much event driven programming, where independent tasks cause others to run. Somewhat similar to using other objects in concert.
Most programming languages already support this, e.g. "+" works with integer, float, string/char, etc.
nill
not sure
People explaining it in videos
Physical analogies
Practical application
Practicing
Python's inherant everything-is-an-object-but-you-don't-know-it-until-you-need-it
Reading Design Patterns book
Real Wolrd Examples
Scratch had sprites which separated code between things kind of like objects
Setting an object's property and having it immediately reflected in the GameMaker environment
Similar to reusing functions, but including data members as well
Simplification of code
The analogy with real-world objects (pardon the pun), such as a blueprint and a building made this very clear
The drag and drop nxt blocks
The fact that it is built in to standard objects like strings (Python)
The idea of data structures was taught in the previous language, and adding methods to that was pretty intuitive.
The usefulness of having classes already written from previous projects that could be recycled.
The way my teacher beat the importance of them into our heads
Visual programs do a good job of showing a switch is like a desicion fork and a loop repeats. So the visual understanding made it transferable
Visualize how objects interact

C.4 Reasons for Difficult Object Oriented Concepts

The reasons participants provided for finding an object oriented concept *difficult* to learn are outlined in Table C.7 and Table C.8.

Table C.7: First half of survey responses for why an object oriented concept was difficult to learn.

a child can be substituted for a parent
Although the concept is intuitive, implementation details have been consistently confusing
Bad instruction
building classes
coding a single class, which becomes multiple objects holding different variables.
Coming from procedural, it was just a foreign concept on why it was needed (until I saw examples of how it makes things nicer)
Confusing
confusion between classes and objects
Couldn't see how the objects interacted as much
Difficult to visualize
Getting used to what could and should be included in a class
Having done functional/procedural for a long time, I didn't immediately see the need for classes, and the value of them.
How they were hidden in nxt-g menus
How they're used
I confused inhritance with being a member
I could not find an easy way to relate this to what I had previously learned in LEGO NXT. This was a completely new concept for me and I had nothing to compare it to.
I still don't understand it!
I was learning by trying stuff, and an "interface" wasn't at all what I thought it was.
I was used to dividing commands into functions by when I wanted to do them (procedural) , not by what they did (OOP).
I'm not too sure of a great way to use which in a given scenario
In hindsight it was not awful. I guess I didn't like the syntax?
Inability to find coherent set of rules for polymorphism
Inheritance made sense - taking one class's traits and adding onto it - but understanding the reverse flow of the inheritance tree and figuring out that variables of type A can be included with type B where B extends A didn't come easily until thoroughly explained.
Interfaces were hard
Is a, has a, umm... which is which?
It is very complex
it required multiple scripts
It was different than the way I had thought about things in the past
It was hard to understand why you could have a variable with a certain type and assign it with another type. If you don't quite understand I wrote a quick example, http://ideone.com/dPCBAW
It was not explained well to me
It's just not explained well. Graphical languages did not set me back. I heard examples such as "fruit is class, banana is object" but it's actually "banana is class, the one you're about to eat is object".
Java doesn't seem to like them
Just got frustrated at first but eventually understood after practice

Table C.8: First half of survey responses for why an object oriented concept was difficult to learn.

LabVIEW did not have support for polymorphism, so I did not have any background in the subject.
Lack of clarity in any documentation.
Lack of exposure to such structures before
minor inconsistancies
No analogy to anything like this in a graphical or functional based language. Very powerful once understood, however.
No graphical corollary
not sure
Partly, the name was confusing. “Inheritance” does not naturally seem like one class is a subset (Zebra inside of animal, for example).
persistance of objects
Poor explanation
Something about this was always tricky for me because I would always miss something and have to go back. Over time they got easier but definitively were the hardest to master.
still not easy
syntax, passing parameters
the amount of variables
The use scenarios where abstractions are needed, and how to decide what kind of abstraction one would need to use.
The ways in which methods can call themselves and change at the same time
There really isn’t a class structure in LabVIEW
Too much to memorize
Very different from more linear coding styles
Was sort of left on my own and didn’t exactly know, at the time, how it would be relevant to anything outside of Codecademy
What made this concept difficult to understand?
When to use them
Which superclass is providing the chosen method?

Appendix D

Scratch Graph Grammar

The basic graph grammar for Scratch is shown in Figure D.1, Figure D.2, Figure D.3, Figure D.4, Figure D.5, Figure D.6, Figure D.7, Figure D.8, Figure D.9, Figure D.10, and Figure D.11.

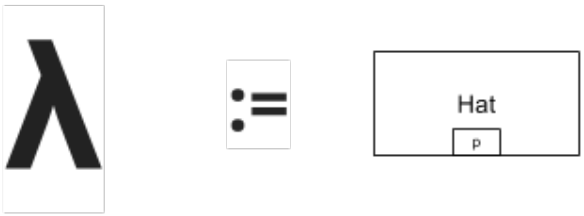

Figure D.1: Graph grammar for beginning a script.

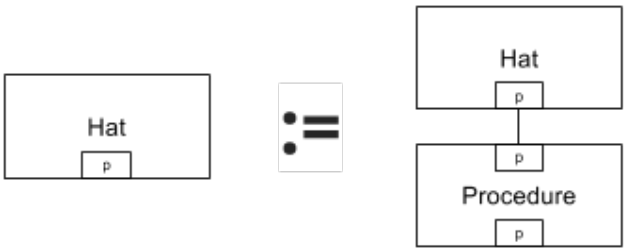

Figure D.2: Graph grammar for a hat block.

Figure D.3: Graph grammar for a script beginning with an event.

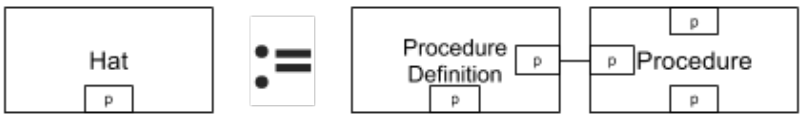

Figure D.4: Graph grammar for a *procedure* definition.

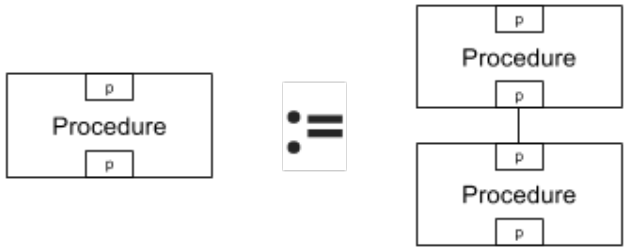

Figure D.5: Graph grammar for *procedures*.

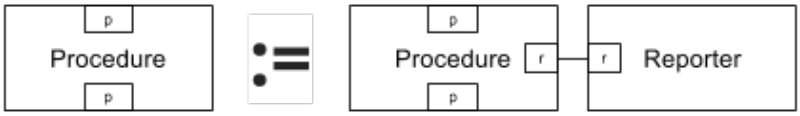

Figure D.6: Graph grammar for *procedures* calling reporters.

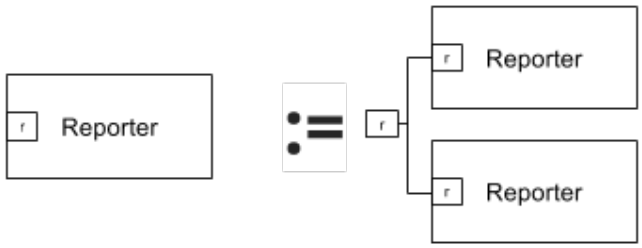

Figure D.7: Graph grammar for having multiple reporters.

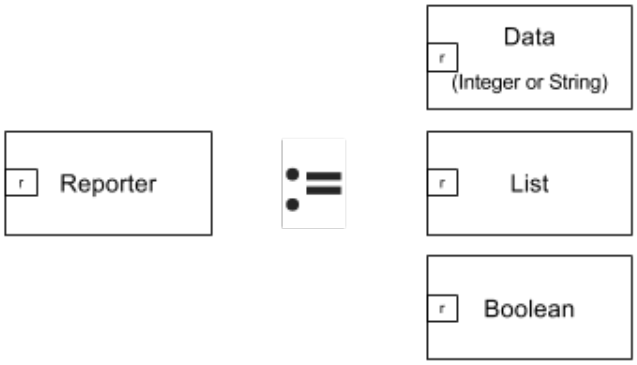

Figure D.8: Graph grammar for different types of reporters.

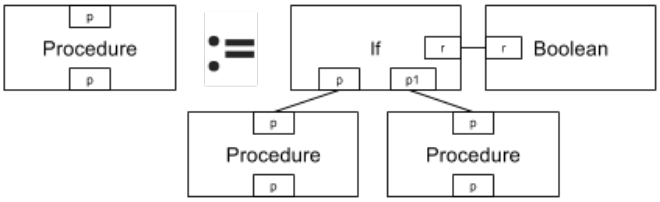

Figure D.9: Graph grammar for an *if* statement.

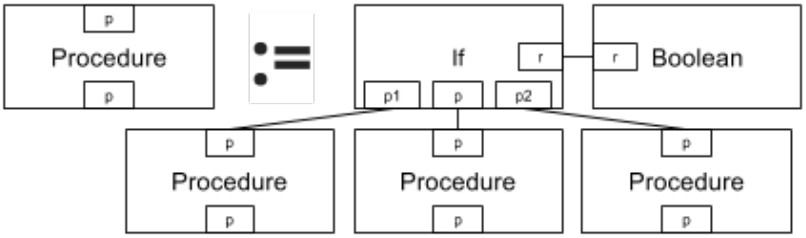

Figure D.10: Graph grammar for an *if-else* statement.

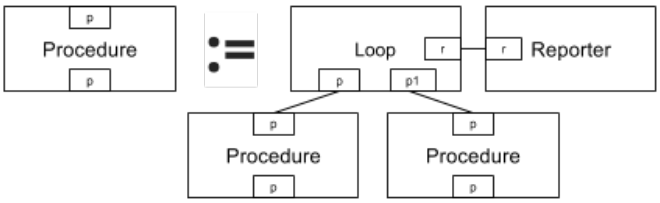

Figure D.11: Graph grammar for a *loop*.

Appendix E

Meeting Attendance Form

Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
—	12-2	In person		
-1	25-2	E-mail		
2	8/3	E-mail		
3	14-3	In person		
4	22-3	E-mail		
5	31/3	E-mail		
6	6-4	E-mail		
mid sem break	13-4	E-mail		
mid sem break	20-4	E-mail		
8	4-5	In person		
9	9-5	In person		
10	15-5	E-mail		
12	2-6	In person		

Appendix F

Code

This chapter contains excerpts from the modified Scratch Flash source code. Code with a green background indicates a new or modified line. The full Scratch Flash source code and this branch are available on GitHub.

It is important to note that the Scratch Flash source code has very few comments. This coding style was adhered to throughout *the system's* creation. Comments have been added here to assist the reader's understanding.

F.1 Scratch.as

In this section, we show excerpts from the **Scratch.as** file.

```
1  // This is the top-level application.
2
3  package {
4  public class Scratch extends Sprite {
5
6    protected function initialize():void {
7
8   // Code to setup other parameters omitted here.
9
10     // Preload the ev3 extension to ScratchX.
11     var ev3 = new ScratchExtension("ev3", 1);
12     ev3.javascriptURL =
13       "http://sheimlich.github.io/OO-EV3-ScratchExtension/ev3_scratch.js";
14     extensionManager.loadCustom(ev3);
15   }
16
17   public function selectSprite(obj:ScratchObj):void {
18     if (isShowing(imagesPart)) imagesPart.editor.shutdown();
19     if (isShowing(soundsPart)) soundsPart.editor.shutdown();
20     viewedObject = obj;
21     libraryPart.refresh();
22     tabsPart.refresh();
```

```
23      // Do not show the imagesPart if it is a ghost.
24      if (isShowing(imagesPart) && !obj.isGhost) {
25        imagesPart.refresh();
26      }
27      // If somehow the imagesPart should be showing for a ghost,
28      // show the scripts tab instead.
29      if (isShowing(imagesPart) && obj.isGhost) {
30        setTab('scripts');
31      }
32      if (isShowing(soundsPart)) {
33        soundsPart.currentIndex = 0;
34        soundsPart.refresh();
35      }
36      if (isShowing(scriptsPart)) {
37        scriptsPart.updatePalette();
38        scriptsPane.viewScriptsFor(obj);
39        scriptsPart.updateSpriteWatermark();
40        scriptsPart.updatePublicDisplay();
41      }
42    }
43
44    // Function to add a new ghost.
45    public function addNewGhost(spr:ScratchGhost, showImages:Boolean = false,
46            atMouse:Boolean = false):void {
47      // Set all the relevent parameters.
48      spr.objName = stagePane.unusedSpriteName(spr.objName);
49      stagePane.addChild(spr);
50      selectSprite(spr);
51      // Once the ghost is created, update the display.
52      setTab('scripts');
53      setSaveNeeded(true);
54      libraryPart.refresh();
55
56    }
```

F.2 Specs.as

```
1  // This file defines the command blocks and categories.
2  // To add a new command:
3  //    a. add a specification for the new command to the commands array
4  //    b. add a primitive for the new command to the interpreter
5
6  package {
7  public class Specs {
```

```
8
9   public static const GET_VAR:String = "readVariable";
10  public static const SET_VAR:String = "setVar:to:";
11  public static const CHANGE_VAR:String = "changeVar:by:";
12  public static const GET_LIST:String = "contentsOfList:";
13  public static const CALL:String = "call";
14  public static const PROCEDURE_DEF:String = "procDef";
15  public static const GET_PARAM:String = "getParam";
16  // Add a string to represent the Sprite Reporter block.
17  public static const GET_SPRITE:String = "getSprite";
18
19  public static const motionCategory:int = 1;
20  public static const looksCategory:int = 2;
21  public static const eventsCategory:int = 5;
22  public static const controlCategory:int = 6;
23  public static const operatorsCategory:int = 8;
24  public static const dataCategory:int = 9;
25  public static const myBlocksCategory:int = 10;
26  public static const listCategory:int = 12;
27  // Add a constant for the creature block category.
28  public static const creatureCateogry:int = 13;
29  public static const extensionsCategory:int = 20;
30
31  public static var variableColor:int = 0xEE7D16; // Scratch 1.4: 0xF3761D
32  public static var listColor:int = 0xCC5B22; // Scratch 1.4: 0xD94D11
33  public static var procedureColor:int = 0x632D99; // 0x531E99;
34  public static var parameterColor:int = 0x5947B1;
35  public static var extensionsColor:int = 0x4B4A60; // 0x72228C; // 0
     x672D79;
36  //Create a unique color for the creature category.
37  public static var creatureColor:int = 0x0a6320;
38
39  private static const undefinedColor:int = 0xD42828;
40
41  public static const categories:Array = [
42   // id   category name  color
43    [0,  "undefined", 0xD42828],
44    [1,  "Motion",    0x4a6cd4],
45    [2,  "Looks",   0x8a55d7],
46    [3,  "Sound",   0xbb42c3],
47    [4,  "Pen",     0x0e9a6c], // Scratch 1.4: 0x009870
48    [5,  "Events",    0xc88330],
49    [6,  "Control",   0xe1a91a],
50    [7,  "Sensing",   0x2ca5e2],
51    [8,  "Operators", 0x5cb712],
52    [9,  "Data",    variableColor],
53    [10, "More Blocks", procedureColor],
54    [11, "Parameter", parameterColor],
55    [12, "List",    listColor],
56    // Add creatures to the categories.
```

```
57      [13, "Creatures",    creatureColor],
58      [20, "Extension", extensionsColor],
59    ];
60
61
62    public static var commands:Array = [
63      // block specification          type, cat, opcode     default args (
        optional)
64      // motion
65    // Update the motion blocks to accept a Sprite as an argument.
66      ["move %g %n steps",              " ", 1, "forward:",          10],
67      ["turn %g @turnRight %n degrees",     " ", 1, "turnRight:",        15],
68      ["turn %g @turnLeft %n degrees",      " ", 1, "turnLeft:",         15],
69      ["-"],
70      ["point %g in direction %d.direction",    " ", 1, "heading:",
71        90],
72      ["point %g towards %m.spriteOrMouse",   " ", 1, "pointTowards:",
73        ""],
74      ["-"],
75      ["%g go to x:%n y:%n",              " ", 1, "gotoX:y:"],
76      ["%g go to %m.location",          " ", 1, "gotoSpriteOrMouse:",
77        "mousepointer"],
78      ["%g glide %n secs to x:%n y:%n",     " ", 1,
79        "glideSecs:toX:y:elapsed:from:"],
80      ["-"],
81      ["change %g x by %n",             " ", 1, "changeXposBy:",       10],
82      ["set %g x to %n",                " ", 1, "xpos:",             0],
83      ["change %g y by %n",             " ", 1, "changeYposBy:",       10],
84      ["set %g y to %n",                " ", 1, "ypos:",             0],
85      ["-"],
86      ["if %g on edge, bounce",           " ", 1, "bounceOffEdge"],
87      ["-"],
88      ["set %g rotation style %m.rotationStyle",  " ", 1, "setRotationStyle",
89        "left-right"],
90      ["-"],
91      ["x position of %g",              "r", 1, "xpos"],
92      ["y position of %g ",             "r", 1, "ypos"],
93      ["direction of %g",             "r", 1, "heading"],
94
95    // Additional commands omitted.
96
97    ];
98
99    public static var extensionSpecs:Array = ["when %m.booleanSensor", "when
       %m.sensor %m.lessMore %n", "sensor %m.booleanSensor?", "%m.sensor sensor
```

```
    value", "turn %m.motor on for %n secs", "turn %m.motor on", "turn %m.
    motor off", "set %m.motor power to %n", "set %m.motor2 direction to %m.
    motorDirection", "when distance %m.lessMore %n", "when tilt %m.eNe %n",
    "distance", "tilt"];

}}
```

F.3 Resources.as

```
package assets {
public class Resources {

  public static function createBmp(resourceName:String):Bitmap {
    var resourceClass:Class = Resources[resourceName];
    if (!resourceClass) {
      trace('missing resource: ', resourceName);
      return new Bitmap(new BitmapData(10, 10, false, 0x808080));
    }
    return new resourceClass();
  }

  // Other Resources omitted.

  // Add images for the ghost button.
  [Embed(source='UI/newsprite/ghostOff.png')] private static const
    ghostOff:Class;
  [Embed(source='UI/newsprite/ghostOn.png')] private static const
    ghostOn:Class;

  // Other Resources omitted.

  // Thumbnail to appear for ghosts in the SpritePane.
  [Embed(source='UI/newsprite/ghostThumb.png')]
/Hilight     private static const ghostThumb:Class;
}}
```

F.4 Block.as

```
// A Block is a graphical object representing a program statement (command)
// or function (reporter). A stack is a sequence of command blocks, where
// the following command and any nested commands (e.g. within a loop) are
// children. Blocks come in a variety of shapes and usually have some
// combination of label strings and arguments (also children).
//
// The Block class manages block shape, labels, arguments, layout, and
// block sequence. It also supports generation of the labels and argument
// sequence from a specification string (e.g. "%n + %n") and type (e.g.
   reporter).
```

```
10
11 package blocks {
12 public class Block extends Sprite {
13
14     // Other data member omitted.
15
16     // Boolean to know if a procedure is public or private.
17     public var isGlobal:Boolean = false;
18   public var isAsyncHat:Boolean = false;
19   public var isReporter:Boolean = false;
20   public var isTerminal:Boolean = false;  // blocks that end a stack like "
     stop" or "forever"
21
22   public function Block(spec:String, type:String = " ", color:int = 0
     xD00000, op:* = 0, defaultArgs:Array = null) {
23
24     if ((Specs.CALL == op) ||
25       (Specs.GET_LIST == op) ||
26       (Specs.GET_PARAM == op) ||
27       (Specs.GET_VAR == op) ||
28       (Specs.PROCEDURE_DEF == op) ||
29       ('proc_declaration' == op) ||
30       (Specs.GET_SPRITE == op)) {
31         this.spec = spec; // don't translate var/list/param reporters
32     }
33
34     var shape:int;
35     if ((type == " ") || (type == "") || (type == "w")) {
36       base = new BlockShape(BlockShape.CmdShape, color);
37       indentTop = 3;
38     } else if (type == "b") {
39       base = new BlockShape(BlockShape.BooleanShape, color);
40       isReporter = true;
41       indentLeft = 9;
42       indentRight = 7;
43     } else if (type == "r" || type == "R") {
44       this.type = 'r';
45       base = new BlockShape(BlockShape.NumberShape, color);
46       isReporter = true;
47       forceAsync = (type == 'r') && Scratch.app.extensionManager.
     shouldForceAsync(op);
48       isRequester = (type == 'R') || forceAsync;
49       indentTop = 2;
50       indentBottom = 2;
51       indentLeft = 6;
52       indentRight = 4;
53     } else if (type == "h" || type == 'H') {
54       base = new BlockShape(BlockShape.HatShape, color);
55       isHat = true;
```

```
56        forceAsync = (type == 'h') && Scratch.app.extensionManager.
      shouldForceAsync(op);
57        isAsyncHat = (type == 'H') || forceAsync;
58        indentTop = 12;
59      }
60      else if (type == "c") {
61        base = new BlockShape(BlockShape.LoopShape, color);
62      } else if (type == "cf") {
63        base = new BlockShape(BlockShape.FinalLoopShape, color);
64        isTerminal = true;
65      } else if (type == "e") {
66        base = new BlockShape(BlockShape.IfElseShape, color);
67        addChild(elseLabel = makeLabel(Translator.map('else')));
68      } else if (type == "f") {
69        base = new BlockShape(BlockShape.FinalCmdShape, color);
70        isTerminal = true;
71        indentTop = 5;
72      } else if (type == "o") { // cmd outline for proc definition
73        base = new BlockShape(BlockShape.CmdOutlineShape, color);
74        base.filters = []; // no bezel
75        indentTop = 3;
76      } else if (type == "p") {
77        base = new BlockShape(BlockShape.ProcHatShape, color);
78        isHat = true;
79      } else if (type == "cl") {
80        base = new BlockShape(BlockShape.ClassDefShape, color);
81      // g is the char that represents a sprite reporter.
82      // If the type is g, set the appropriate values.
83      } else if (type == "g") {
84        base = new BlockShape(BlockShape.SpriteShape, color);
85        isReporter = true;
86      } else {
87        base = new BlockShape(BlockShape.RectShape, color);
88      }
89
90      // Code omitted.
91    }
92
93    public function setSpec(newSpec:String, defaultArgs:Array = null):void {
94      if (op == Specs.PROCEDURE_DEF) {
95
96      // Code omitted.
97
98      // If the operation is any of the getter reporters.
99      } else if (op == Specs.GET_VAR || op == Specs.GET_LIST ||
100       op == Specs.GET_SPRITE) {
101       labelsAndArgs = [makeLabel(spec)];
102     }
103   }
```

```
104
105   private function argOrLabelFor(s:String, c:int):DisplayObject {
106     // Possible token formats:
107     //  %<single letter>
108     //  %m.<menuName>
109     //  @<iconName>
110     //  label (any string with no embedded white space that does not start
      with % or @)
111     //  a token consisting of a single % or @ character is also a label
112     if (s.length >= 2 && s.charAt(0) == "%") { // argument spec
113       var argSpec:String = s.charAt(1);
114       if (argSpec == "b") return new BlockArg("b", c);
115       if (argSpec == "c") return new BlockArg("c", c);
116       if (argSpec == "d") return new BlockArg("d", c, true, s.slice(3));
117       if (argSpec == "m") return new BlockArg("m", c, false, s.slice(3));
118       if (argSpec == "n") return new BlockArg("n", c, true);
119       if (argSpec == "s") return new BlockArg("s", c, true);
120       // If the block is a sprite reporter, return the appropriate
121       // argument block.
122       if (argSpec == "g") return new BlockArg("g", c);
123     } else if (s.length >= 2 && s.charAt(0) == "@") { // icon spec
124       var icon:* = Specs.IconNamed(s.slice(1));
125       return (icon) ? icon : makeLabel(s);
126     }
127     return makeLabel(ReadStream.unescape(s));
128   }
129
130 }}
```

F.5 BlockArg.as

```
1  // A BlockArg represents a Block argument slot. Some BlockArgs, contain
2  // a text field that can be edited by the user. Others (e.g. booleans)
3  // are immutable. In either case, they be replaced by a reporter block
4  // of the right type. That is, dropping a reporter block onto a BlockArg
5  // inside a block causes the BlockArg to be replaced by the reporter.
6  // If a reporter is removed, a BlockArg is added to the block.
7  //
8  // To create a custom BlockArg widget such as a color picker, make a
9  // subclass of BlockArg for the widget. Your constructor is responsible
10 // for adding child display objects and setting its width and height.
11 // The widget must initialize argValue and update it as the user
12 // interacts with the widget. In some cases, the widget may need to
13 // override the setArgValue() method. If the widget can accept dropped
14 // arguments, it should set base to a BlockShape to support drag feedback.
15
16 package blocks {
17 public class BlockArg extends Sprite {
18
19   // BlockArg types:
```

```
20    //  b - boolean (pointed)
21    //  c - color selector
22    //  d - number with menu (rounded w/ menu icon)
23    //  m - string with menu (rectangular w/ menu icon)
24    //  n - number (rounded)
25    //  s - string (rectangular)
26      // g - Sprite (Flag-shaped)
27    //  none of the above - custom subclass of BlockArg
28    public function BlockArg(type:String, color:int, editable:Boolean = false
        , menuName:String = '') {
29
30      if (type == 'b') {
31        base = new BlockShape(BlockShape.BooleanShape, c);
32        argValue = false;
33      // Create the appropriate shape for a Sprite reporter.
34      } else if (type == 'g') {
35        base = new BlockShape(BlockShape.SpriteShape, c);
36      }
37
38      // Code emitted.
39
40      }
41
42  }}
```

F.6 BlockIO.as

```
1   // Convert blocks and stacks to/from an array structure or JSON string
        format.
2   // The array structure format captures the meaning of scripts in a compact
        form that
3   // is independent of the internal representation and is easy to convert to/
        from JSON.
4
5   package blocks {
6   public class BlockIO {
7
8     private static function blockToArray(b:Block):Array {
9       // Return an array structure for this block.
10      var result:Array = [b.op];
11      if (b.op == Specs.GET_VAR) return [Specs.GET_VAR, b.spec];     //
        variable reporter
12    // Sprite Reporter
13      if (b.op == Specs.GET_SPRITE) return [Specs.GET_SPRITE, b.spec];
14      if (b.op == Specs.GET_LIST) return [Specs.GET_LIST, b.spec];  // list
        reporter
15      if (b.op == Specs.GET_PARAM) return [Specs.GET_PARAM, b.spec, b.type];
        // parameter reporter
16
```

```
    // Code omitted.
  }

  private static function specialCmd(cmd:Array, forStage:Boolean):Block {
    // If the given command is special (e.g. a reporter or old-style a hat block), return a block for it.
    // Otherwise, return null.
    var b:Block;
    switch (cmd[0]) {
    // If a get sprite reporter, create the appropriate block.
    case Specs.GET_SPRITE:
      return new Block(cmd[1], 'r', Specs.creatureColor, Specs.GET_SPRITE);
    case Specs.PROCEDURE_DEF:
      b = new Block('', 'p', Specs.procedureColor, Specs.PROCEDURE_DEF);
      b.parameterNames = cmd[2];
      b.defaultArgValues = cmd[3];
      if (cmd.length > 4) b.warpProcFlag = cmd[4];
        // Include the global flag.
        if (cmd.length > 5) b.isGlobal = cmd[5];
      if (cmd.length > 6) b.owner = cmd[6];
      b.setSpec(cmd[1]);
      b.fixArgLayout();
      return b;
      // Code omitted.
  }
}}
```

F.7 BlockShape.as

```
package blocks {
public class BlockShape extends Shape {

  // Shapes
  public static const RectShape:int = 1;
  public static const BooleanShape:int = 2;
  public static const NumberShape:int = 3;
  public static const CmdShape:int = 4;
  public static const FinalCmdShape:int = 5;
  public static const CmdOutlineShape:int = 6;
  public static const HatShape:int = 7;
  public static const ProcHatShape:int = 8;
  // Added SpriteShape Constant.
  public static const SpriteShape:int = 9;
  // C-shaped blocks
  // Updated to allow the Sprite Shape to be in the
  // normal shape block range.
  public static const LoopShape:int = 10;
  public static const FinalLoopShape:int = 11;
```

```
20    // E-shaped blocks
21    public static const IfElseShape:int = 12;
22    // M-Shaped blocks
23    public static const ClassDefShape:int = 13;
24
25  private function setShape(shape:int):void {
26      this.shape = shape;
27      switch(shape) {
28      case RectShape:      drawFunction = drawRectShape; break;
29      case BooleanShape:     drawFunction = drawBooleanShape; break;
30      case NumberShape:    drawFunction = drawNumberShape; break;
31      // Add a case for the Sprite Shape.
32      case SpriteShape:        drawFunction = drawSpriteShape; break;
33      case CmdShape:
34      case FinalCmdShape:    drawFunction = drawCmdShape; break;
35      case CmdOutlineShape: drawFunction = drawCmdOutlineShape; break;
36      case LoopShape:
37      case FinalLoopShape:  drawFunction = drawLoopShape; break;
38      case IfElseShape:    drawFunction = drawIfElseShape; break;
39      case HatShape:       drawFunction = drawHatShape; break;
40      case ProcHatShape:     drawFunction = drawProcHatShape; break;
41      case ClassDefShape:      drawFunction = drawClassDefShape; break;
42      }
43    }
44
45    //Create the distinct sprite shape.
46  private function drawSpriteShape(g:Graphics):void {
47      var centerY:int = topH / 2;
48      g.moveTo(0, topH);
49      g.lineTo(centerY/2, centerY);
50      g.lineTo(0, 0);
51      g.lineTo(w, 0);
52      g.lineTo(w - centerY/2, centerY);
53      g.lineTo(w, topH);
54    }
55
56  }}
```

F.8 Interpreter.as

```
1   // A simple yet efficient interpreter for blocks.
2   //
3   // Interpreters may seem mysterious, but this one is quite straightforward.
        Since every
4   // block knows which block (if any) follows it in a sequence of blocks, the
        interpreter
```

```
5  // simply executes the current block, then asks that block for the next
       block. The heart
6  // of the interpreter is the evalCmd() function, which looks up the opcode
       string in a
7  // dictionary (initialized by initPrims()) then calls the primitive
       function for that opcode.
8  // Control structures are handled by pushing the current state onto the
       active thread's
9  // execution stack and continuing with the first block of the substack.
       When the end of a
10 // substack is reached, the previous execution state is popped. If the
       substack was a loop
11 // body, control yields to the next thread. Otherwise, execution continues
       with the next
12 // block. If there is no next block, and no state to pop, the thread
       terminates.
13 //
14 // The interpreter does as much as it can within workTime milliseconds,
       then returns
15 // control. It returns control earlier if either (a) there are are no more
       threads to run
16 // or (b) some thread does a command that has a visible effect (e.g. "move
       10 steps").
17 //
18 // To add a command to the interpreter, just add a new case to initPrims().
        Command blocks
19 // usually perform some operation and return null, while reporters must
       return a value.
20 // Control structures are a little tricky; look at some of the existing
       control structure
21 // commands to get a sense of what to do.
22 //
23 // Clocks and time:
24 //
25 // The millisecond clock starts at zero when Flash is started and, since
       the clock is
26 // a 32-bit integer, it wraps after 24.86 days. Since it seems unlikely
       that one Scratch
27 // session would run that long, this code doesn't deal with clock wrapping.
28 // Since Scratch only runs at discrete intervals, timed commands may be
       resumed a few
29 // milliseconds late. These small errors accumulate, causing threads to
       slip out of
30 // synchronization with each other, a problem especially noticeable in
       music projects.
31 // This problem is addressed by recording the amount of time slippage and
       shortening
32 // subsequent timed commands slightly to "catch up".
33 // Delay times are rounded to milliseconds, and the minimum delay is a
       millisecond.
34
35 package interpreter {
```

```
36 public class Interpreter {
37
38 private function initPrims():void {
39     primTable = new Dictionary();
40
41     // variables
42     primTable[Specs.GET_VAR]     = primVarGet;
43     primTable[Specs.SET_VAR]     = primVarSet;
44     primTable[Specs.CHANGE_VAR]    = primVarChange;
45     primTable[Specs.GET_PARAM]     = primGetParam;
46     \\ Add the Sprite reporter to the primitive dictionary.
47     primTable[Specs.GET_SPRITE]      = primGetSprite;
48
49     // Code emitted.
50
51 }
52
53   // Add the function for the sprite reporter block.
54   private function primGetSprite(b:Block):* {
55     // The Sprite's name is the block's spec.
56     var name:String = b.spec;
57     // Lookup the sprite based on the name.
58     var sprites:Array = app.stagePane.spritesAndClonesNamed(name);
59     for each (var s:ScratchSprite in sprites) {
60       return s;
61     }
62   }
```

F.9 Variable.as

```
1  // A variable is a name-value pair.
2
3  package interpreter {
4  public class Variable {
5
6    // Add the creator of the variable to enable private variables to pass
7    // from Nymph to Sprite.
8    public var creator:ScratchObj;
9
10   // Add the creator to the instantiation.
11   public function Variable(vName:String, initialValue:*, c:ScratchObj) {
12     name = vName;
13     value = initialValue;
14     creator = c;
15   }
```

F.10 MotionAndPenPrims.as

One of the motion and pen primatives has been shown. This was updated to enable a Sprite to be a parameter of the function. The other motion primitive functions have been updated in a similar manner.

```
private function primMove(b:Block):void {
  // Grab the first argument from the block.  This is the Sprite the block
  // will update.
  var s:ScratchSprite = interp.arg(b, 0);
  if (s == null) return;
  var radians:Number = (Math.PI * (90 - s.direction)) / 180;
  var d:Number = interp.numarg(b, 1);
  moveSpriteTo(s, s.scratchX + (d * Math.cos(radians)), s.scratchY + (d *
   Math.sin(radians)));
}
```

F.11 PaletteBuilder.as

```
package scratch {
public class PaletteBuilder {
public function showBlocksForCategory(selectedCategory:int, scrollToOrigin:
   Boolean, shiftKey:Boolean = false):void {
    if (app.palette == null) return;

    app.palette.clear(scrollToOrigin);
    nextY = 7;

    // If the viewed object is a Ghost and the event tab is selected.
    if (app.viewedObj() &&
      app.viewedObj().isGhost &&
      selectedCategory == 5) {
      // Don\rq{}t show any blocks.
      addItem(makeLabel(Translator.map('Ghost Selected:')));
      nextY -= 6;
      addItem(makeLabel(Translator.map('No event blocks')));
      return;
    }

    if (selectedCategory == Specs.dataCategory) return showDataCategory();
    if (selectedCategory == Specs.myBlocksCategory) return
   showMyBlocksPalette(shiftKey);
    // Use a special function if the creature category should be shown.
    if (selectedCategory == Specs.creatureCateogry)
      return showCreaturesCategory();
```

```
25
26      var catName:String = Specs.categories[selectedCategory][1];
27      var catColor:int = Specs.blockColor(selectedCategory);
28      if (app.viewedObj() && app.viewedObj().isStage) {
29        // The stage has different blocks for some categories:
30        var stageSpecific:Array = ['Control', 'Looks', 'Motion', 'Pen', '
    Sensing'];
31        if (stageSpecific.indexOf(catName) != -1) selectedCategory += 100;
32        if (catName == 'Motion') {
33          addItem(makeLabel(Translator.map('Stage selected:')));
34          nextY -= 6;
35          addItem(makeLabel(Translator.map('No motion blocks')));
36          return;
37        }
38      }
39      addBlocksForCategory(selectedCategory, catColor);
40      updateCheckboxes();
41    }
42
43    private function showMyBlocksPalette(shiftKey:Boolean):void {
44      // show creation button, hat, and call blocks
45      var catColor:int = Specs.blockColor(Specs.procedureColor);
46      addItem(new Button(Translator.map('Make a Block'), makeNewBlock, false,
      '/help/studio/tips/blocks/make-a-block/'));
47
48      // Add procedures if they are visible based on
49      // the isGlobal boolean.
50      var definitions:Array =
51        app.runtime.visibleProcs().concat(
52          app.runtime.ObjProcs(app.viewedObj().getNymph()));
53      if (definitions.length > 0) {
54        nextY += 5;
55        for each (var proc:Block in definitions) {
56          var b:Block = new Block(proc.spec, ' ', Specs.procedureColor,
57            Specs.CALL, proc.defaultArgValues);
58          addItem(b);
59        }
60        nextY += 5;
61      }
62
63    // Show the Sprite and Ghost reporters
64    private function showCreaturesCategory():void {
65      // Get all Sprites and Ghosts.
66      var sprites:Array = app.stagePane.sprites();
67      // Display all Sprites and Ghosts.
68      for each (var s:ScratchSprite in sprites) {
69        nextY += 5;
```

```
70        var b:Block = new Block(s.objName, "g", Specs.creatureColor, Specs.
      GET_SPRITE);
71        addItem(b)
72        nextY += 5;
73      }
74    }
75
76  protected function createVar(name:String, varSettings:VariableSettings):* {
77      var obj:ScratchObj = (varSettings.isLocal) ? app.viewedObj() : app.
      stageObj();
78      if (obj.hasName(name)) {
79        DialogBox.notify("Cannot Add", "That name is already in use.");
80        return;
81      }
82      // Update the scope of variables to include the Nymph
83      var variable:* = (varSettings.isList ?
84        obj.lookupOrCreateList(name, app.viewedObj()) :
85        obj.lookupOrCreateVar(name, app.viewedObj()));
86
87      app.runtime.showVarOrListFor(name, varSettings.isList, obj);
88      app.setSaveNeeded();
89
90      return variable;
91    }
92  }}
```

F.12 ScratchGhost.as

```
1  // The ScratchGhost class removes a lot of the functionality of a
       ScratchSprite.
2  package scratch {
3  public class ScratchGhost extends ScratchSprite {
4
5    public function ScratchGhost(name:String = null) {
6      // Set the name to be based off Ghost.
7      objName = Scratch.app.stagePane.unusedSpriteName(name || Translator.map
       ('Ghost1'));
8      filterPack = new FilterPack(this);
9      img = new Sprite();
10     img.cacheAsBitmap = true;
11     addChild(img);
12     // Set isGhost to true.
13     isGhost = true;
14   }
15 }}
```

F.13 ScratchObj.as

```
1  package scratch {
2  public class ScratchObj extends Sprite {
3
4    public var nymphName:String = 'no nymph';
5    public var myChild:ScratchObj = null;
6    public var isStage:Boolean = false;
7    public var isGhost:Boolean = false;
8    public var variables:Array = [];
9    public var lists:Array = [];
10
11   \\ Code omitted.
12
13   private var nymph:ScratchObj = null;
14
15   public function updateCostume():void {
16     updateImage();
17     // Update the costumes to include the Nymph.
18     if(myChild != null) myChild.updateCostumesAsChild();
19   }
20
21   // Update function to include creator objects.
22   // This assists in providing private functions from Nymph
23   // to Sprite.
24 public function lookupOrCreateVar(varName:String, creator:ScratchObj=null):
     Variable {
25     // Lookup and return a variable. If lookup fails, create the variable
     in this object.
26     var v:Variable = lookupVar(varName);
27     if (v == null) { // not found; create it
28       v = new Variable(varName, 0, creator);
29       variables.push(v);
30       Scratch.app.updatePalette(false);
31     }
32     return v;
33   }
34
35   // Update function to include creator objects.
36   // This assists in providing private functions from Nymph
37   // to Sprite.
38   public function lookupOrCreateList(listName:String, obj:ScratchObj = null
     ):ListWatcher {
39     // Look and return a list. If lookup fails, create the list in this
     object.
40     var list:ListWatcher = lookupList(listName);
41     if (list == null) { // not found; create it
42       list = new ListWatcher(listName, [], this, false, obj);
43       lists.push(list);
```

```
44        Scratch.app.updatePalette(false);
45      }
46      return list;
47    }
48
49  // Nymph Getter.
50    public function getNymph():ScratchObj {
51      return nymph;
52    }
53
54    // Delete a Nymph, including removing the costumes.
55    public function deleteNymph():void {
56      if (nymph == null) return;
57
58      for each (var c:ScratchCostume in nymph.costumes) {
59        for each (var l:ScratchCostume in this.costumes) {
60          if (c == l) {
61            deleteCostume(c);
62          }
63        }
64      }
65      nymph = null;
66      nymphName = null;
67    }
68
69    // Set the Nymph
70    public function setNymph(s:ScratchObj):void {
71      // Begin by deleting any previous Nymph.
72      deleteNymph();
73      // Set the new Nymph and Name.
74      nymph = s;
75      nymphName = s.objName;
76      // Set the child to be this object.
77      nymph.myChild = this;
78      // Obtain the nymph costumes.
79      costumes = costumes.concat(nymph.costumes);
80      for each (var c in nymph.costumes) {
81        for each (var l in this.costumes) {
82          if (c == l) break;
83        }
84        costumes.concat(c);
85      }
```

```
86    }
87
88    // Get the Nymph Costumes
89    public function updateCostumesAsChild() {
90      // Begin by removing all the nymph costumes to ensure no duplicates.
91      for each (var c:ScratchCostume in nymph.costumes) {
92        for each (var l:ScratchCostume in this.costumes) {
93          if (c == l) {
94            deleteCostume(c);
95          }
96        }
97      }
98      // Then add all the costumes to this sprite.
99      costumes = costumes.concat(nymph.costumes);
100     for each (var c in nymph.costumes) {
101       for each (var l in this.costumes) {
102         if (c == l) break;
103       }
104       costumes.concat(c);
105     }
106
107   }
```

F.14 ScratchRuntime.as

```
1   package scratch {
2   public class ScratchRuntime {
3
4     public function allVarNames():Array {
5       var result:Array = [], v:Variable;
6       for each (v in app.stageObj().variables) result.push(v.name);
7       if (!app.viewedObj().isStage) {
8         for each (v in app.viewedObj().variables) result.push(v.name);
9         // If the variable belongs to the Nymph, it should be included
10        // in the variable names.
11        if(app.viewedObj().getNymph() != null) {
12          for each (v in app.viewedObj().getNymph().variables) result.push(v.
      name);
13        }
14      }
15      return result;
16    }
17
18    // Return all the user defined procedures that are visible
```

```
19  // based on the isGlobal boolean.
20  public function visibleProcs(obj:ScratchObj = null):Array {
21    var result:Array = [], b:Block, s:ScratchObj;
22    // The default is the viewed-object.
23    if (obj == null) obj = app.viewedObj();
24    // For all the Scratch Objects
25    for each (s in app.stageObj().sprites().concat(app.stageObj())) {
26      // For all the procedure definitions
27      for each (b in s.procedureDefinitions()) {
28        // Determine if the procedure is visible.
29        var local:Array = obj.procedureDefinitions();
30        var flag:Boolean = false;
31        // Find if the ProcDef is for the viewed object
32        for each (var l:Block in local) {
33          if (l == b) {
34            flag = true;
35            break;
36          }
37      }
38        if(b.isGlobal || flag)
39          result.push(b);
40      }
41    }
42    return result;
43  }
44
45  // Return all the procedure definitions for the given object.
46  public function ObjProcs(s:ScratchObj):Array {
47    if (s == null) return [];
48    var result:Array = [], b:Block;
49    for each (b in s.procedureDefinitions()) {
50      result.push(b);
51    }
52    return result;
53  }
54
55  // Return the Scratch Object that owns the given procedure.
56  public function procOwner(b:Block):ScratchObj {
57    var s:ScratchObj, l:Block;
58    for each (s in app.stageObj().sprites().concat(app.stageObj())) {
59      for each (l in s.procedureDefinitions()) {
```

```
60          if (b == l) {
61            return s;
62          }
63        }
64      }
65      return null;
66    }
67
68    // Include the object that created the variable.
69    public function makeVariable(varObj:Object):Variable {
70      return new Variable(varObj.name, varObj.value, app.viewedObj());
71    }
72
73    public function allListNames():Array {
74      var result:Array = app.stageObj().listNames();
75      if (!app.viewedObj().isStage) {
76        result = result.concat(app.viewedObj().listNames());
77        // If there is a Nymph, include those lists
78        // in the Sprite lists.
79        if(app.viewedObj().getNymph() != null) {
80          result = result.concat(app.viewedObj().getNymph().listNames())
81        }
82      }
83      return result;
84    }
```

F.15 PaletteSelector.as

```
1  package ui {
2  public class PaletteSelector extends Sprite {
3    private static const categories:Array = [
4      \\ Add Creatures to column one.
5      'Motion', 'Looks', 'Sound', 'Pen', 'Data', 'Creatures', // column 1
6      'Events', 'Control', 'Sensing', 'Operators', 'More Blocks']; // column
       2
7    ];
8
9    private function initCategories():void {
10     \\ increase the number of rows to 6.
11     const numberOfRows:int = 6;
12   }
13 }}
```

F.16 ProcedureSpecEditor.as

```
package ui {
public class ProcedureSpecEditor extends Sprite {

 private var owner:ScratchObj;

  public function ProcedureSpecEditor(originalSpec:String, inputNames:Array
    , warpFlag:Boolean, globalFlag:Boolean, o:ScratchObj) {
    owner = o;

    // Code emitted.

    addSpecElements(originalSpec, inputNames);
    warpCheckbox.setOn(warpFlag);
    // Add checkbox to set isGlobal flag.
    globalButton.setOn(globalFlag);
    showButtons(false);

  }

  public static function strings():Array {
    return [
      'Options', 'Run without screen refresh',
      'Add number input:',
      'Add string input:',
      'Add boolean input:',
        \\ String to display for adding a sprite input parameter.
        'Add sprite input:',
      'Add label text:',
      'text',
    ];
  }

  private function addSpecElements(spec:String, inputNames:Array):void {
    function addElement(o:DisplayObject):void {
      row.push(o);
      addChild(o);
    }
    clearRow();
    var i:int = 0;
    for each (var s:String in ReadStream.tokenize(spec)) {
      if (s.length >= 2 && s.charAt(0) == '%') { // argument spec
        var argSpec:String = s.charAt(1);
        var arg:BlockArg = null;
        if (argSpec == 'b') arg = makeBooleanArg();
        if (argSpec == 'n') arg = makeNumberArg();
        if (argSpec == 's') arg = makeStringArg();
        // Add Sprite Parameter
        if (argSpec == 'g') arg = makeSpriteArg();
        if (arg) {
```

```
49            arg.setArgValue(inputNames[i++]);
50            addElement(arg);
51          }
52        }
53
54        // Code emitted
55
56    }
57
58    public function defaultArgValues():Array {
59      var result:Array = [];
60      for each (var el:* in row) {
61        if (el is BlockArg) {
62          var arg:BlockArg = BlockArg(el);
63          var v:* = 0;
64          if (arg.type == 'b') v = false;
65          if (arg.type == 'n') v = 1;
66          if (arg.type == 's') v = '';
67          \\ The default nymph value is null.
68          if (arg.type == 'g') v = null;
69          result.push(v);
70        }
71      }
72      return result;
73    }
74
75    private function addButtonsAndLabels():void {
76      buttonLabels = [
77        makeLabel('Add number input:', 14),
78        makeLabel('Add string input:', 14),
79        makeLabel('Add boolean input:', 14),
80        makeLabel('Add Sprite input:', 14),
81        makeLabel('Add label text:', 14)
82      ];
83      buttons = [
84        new Button('', function():void { appendObj(makeNumberArg()) }),
85        new Button('', function():void { appendObj(makeStringArg()) }),
86        new Button('', function():void { appendObj(makeBooleanArg()) }),
87        new Button('', function():void { appendObj(makeSpriteArg())}),
88        new Button(Translator.map('text'), function():void { appendObj(
   makeTextField('')) })
89      ];
90
91      const lightGray:int = 0xA0A0A0;
92
93      icon = new BlockShape(BlockShape.NumberShape, lightGray);
94      icon.setWidthAndTopHeight(25, 14, true);
95      buttons[0].setIcon(icon);
96
97      icon = new BlockShape(BlockShape.RectShape, lightGray);
```

```
98      icon.setWidthAndTopHeight(22, 14, true);
99      buttons[1].setIcon(icon);
100
101     var icon:BlockShape = new BlockShape(BlockShape.BooleanShape, lightGray
        );
102     icon.setWidthAndTopHeight(25, 14, true);
103     buttons[2].setIcon(icon);
104
105     // Add the Sprite Shape to the button.
106     var icon:BlockShape = new BlockShape(BlockShape.SpriteShape, lightGray)
        ;
107     // Add the button to the editor.
108     icon.setWidthAndTopHeight(25, 14, true);
109     buttons[3].setIcon(icon);
110
111     for each (var label:TextField in buttonLabels) addChild(label);
112     for each (var b:Button in buttons) addChild(b);
113   }
114
115   // Add a Sprite parameter to the new procedure.
116   private function makeSpriteArg():BlockArg {
117     var result:BlockArg = new BlockArg('s', 0xFFFFFF, true);
118     result.setArgValue(unusedArgName('sprite'));
119     return result;
120   }
121
122 }}
```

F.17 GhostThumbnail.as

```
1  package ui {
2  public class SpriteThumbnail extends Sprite {
3  public function updateThumbnail(translationChanged:Boolean = false):void {
4      if (targetObj == null) return;
5      // If a ghost, there is no costume to be the thumbnail.
6      // Instead, set the provided image of a ghost as the
7      // thumbnail.
8      if(targetObj.isGhost) thumbnail.bitmapData =
9        Resources.createBmp("ghostThumb").bitmapData;
10
11     // Code Emitted.
12
13   }
14 }}
```

F.18 LibraryPart.as

```
package ui.parts {
public class LibraryPart extends UIPart {
  private var ghostTitle:TextField;
  private var ghostButton:IconButton;

public function LibraryPart(app:Scratch) {
    this.app = app;
    shape = new Shape();
    addChild(shape);

    // Make the create ghost button.
    ghostTitle = makeLabel(Translator.map('New ghost:'),
      CSS.titleFormat, app.isMicroworld ? 2: stageAreaWidth + 2, 5);
    addChild(ghostTitle);
    addChild(ghostButton = makeButton(makeGhost, 'ghost'));

    // Code emitted.

    addSpritesArea();

  }

    // Make a ghost.
    private function makeGhost(b:IconButton):void {
      var s:ScratchGhost = new ScratchGhost();
      app.addNewGhost(s);
    }
}}
```

F.19 ScriptsPart.as

```
package ui.parts {
public class ScriptsPart extends UIPart {

private var publicDisplay:Sprite;
private var procs:Array = [];

  public function ScriptsPart(app:Scratch) {
    this.app = app;

    addChild(shape = new Shape());
    addChild(spriteWatermark = new Bitmap());
    addXYDisplay();
    addPublicDisplay();
```

```
14      addChild(selector = new PaletteSelector(app));
15
16      // Code Emitted.
17
18    }
19
20    // Add the encapsulation display to the scripting area background.
21    private function addPublicDisplay():void {
22      publicDisplay = new Sprite();
23      // Make title.
24      publicDisplay.addChild(makeLabel('My Interface:', readoutTitleFormat,
25        0, 0));
26      var nextY:Number = 13;
27      publicDisplay.addChild(makeLabel('Procedures:', readoutTitleFormat, 0,
28        nextY));
29      nextY += 13;
30      // Add all the visible procedures.
31      for each (var p:Block in app.runtime.visibleProcs()) {
32        if (p.isGlobal && app.runtime.procOwner(p) == app.viewedObj()) {
33          procs.concat(p);
34          publicDisplay.addChild(makeLabel("\t" + p.spec, readoutLabelFormat,
35            0, nextY));
36          nextY += 13;
37        }
38      }
39      // Add all the visible variables.
40      if (app.viewedObj() != null) {
41        publicDisplay.addChild(makeLabel('Variables:', readoutTitleFormat, 0,
42          nextY));
43        nextY += 13;
44        for each (var obj:ScratchObj in app.stageObj().sprites().concat(
45          app.stageObj())) {
46          for each (var v:Variable in obj.variables) {
47            if (app.viewedObj() == v.creator && app.stageObj().ownsVar(
48              v.name)) {
49              publicDisplay.addChild(makeLabel("\t" +
50                v.name, readoutLabelFormat, 0, nextY));
51              nextY += 13;
52            }
53          }
54        }
55        // Add all the visible lists.
```

```
56        publicDisplay.addChild(makeLabel('Lists:', readoutTitleFormat, 0,
57          nextY));
58        nextY += 13;
59        for each (var obj:ScratchObj in app.stageObj().sprites().concat(
60          app.stageObj())) {
61          for each (var l:ListWatcher in obj.lists) {
62            if (app.viewedObj() == l.creator && app.stageObj().ownsList(
63              l.listName)) {
64              publicDisplay.addChild(makeLabel("\t" +
65                l.listName, readoutLabelFormat, 0, nextY));
66              nextY += 13;
67            }
68          }
69        }
70      }
71
72      addChild(publicDisplay);
73    }
74
75    public function updatePublicDisplay():void {
76      // Clear the display.
77      while (publicDisplay.numChildren > 0) {
78        publicDisplay.removeChildAt(0);
79      }
80      // Create a new display.
81      addPublicDisplay();
82      fixlayout();
83
84  }}
```

F.20 SpriteInfoPart.as

```
1  package ui.parts {
2  public class SpriteInfoPart extends UIPart implements DragClient {
3    // Location to set the Nymph.
4    private var nymphName:EditableLabel;
5    private var nymphLabel:TextField;
6
7    public function refresh():void {
8      spriteName.setContents(app.viewedObj().objName);
9      // If the nymph is already set, display it in the appropriate field.
10     nymphName.setContents(app.viewedObj().nymphName);
11
```

```
12      updateSpriteInfo();
13      if (app.stageIsContracted) layoutCompact();
14      else layoutFullsize();
15
16      // Code emitted.
17
18    }
19
20    private function addParts():void {
21      addChild(closeButton = new IconButton(closeSpriteInfo, 'backarrow'));
22      closeButton.isMomentary = true;
23
24      // Add the field to set the nymph to the display.
25      addChild(spriteName = new EditableLabel(nameChanged));
26      spriteName.setWidth(200);
27
28      addChild(nymphLabel = makeLabel('Nymph:', readoutLabelFormat));
29      addChild(nymphName = new EditableLabel(nymphChanged));
30      nymphName.setWidth(200);
31
32      // Code emitted.
33
34  }
35
36    private function nymphChanged():void {
37      var n:ScratchObj;
38      // Set the nymph as the Stage if appropriate
39      if(nymphName.contents() == app.stageObj().objName) n = app.stageObj();
40      // Otherwise, find the Sprite with the correct name
41      else {
42        for each (var s2:ScratchObj in
43          app.stagePane.spritesAndClonesNamed(nymphName.contents()))
44          n = s2;
45      }
46      // Set the Nymph.
47      app.viewedObj().setNymph(n);
48      app.updatePalette(false);
49    }
50
51  }}
```

F.21 TabsPart.as

Previously the **makeTabs** function was in the constructor. This was refactored into a separate function for re usability.

```
1  package ui.parts {
```

```
2  public class TabsPart extends UIPart {
3    public function TabsPart(app:Scratch) {
4
5      this.app = app;
6      makeTabs();
7
8    }
9
10   private function makeTabs():void {
11     function selectScripts(b:IconButton):void { app.setTab('scripts') }
12     function selectImages(b:IconButton):void { app.setTab('images') }
13     function selectSounds(b:IconButton):void { app.setTab('sounds') }
14
15     scriptsTab = makeTab('Scripts', selectScripts);
16     imagesTab = makeTab('Images', selectImages); // changed to 'Costumes'
       or 'Scenes' by refresh()
17     soundsTab = makeTab('Sounds', selectSounds);
18     addChild(scriptsTab);
19     addChild(imagesTab);
20     // The ghost tab should not have costumes.
21     // Hide the tab to enforce this rule.
22     if(app.viewedObj() && app.viewedObj().isGhost) imagesTab.visible =
       false;
23     addChild(soundsTab);
24     scriptsTab.turnOn();
25   }
26
27   public function refresh():void {
28     // With the addition of Ghosts, the number of tabs changes.
29     // Remove all tabs and remake them to ensure the proper
30     // tabs are shown.
31     while (this.numChildren > 0) {
32       this.removeChildAt(0);
33     }
34     makeTabs();
35     var label:String = ((app.viewedObj() != null) && app.viewedObj().
       isStage) ? 'Backdrops' : 'Costumes';
36     imagesTab.setImage(makeTabImg(label, true), makeTabImg(label, false));
37     fixLayout();
38   }
39
40   public function selectTab(tabName:String):void {
41     scriptsTab.turnOff();
42     imagesTab.turnOff();
43     soundsTab.turnOff();
44     if (tabName == 'scripts') scriptsTab.turnOn();
45     \\ Do not turn on the images tab if a ghost is being viewed.
```

```
46      if (tabName == 'images' && !app.viewedObj().isGhost) imagesTab.turnOn()
        ;
47      if (tabName == 'sounds') soundsTab.turnOn();
48    }
49
50    public function fixLayout():void {
51      scriptsTab.x = 0;
52      scriptsTab.y = 0;
53      // If a ghost is not being viewed, display the costume and sound tabs.
54      if(app.viewedObj() != null && !app.viewedObj().isGhost) {
55        imagesTab.x = scriptsTab.x + scriptsTab.width + 1;
56        imagesTab.y = 0;
57        soundsTab.x = imagesTab.x + imagesTab.width + 1;
58        soundsTab.y = 0;
59      // If a ghost is being viewed, only display the sound tab.
60      } else {
61        soundsTab.x = scriptsTab.x + scriptsTab.width + 1;
62        soundsTab.y = 0;
63      }
64      this.w = soundsTab.x + soundsTab.width;
65      this.h = scriptsTab.height;
66    }
67  }}
```

F.22 ScriptsPane.as

```
1   // A ScriptsPane is a working area that holds blocks and stacks. It
        supports the
2   // logic that highlights possible drop targets as a block is being dragged
        and
3   // decides what to do when the block is dropped.
4
5   package uiwidgets {
6   public class ScriptsPane extends ScrollFrameContents {
7     // This function determines whether a reporter can be placed into
8     // a procedure block argument.
9     private function dropCompatible(droppedBlock:Block, target:DisplayObject)
        :Boolean {
10
11      // Code emitted.
12
13      var dropType:String = droppedBlock.type;
14      var targetType:String = target is Block ? Block(target.parent).argType(
        target).slice(1) : BlockArg(target).type;
15      if (targetType == 'm') {
16        if (Block(target.parent).type == 'h') return false;
```

```
17          return menusThatAcceptReporters.indexOf(BlockArg(target).menuName) >
        -1;
18      }
19      if (targetType == 'b') return dropType == 'b';
20      // Only allow Sprite reporters into Sprite procedure block arguments.
21      if (targetType == 'g') return dropType == 'g';
22      return true;
23    }
24  }}
```

Bibliography

[1] "2015 FIRST LEGO League (FLL) Challenge," [Accessed 7-April-2016]. [Online]. Available: http://www.firstlegoleague.org/sites/default/files/TRASH-TREK-Challenge.pdf

[2] "Digital Technologies," [Accessed April 6, 2016]. [Online]. Available: http://www.australiancurriculum.edu.au/technologies/digital-technologies/curriculum/f-10?layout=1

[3] "Downloads," [Accessed: 30-May-2015]. [Online]. Available: http://education.lego.com/en-au/downloads/?q=%7bdc0ce993-6544-45a1-8680-b2a547d1eeb6%7d

[4] "Education System in the UK," [Accessed 7-April-2016]. [Online]. Available: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/219167/v01-2012ukes.pdf

[5] "Engineering for the Developing World," [Accessed: 6-April-2016]. [Online]. Available: http://www.engineeringchallenges.org/cms/7126/7356.aspx

[6] "FIRST LEGO League Jr." [Accessed 7-April-2016]. [Online]. Available: http://www.firstinspires.org/robotics/flljr

[7] "FIRST Robotics Competition," [Accessed 7-April-2016]. [Online]. Available: https://firstaustralia.org/programs/first-robotics-competition/

[8] "GitHub Status," [Accessed May 30, 2016]. [Online]. Available: https://status.github.com/graphs/past_month

[9] "LEARN TO PROGRAM," [Accessed: 31-May-2015]. [Online]. Available: http://www.lego.com/en-us/mindstorms/learn-to-program

[10] "LEGO MINDSTORMS Ed EV3 Base Set," [Accessed: 30-May-2015]. [Online]. Available: http://www.mooreed.com.au/LEGOregEducationResources/LEGOMindstormsEdEV3BaseSet.aspx

[11] "LEGO MINDSTORMS EV3," [Accessed: 30-May-2015]. [Online]. Available: http://shop.lego.com/en-AU/LEGO-MINDSTORMS-EV3-31313

[12] "LEGO MINDSTORMS EV3 source code," [Accessed: 30-May-2015]. [Online]. Available: https://github.com/mindboards/ev3sources

[13] "Scratch," [Accessed 7-April-2016]. [Online]. Available: https://scratch.mit.edu/

[14] "Scratch Statistics," [Accessed May 29, 2016]. [Online]. Available: https://scratch.mit.edu/statistics/

[15] "Creating Blocks for LEGO Mindstorms EV3," Tech. Rep., 2013, [Accessed: 11-April-2016]. [Online]. Available: https://education.lego.com/en-au/learn/middle-school/mindstorms-ev3/support/ev3-developer-kits

[16] "Hackable Lego Robot Runs Linux," January 2013, [Accessed: 30-May-2015]. [Online]. Available: https://www.linux.com/news/software/applications/688254-hackable-lego-robot-runs-linux

[17] "LEGO Mindstorms EV3 Firmware Developer Kit," Tech. Rep., 2013, [Accessed 7-April-2016]. [Online]. Available: https://education.lego.com/en-au/learn/middle-school/mindstorms-ev3/support/ev3-developer-kits

[18] "National curriculum in England: computing programmes of study," Sep 2013, [Accessed April 6, 2016]. [Online]. Available: https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study

[19] "ROBOLAB 2.9.4c patch," July 2013, [Accessed: 30-May-2015]. [Online]. Available: http://www.legoengineering.com/robolab-2-9-4c-patch/

[20] "Desktop Operating System Market Share," April 2015, [Accessed: 30-May-2015]. [Online]. Available: http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0

[21] "FTC Tech Talk: Programming," March 2015, [Accessed 7-April-2016]. [Online]. Available: http://firsttechchallenge.blogspot.com.au/2015/03/ftc-tech-talk-programming.html

[22] "Subclass 457 quarterly report: quarter ending at 31 December 2015," Tech. Rep., 2015, [Accessed: 6-April-2016]. [Online]. Available: http://www.border.gov.au/ReportsandPublications/Documents/statistics/457-quarterly-report-2015-12-31.pdf#search=457quarterlyreport

[23] D. H. Akehurst, "An OO visual language definition approach supporting multiple views," in *Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on.* IEEE, 2000, pp. 57–58.

[24] O. Astrachan and D. Reed, "AAA and CS 1: the applied apprenticeship approach to CS 1," *ACM SIGCSE Bulletin*, vol. 27, no. 1, pp. 1–5, 1995.

[25] O. Banyasad and P. T. Cox, "Design and Implementation of an Editor/Interpreter for a Visual Logic Programming Language," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 06, pp. 801–838, 2013.

[26] M. Ben-Ari, "Constructivism in computer science education," *Journal of Computers in Mathematics and Science Teaching*, vol. 20, no. 1, pp. 45–73, 2001.

[27] A. F. Blackwell, "Metacognitive theories of visual programming: what do we think we are doing?" in *Visual Languages, 1996. Proceedings., IEEE Symposium on.* IEEE, 1996, pp. 240–246.

[28] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander, "Threshold concepts in computer science: do they exist and are they useful?" in *ACM SIGCSE Bulletin*, vol. 39, no. 1. ACM, 2007, pp. 504–508.

[29] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, pp. 471–523, 1985.

[30] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-D tool for introductory programming concepts," in *Journal of Computing Sciences in Colleges*, vol. 15, no. 5. Consortium for Computing Sciences in Colleges, 2000, pp. 107–116.

[31] R. Decker and S. Hirshfield, "The top 10 reasons why object-oriented programming can't be taught in CS 1," *ACM SIGCSE Bulletin*, vol. 26, no. 1, pp. 51–55, 1994.

[32] G. Dowek, *Principles of programming languages.* Springer Science & Business Media, 2009.

[33] D. A. Economics *et al.*, "Australia's digital pulse: key challenges for our nation: digital skills, jobs and education," 2015.

[34] B. Erwin, M. Cyr, and C. Rogers, "LEGO engineer and RoboLab: Teaching engineering with LABView from kindergarten to graduate school," *International Journal of Engineering Education*, vol. 16, no. 3, pp. 181–192, 2000.

[35] ——, "LEGO engineer and ROBOLAB: Teaching engineering with LabVIEW from kindergarten to graduate school," *International Journal of Engineering Education*, vol. 16, no. 3, pp. 181–192, 2000.

[36] Q. FACTS, "Designing personal robots for education: Hardware, software, and curriculum," 2008.

[37] E. J. Golin, "Theory of visual languages," *Journal of Visual Languages & Computing*, vol. 2, no. 4, pp. 309–310, 1991.

[38] E. J. Golin and S. P. Reiss, "The specification of visual language syntax," in *Visual Languages, 1989., IEEE Workshop on.* IEEE, 1989, pp. 105–110.

[39] T. R. Green, M. Petre, and R. Bellamy, "Comprehensibility of visual and textual programs: A test of superlativism against the match-mismatch conjecture," *ESP*, vol. 91, no. 743, pp. 121–146, 1991.

[40] IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board, "IEEE Recommended Practice for Software Requirements Specifications." Institute of Electrical and Electronics Engineers, 1998.

[41] F. Klassner and S. D. Anderson, "Lego MindStorms: Not just for K-12 anymore," *IEEE Robotics & Automation Magazine*, vol. 10, no. 2, pp. 12–18, 2003.

[42] M. Kölling and J. Rosenberg, "Bluea language for teaching object-oriented programming," in *ACM SIGCSE Bulletin*, vol. 28, no. 1. ACM, 1996, pp. 190–194.

[43] J. Kramer, "Is abstraction the key to computing?" *Communications of the ACM*, vol. 50, no. 4, pp. 36–42, 2007.

[44] U. Last and A. Aharoni-Etzioni, "Secrets and reasons for secrecy among school-aged children: Developmental trends and gender differences," *The Journal of Genetic Psychology*, vol. 156, no. 2, pp. 191–203, 1995.

[45] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.

[46] J. H. Meyer and R. Land, "Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning," *Higher education*, vol. 49, no. 3, pp. 373–388, 2005.

[47] B. A. Myers, "Visual programming, programming by example, and program visualization: a taxonomy," in *ACM SIGCHI Bulletin*, vol. 17, no. 4. ACM, 1986, pp. 59–66.

[48] G. Pascoe, "Elements of object-oriented programming." *Byte*, vol. 11, no. 8, pp. 139–144, 1986.

[49] K. Powers, P. Gross, S. Cooper, M. McNally, K. J. Goldman, V. Proulx, and M. Carlisle, "Tools for teaching introductory programming: what works?" in *ACM SIGCSE Bulletin*, vol. 38, no. 1. ACM, 2006, pp. 560–561.

[50] J. Rekers and A. Schürr, "Defining and parsing visual languages with layered graph grammars," *Journal of Visual Languages & Computing*, vol. 8, no. 1, pp. 27–55, 1997.

[51] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[52] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer science education*, vol. 13, no. 2, pp. 137–172, 2003.

[53] I. E. Sigel, "Developmental trends in the abstraction ability of children," *Child Development*, pp. 131–144, 1953.

[54] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *ACM Sigplan Notices*, vol. 21, no. 11. ACM, 1986, pp. 38–45.

[55] P. Starkey, E. S. Spelke, and R. Gelman, "Numerical abstraction by human infants," *Cognition*, vol. 36, no. 2, pp. 97–127, 1990.

[56] B. Stroustrup, "What is object-oriented programming?" *Software, IEEE*, vol. 5, no. 3, pp. 10–20, 1988.

[57] A. Vihavainen, M. Paksula, and M. Luukkainen, "Extreme apprenticeship method in teaching programming for beginners," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 93–98.

[58] B. E. Wampler, *The essence of object-oriented programming with Java and UML*. Addison-Wesley, 2002.

[59] H. Werner, "Comparative psychology of mental development." 1948.

[60] U. Wolz, H. H. Leitner, D. J. Malan, and J. Maloney, "Starting with scratch in CS 1," in *ACM SIGCSE Bulletin*, vol. 41, no. 1. ACM, 2009, pp. 2–3.