Developing high-fidelity mental models of programming concepts using manipulatives and interactive metaphors

Submitted in fulfilment of the requirements of the degree of

MASTER OF SCIENCE

of Rhodes University

Author: Matthew Funcke

Supervisor: Prof. Peter Wentworth

Grahamstown, South Africa

30/10/2014

Abstract

It is well established that both learning and teaching programming are difficult tasks. Difficulties often occur due to weak mental models and common misconceptions. This study proposes a method of teaching programming that both encourages high-fidelity mental models and attempts to minimise misconceptions in novice programmers, through the use of metaphors and manipulatives. The elements in ActionWorld with which the students interact are realizations of metaphors. By simple example, a variable has a metaphorical representation as a labelled box that can hold a value. The dissertation develops a set of metaphors which have several core requirements: metaphors should avoid causing misconceptions, they need to be high-fidelity so as to avoid failing when used with a new concept, students must be able to relate to them, and finally, they should be usable across multiple educational media.

The learning style that ActionWorld supports is one which requires active participation from the student - the system acts as a foundation upon which students are encouraged to build their mental models. This teaching style is achieved by placing the student in the role of code interpreter, the code they need to interpret will not advance until they have demonstrated its meaning via use of the aforementioned metaphors.

ActionWorld was developed using an iterative developmental process that consistently improved upon various aspects of the project through a continual evaluation-enhancement cycle.

The primary outputs of this project include a unified set of high-fidelity metaphors, a virtual-machine API for use in similar future projects, and two metaphor-testing games. All of the aforementioned deliverables were tested using multiple quality-evaluation criteria, the results of which were consistently positive. ActionWorld and its constituent components contribute to the wide assortment of methods one might use to teach novice programmers.

Acknowledgements

First and foremost I would like to thank my wife for her continual support while working on this project, especially when I felt like I simply could not keep going. I would also like to thank my supervisor, Prof. Peter Wentworth, for his valuable advice and guidance. Finally, I would like to acknowledge the financial and technical support of the NRF, Telkom, Tellabs, Stortech, Genband, Easttel, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Contents

1 Introduction **1**

1.1 Motivation 1

1.1.1 Objectives and Deliverables Preamble 3

1.2 Preliminary Definitions 4

1.3 Overview of Deliverables and Objectives 4

1.3.1 The Metaphor Set 5

1.3.2 The Demonstration-and-Validation Game Implementations 5

1.3.3 The Virtual Machine and its API 6

1.3.4 The Test Framework 7

1.3.5 Scope Delimitation 7

1.3.6 Methodology Used for Deliverable Creation 8

1.4 Limitations of the Study 8

1.5 Layout of the Dissertation 9

1.6 Summary 9

2 Related Work **11**

2.1 Education Theory and Learning Styles 11

2.1.1 Constructivism 11

2.1.2 Barriers of Traditional Learning 12

2.1.3 Abstraction of Complexity 13

2.1.4 Threshold Concepts and Cognitive Thresholds 15

2.1.5 Multi-Modality 17

2.2 Importance of Visualisation 18

2.3 Metaphor Background 20

2.3.1 System 1 - Gilligan's Analogies 22

2.3.2 System 2 - The Hackety Hack Metaphors 24

2.3.3 Potential Dangers when using Metaphors 25

2.4 Common Misunderstandings and Troublesome Concepts 27

2.5 Virtual Learning Environments. 28

2.6 Gilligan's System and Programming by Demonstration 29

2.6.1 An Overview 29

2.6.2 Key System Distinctions 31

2.6.3 String Type Implications 33

2.6.4 The Interface 33

2.6.5 The API and Gilligan's Virtual Machine 34

2.7 Summary 34

3 Methodology **36**

3.1 Defining Artefacts and Deliverables 36

3.2 Iterative Process 37

3.2.1 Metaphor Development Methodology 37

3.2.2 API Development Methodology 38

3.2.3 Game Development Methodology 39

3.2.4 Iterative Hallway Testing 40

3.2.5 Validation and Evaluation 41

3.3 Design Based Methodology 41

3.4 MVC Build 43

3.5 Functionality over Aesthetics 45

3.6 Summary 45

4 The Metaphor Set **46**

4.1 The Finished Metaphors 47

4.1.1 Values 47

4.1.2 Variables 47

4.1.3 Expressions, Arithmetic and Calculation. 48

4.1.4 Conditionals 50

4.1.5 Local Variables and the Current Stack Frame. 51

4.1.6 The Stack and Methods 52

4.1.7 Global Variables 56

4.1.8 Representing the Heap 57

4.1.9 Interacting with Objects and other Reference Types 60
4.1.10 Strings 62
4.2 Early Developmental Stages - how to Represent the Game World? 63
4.3 Pre-Optimisation Metaphors and Functionality 65
4.3.1 "What am I supposed to do next?" 66
4.3.2 The In-Game Hand's Function is Unclear 66
4.3.3 Never-Empty Variables 66
4.3.4 Expressions Overwhelming Students 67
4.3.5 Conditional Structures, and Boolean Value Disposal 68
4.3.6 Unclear Scope Division 68
4.3.7 The Method Mechanism 69
4.4 Alternatives and Potential Enhancements 69
4.4.1 Expression Enhancements 70
4.4.2 More Distinct Variable and Value Types 71
4.4.3 Non-Metaphor Heap Alterations 71
4.4.4 Potentially Representable, Non-Novice Concepts 72
4.4.5 Flow of Control Given to the Users 73
4.5 Summary 73

5 Implementation 75

5.1 Design Decisions 75
5.1.1 Register or Stack-Based Architecture 75
5.1.2 Level Generation and Storage 77

5.1.3 Text, 2D, and 3D Visuals 78
5.2 The API 78
5.2.1 Calculator Component Functionality 78
5.2.2 Final API State 80
5.3 The Game 83
5.3.1 Transitional Text-2D Stage 83
5.3.2 PIP Magnifier 83
5.3.3 Calculator State Machine 83
5.3.4 Bypassing the XNA Guide 86
5.4 Hallway Usability Testing 86
5.4.1 Devising Appropriate Test Levels 88
5.4.2 The Test Process 89
5.4.3 Important Changes 90
5.5 Summary 91

6 The Test Framework **93**

6.1 Mark Driven Quantitative Evaluation 94
6.2 General Qualitative Tests 95
6.2.1 Surveys 95
6.2.2 Focus Groups and Interviews 96
6.2.3 Evaluation through Experimental Extensions 96
6.3 Metaphor Evaluation 97
6.4 API Assessment 98

6.5 Quantitative Game Tests 99
6.5.1 Checklists 99
6.5.2 Rubrics 100
6.5.3 Heuristics 101
6.6 Qualitative Game Tests 101
6.7 One-on-One System Comparisons 102
6.8 Summary 102

7 Non-Technical Test Results 103

7.1 Checklist Evaluation - Kelleher and Pausch's Attribute Survey 103
7.1.1 General Feature Comparison 104
7.1.2 Specialist Category Comparison 105
7.1.3 Brief Comparison with Gilligan's System. 106
7.1.4 Statistical Comparison 106
7.2 Evaluation via Specialist Heuristics 107
7.2.1 Heuristic 1 - Matching Mental Models 108
7.2.2 Heuristic 2 - Navigational Fidelity 108
7.2.3 Heuristic 3 - Appropriate Learner Control 109
7.2.4 Heuristic 4 - Avoiding Tangential Complexity 110
7.2.5 Heuristic 5 - Meaningful Metaphors 110
7.2.6 Heuristic 6 - Personal Approaches 111
7.2.7 Heuristic 7 - Fast Feedback 112
7.2.8 Heuristic 8 - Custom Curricula 113

7.2.9 Summary of Heuristic Adherence . . . 113
7.3 A One-on-One Comparison of Systems . . . 113
7.3.1 Comparing against Gilligan's System . . . 113
7.4 Test Result Limitations . . . 115
7.5 Summary . . . 116

8 Technical Evaluation and Enhancement 117

8.1 A WPF Implementation and what it Demonstrates . . . 117
8.1.1 MVC meets WPF . . . 118
8.1.2 MVC Compliance . . . 119
8.1.3 XNA vs. WPF . . . 123
8.1.4 Not using XAML in WPF, a Design Choice . . . 128
8.2 Specific Technical Evaluation . . . 129
8.2.1 Completely New Functionality . . . 129
8.2.1.1 Adding the BoolEater . . . 129
8.2.1.2 Implementing an Undo Button . . . 132
8.2.2 The Code Interface Form, and the Modularity it Demonstrates. . . 134
8.3 Summary . . . 136

9 Future Work 137

9.1 Enhancing the Metaphors . . . 137
9.2 Enhancing the API . . . 138
9.3 Enhancing the Finished Games . . . 139
9.4 The Test Framework and More Rigorous Evaluation . . . 139

9.5 Enhancing the Level Editor . . . 140
9.6 Other Enhancements . . . 140
9.6.1 Reverse-Mode Code Generator for Sandbox Play . . . 141
9.7 Summary . . . 142

10 Conclusion **143**

10.1 Introduction . . . 143
10.2 Contributions of the Dissertation . . . 144
10.3 More Detailed Conclusions . . . 145
10.3.1 The Metaphors . . . 145
10.3.2 The API . . . 146
10.3.3 The Test Framework . . . 146
10.3.4 The Games . . . 147
10.4 In Closing . . . 147

A In-Game Screenshots **157**

B Hallway Test Levels **161**

B.1 Level 1 - Variables, Assignment and Operators . . . 161
B.2 Level 2 - Conditional Branches . . . 161
B.3 Level 3 - While Loops . . . 162
B.4 Level 4 - For Loops . . . 162
B.5 Level 5 - Introduction to Methods . . . 163
B.6 Level 6 - All in One . . . 163

C Unabridged Hallway Test Results **165**

C.1 Test Group One 165

C.2 Test Group Two 166

C.3 Test Group Three 168

C.4 Test Group Four 169

C.5 Test Group Five 169

D Kelleher and Pausch's Survey Data **171**

E Sample Survey **177**

E.1 Metaphor-Validity Survey 177

F Overlarge Sample Code **197**

List of Acronyms

API - Application programming interface.

I/O - Input/Output.

VLE - Virtual learning environment.

WPF - Windows Presentation Foundation.

GUI - Graphical user interface.

UI - User interface.

FSM - Finite state machine.

MVC - Model-view-controller.

DBR - Design Based Research.

FAT - File Allocation Table.

IL - Intermediate Language.

XML - Extensible Markup Language.

XAML - Extensible Application Markup Language.

ALU - Arithmetic Logic Unit.

List of Figures

2.1 The Kolb Learning Cycle 12

2.2 Two possible equations presented by DragonBox (Left), and their equivalent equations (Right). Users drag terms around, and add new ones in, in order to cancel out and simplify the equations. 14

2.3 Bostock's[1] static visualisation of a mergesort, where each line's angle represents the corresponding number's relative size, and each row represents the state of the list after a new pass. 20

2.4 Program patterns as sewing patterns[2] 21

2.5 The WPF implementation of the proposed system at the start of a level. Left: the code that the user needs to interpret. Right: the metaphor playground that users must interact with. The Image has been cropped and compressed somewhat in order to accommodate spatial limitations. . . 30

2.6 A screenshot of Gilligan's second prototype. Users drag objects from area C onto the clipboard in area A, the corresponding code is then inferred and displayed in area B. 31

3.1 A block-based flow diagram illustrating the iterative steps involved when using a DBR approach [3]. De Villiers and Harpur did not state that the demonstration phase is optional - this was included to demonstrate one of the steps that was occasionally bypassed during the course of this work. . . 42

3.2 The DBR flow diagram used during the design and creation of the metaphors. Notice that later stages in the development process feed into this diagram (or are fed by it) - this can include the game development stage, test framework application, or even the alteration of a related metaphor. . 43

3.3 An illustration of the MVC design the system used. View keeps track of components such as sprites (and their associated details), the Model keeps track of the state of the virtual machine, while Control manages user interaction and facilitates communication between the View and the Model. 44

4.1 A: An int value-notepad. B: User inputting a value. C: The user holding the final value. D: Alternative value-notepads. 48

4.2 A variable box called counter, containing the integer value 12345678. . . . 48

4.3 The steps required to evaluate an expression using the calculator. A: the calculator is blank and requires an expression. B: An expression has been entered and now requires value-substitutions for every variable. C: A value has been substituted (this requires a variable read first), this step is repeated 3 times. D: All values have been inserted and the user has asked for the answer (which was printed onto a piece of value-paper). 50

4.4 Two possible representations of the BoolEater conditionals mechanism. . . 51

4.5 Top: Barbed wire memory barrier. Bottom: Police tape memory barrier. . 51

4.6 Four snapshots of the current frame and local variables bookshelf - displayed on top of the frame conveyor belt, and inside the user space. Demonstrating the various states during the declaration of local variables ‘a’ to ‘g’. 53

4.7 Changes in state and visualisation for the stack and the frames that correspond to each method call. Read from left to right this shows the calling of each method, while read from right to left it demonstrates returning back down the stack (pushing and popping respectively). 53

4.8 Left: Metaphor one for method preparation, where the user is responsible for everything. Right: Metaphor two (signature-picking metaphor) for method preparation, where the user is responsible for arguments and calling but nothing more. The stages from top to bottom are: no method named, a method named but no parameters assigned or declared, one parameter (a) named and assigned, all the parameters named and assigned with the method ready for calling. 55

4.9 A simple fixed-size bookshelf that holds the global variables. It is directly accessible to the user. 56

4.10 The heap bookshelf with two objects in it. If they were array objects they would be of length 4 and 6 respectively. This heap can only contain 36 simple values, and thus demonstrates how a compressed representation would be desirable. 58

4.11 A compressed representation of the heap. A: an empty heap. B and C: one array present on the heap, with details above. D: highlighting the free space. E: three arrays on the heap, with ys being highlighted. G: a demonstration of what might happen after a garbage collection when ys is no longer being used. 59

4.12 A small section of the heap bookshelf showing potential representations of two non-array objects. 60

4.13 The memory manager robot (left) and the terminal used to communicate with him (right). These images of memory space communication metaphors were geared towards array communication, in the event of more complex object "Offset" might be replaced by "Field name". 61

5.1 A simplified class diagram of the API. Note how almost all interactions go through the WorldTracker class in some way. 81

5.2 A compressed screen shot demonstrating the magnification of the region pointed to by the hand (top right), the call stack (top left), the variables accessible in the current scope (bottom left), and the heap with two arrays present (top centre). 84

5.3 A visual representation of the logic used to make the calculator more intelligent, and thus more user friendly. 85

5.4 The XNA asynchronous Guide: an intrusive tool that obscures the state of the game. 87

5.5 Fundamental Concept Dependency Graph. Illustrates what concepts rely on other, simpler, ones. Arrows can be interpreted as saying "This one is depended on by that which it points to". Classes have not had their dependency defined, as it is in fact unclear - this is because almost everything done in C# is within a class, even if the user does not realise it. 88

7.1 Frequency of total features per system. The proposed system qualifies for 14 of Kelleher and Pausch's [4] features, while Gilligan's [5] Prototype 2 qualifies for 10. 107

8.1 The assorted classes and their lines of communication in the WPF implementation. 119

8.2 A pictographic representation of how the existing Model API simplifies View method calls. Where ReflectState is a general visual update method which belongs to View, and is passed as a delegate down to the FinishingAction associated with a specific sprite. 122

8.3 Left: non-intrusive built-in WPF menu. Right: intrusive custom XNA menu.127

A.1 The start of a sample level. The user has not done anything yet, and is being prompted to create a new integer variable 'x' (highlighted in yellow on the left). 157

A.2 The user has succesfully declared the integer 'x' which currently has no value. The user is currently creating an integer value via the int notepad. . 158

A.3 The user is now holding the newly created value, and is about to assign it to the integer variable 'x'. 158

A.4 By this stage the user has declared and assigned values for the variable 'y', and is about to feed the newly created Boolean value to the BoolEater in order to move into the body of the if(). 159

A.5 The user has complete all the steps required by the various conditional structures and loops (all very similar), and has prepared an empty stack frame for calling 'sampleMethod'. 159

A.6 The user has created and assigned a value to the 'num' parameter, and is preparing to perform the method call by clicking the call arrow. 160

A.7 The user has entered 'sampleMethod', and has created a value based on the expression 'num/2'. They are preparing to return to the previous stack frame by clicking the return arrow. Notice the cut-off stack frame near the top of the image - this is the frame of the method they will be returning to. 160

D.1 The first half of Kelleher and Pausch's [4] educational software categorisation taxonomy. The most suitable category for the proposed system is shown - next to Prototype 2 and ToonTalk. 172

D.2 The second half of Kelleher and Pausch's [4] educational software categorisation taxonomy. 173

D.3 The first page of Kelleher and Pausch's [4] attribute frequency table. . . . 174

D.4 The second page of Kelleher and Pausch's [4] attribute frequency table. The proposed system has been inserted here, and is highlighted. 175

D.5 The final page of Kelleher and Pausch's [4] attribute frequency table. . . . 176

Chapter 1

Introduction

1.1 Motivation

> "It was found that students with viable mental models performed significantly better in the course examination and programming tasks than those with non-viable mental models."
>
> Ma, Ferguson, Roper, and Wood [6], on the subject of teaching novice programmers.

This piece of information alone is enough to warrant investigation into what a viable mental model is, how they are developed normally, and how they can be fostered. Ma et al. [6] go on to say that "students must be supported to create new viable models", which is the issue that this work aims to address. In short, this work proposes a means to help build high-fidelity mental models amongst novice programmers.

There is no lack of evidence to support the statement that both teaching and learning to program are difficult. No single paper demonstrates this quite so well as that by Kelleher and Pausch [4]: over a 40 year period, more than 80 different fully functional systems aimed at teaching programming have been published, with system releases becoming more common from year to year. If one were to extrapolate this data, and assume that for every game or system included in Kelleher and Pausch's survey at least one other was not (either due to completion issues, language barriers or simply because the authors were unaware of the program), then by 2015 there should be *at least* 200 different systems all attempting to make programming easier (an extrapolation based on these assumptions

should be considered nothing more than an illustrative example). This abundance of different approaches and implementations is indicative of the fact that no one has been able to nail down the 'right' way to teach programming.

Most of the educational programming environments available follow a similar fundamental formula: present users with a problem, ask them for code to solve the problem, and compare the result of their program to the desired outcome. *Some* environments and languages that work this way (not including any of those surveyed by Kelleher and Pausch) include: Alice [7, 8], TurtleAcademy [9], Lightbot [10], RU-Bot [11], Core War [12] (an interesting way to visualise and learn simplified assembler-type programming), Scratch [13, 14], Snap! [15] (an extended version of Scratch),and Hackety Hack [16]. What seems to be missing from the assortment of teaching aids is one in which the user is required to explain the meaning behind each line of an existing piece of code. This is the niche which the proposed work aims to address.

It is widely recognized that one conceptual difficulty for many novices is that program execution is not concrete or tangible, and requires considerable abstraction skills on the part of the learner [17]. To address this, a number of authors agree that one of the best ways to teach programming is to always show the user the state of their program and the underlying data [1, 18, 19, 20]. Considering how important visualisation appears to be, it was decided that it be combined with the first goal of creating strong mental models. By combining the two ideas (programming through demonstration, and the importance of visualisation), users would have to explain the code they are given through a relatable visual medium.

A study performed by Lister et al. [21] demonstrates how, regardless of programming language, there is a strong relationship between students' abilities to trace, to understand, and to write code. This system attempts to strengthen tracing and understanding of code with the ultimate goal of improving code-writing skills.

There is also a great deal of literature on the topic of conceptual hurdles or cognitive thresholds (also known as troublesome concepts) which beginner programmers struggle to overcome [22, 23, 24, 25, 26, 27]. In some situations the conceptual models that a student builds to overcome such hurdles have to later be modified or completely replaced because the model they came up with was flawed. If students cannot adapt their weak mental models, they might simply give up on programming altogether. Explicit attention to building consistent and strong mental models may therefore have the potential to reduce attrition rates in programming courses.

Many authors address the issue of where students seem to struggle, however there seems to be no true consensus: Bayman and Mayer [22] show that more than 50% of self-taught BASIC programmers have trouble with simple ReadLines, while Götschi et al. [23] show that more than 50% of first year students do not have viable mental models for recursion (two concepts that are not at all similar regarding complexity). The wide range of troubles that novices seem to have could be attributed to poor fundamental mental models of what is actually happening.

The proposed system tries to address the various concerns by utilising information from all the areas of research discussed above: this research aims to support the student's learning through the use of visualisable metaphors that show the state of a program as they actively manipulate the environment to reflect the semantics of the associated code.

According to Clark [28], there is no one right approach to science or science education, but rather several alternative valid methods - with that in mind, this dissertation attempts to add to the pool of alternative methodologies.

1.1.1 Objectives and Deliverables Preamble

This sub-section serves as a pre-amble to the remainder of this chapter's contents. This was done in order to give a broad overview of the various topics, before delving into more detail. There where multiple objectives and deliverables established during the coarse of this dissertation, and one might summarise this extensive list as follows:

- In order to help learners overcome barriers to learning caused by difficulty with abstraction, it was asked whether a more visual teaching technique would be beneficial.
- This teaching technique needed a set of high-fidelity, easy to understand, interactive visual metaphors to be created. This then became one of the primary project deliverable.
- The aforementioned metaphors needed to be demonstrated and tested in some way. As the metaphors needed to be interactive, it was proposed that a game be created based on them, which could then be used for testing and development.
- After beginning work on the game, certain framework limitations began to show up - in order to overcome these, a new objective was added whereby an API for the game would be created (thus making it more framework independent).

- In the interest of rigour, the creation of a separate test framework was also included as a goal - this framework was meant to try and evaluate the metaphors (and the proposed game), in as objective a fashion as possible.

1.2 Preliminary Definitions

For the sake of clarity it is necessary to define certain terms such as 'manipulative' and 'metaphor', in the context of this project.

ActionWorld refers to the proposed system as a whole (rather than any individual component). This name was derived from the fact that students have to take actions in a virtual machine world, and the virtual machine that governs the state of this world makes heavy use of Actions (void Delegates).

The terms 'metaphor' and 'analogy' are used interchangeably in this text, and mean any sort of representation of a concept that "helps students join the dots" by decreasing abstraction, generally by equating the concept in question to something in the real world. In mathematics a 'manipulative' is any physical object that encourages understanding of a particular concept through manipulation [29]. One example would be a set of wooden pieces that could be rearranged to demonstrate Pythagoras' theorem, or the equivalence of the areas of two rectangles. A virtual manipulative is the non-physical (usually digital) equivalent, for example, a program that lets the user adjust the coefficients of an equation and view its graph or interpretation.

A manipulative, in the context of this work, refers to any interactive version of the metaphors mentioned above (whether physical or virtual). For example, one theoretical metaphor of a variable could be a special box, and when implemented in a computer game as a clickable sprite, one has a manipulative (the key differentiator is that one is theoretical while the other is an interactive implementation).

1.3 Overview of Deliverables and Objectives

As explained in the preamble, this work aims to create several key deliverables, all of which contribute to the larger goal of providing an alternative method of helping novice programmers form stronger mental models. As many of the deliverables are referred to

before they are fully elaborated upon, it is necessary to add context regarding their form and function. This section provides a brief overview of the deliverables, without going into the details of various design decisions, technical aspects, or other reasoning.

1.3.1 The Metaphor Set

The metaphor set is an attempt to create a platform independent collection of high-fidelity analogies that can be used to help novice programmers form strong mental models, and overcome difficult conceptual hurdles. A high-fidelity metaphor needs to meet certain criteria: it must give a fair representation of what it is trying to describe, but more importantly it needs to remain valid when used in conjunction with other concepts. For example, if one tried to explain variables using an analogy, and then later tried to liken parameters to specialist variables, the original analogy needs to remain valid. Use of high-fidelity metaphors is critical to this work, as any mental models built upon a framework of weak metaphors are likely to fail as soon as the metaphors they are built around begin to break down.

The metaphors in question have additional requirements: they need to be usable on different mediums (for example: textbooks, games, and class rooms), they need to accurately portray machine state, and finally they need to be relatable (in that students must be able to relate to them). Creating such a set of metaphors requires several other deliverables for testing and demonstration purposes, which are elaborated on in the remainder of this section.

1.3.2 The Demonstration-and-Validation Game Implementations

As a means of demonstrating medium-independence, usability, and fidelity (among other things), the creation of a 'test game' built around the metaphors was proposed (where the metaphors essentially visualise the state of program execution). A typical game-level is comprised of two major parts: the code that the users are provided with, and the series of metaphor interactions that the users need to undertake in order to demonstrate a detailed understanding of every line of the given code.

For example, the code presented to users for a *very* simple level might be:

```
int x = 3 + 4;
```

The user's interactions would need to demonstrate an understanding of three separate ideas: declaring the variable `x`, evaluating the expression `3 + 4`, and assigning the result to the newly created variable. Each one of these actions would entail a separate interaction with the environment.

The final versions of these test games also include a sandbox mode, where no code is provided, and users can simply interact with the metaphors as they like (it was believed this would encourage experimentation). Regardless of the mode being used, the state of execution is always represented through the highly visual metaphors.

This student-as-interpreter model allows one to check whether or not a given student can see a relationship between the code they are given, and the metaphors that they need to interact with. A series of successful interactions is probably a positive indicator that the metaphors meet the criteria described in Section 1.3.1 - while a student not being able to explain a level may indicate that they do not understand the code, or that the metaphors are not being understood, or both.

1.3.3 The Virtual Machine and its API

Any game implementation, such as those explained above, would require an underlying execution engine, or virtual machine, to keep track of the user's progress through a level (in addition to other state tracking functionality). It seemed reasonable to create a separate component to serve this role. Its specifications and usage are presented as an application programming interface (API). An API such as this would allow others to easily create alternative implementations of the demo game, to test their own (or this work's) metaphors, or for any other purpose which might require a code execution engine.

The functionality of the execution engine can be summarised as follows: it is sent a single operation code, the operation is then performed inside the engine (without any further input from the developer or player), and changes to the virtual machine's state are then reflected on the front-end which called the engine in the first place. This whole process takes place with a single method call, making the API particularly easy to use from a developers perspective.

The API that was created for this work is loosely coupled to the front-end games which are built on it. One should also note that the novice programmers, whom ActionWorld is trying to help, never see or interact with the engine directly (which is fairly technical):

their experience is limited to the point and click visual interactions with the metaphors that they are presented with.

The execution engine is not strictly speaking a *program* execution engine, and can rather be thought of as an *instruction* execution engine. The difference between these two terms is that a program execution engine is optimized for speed of execution, where single-step debugging is an additional (often difficult to implement) feature. While an instruction execution engine is built from the ground up as a single-step debugger. This shift in focus bypasses the need to set up complex execution interrupts, callbacks, hooks, and monitoring conditions.

1.3.4 The Test Framework

Each of the above deliverables is a multi-faceted component of a larger whole. Because of the scale and complexity of evaluating such a system, it was deemed necessary to create a clearly defined test framework with which to evaluate each deliverable. The framework that was proposed is referred as a ***constituent evaluation*** framework, and was created in such a way as to facilitate testing of components independently of one another. As this framework is only really referred to again in Chapter 6, this synopsis should suffice for now.

1.3.5 Scope Delimitation

Four deliverables have been described so far: the metaphor set, a debugging engine and its API, a game which uses them both, and a test framework. However, there are an assortment of other aspects one needs to consider when creating a set of deliverables such as these. For example, there is a need for boundaries which delimit where the metaphors no longer need to hold (such as how novices are unlikely to gain any extra benefit from an analogy that can go so far as to explain multi-threaded programs). One also needs to decide which concepts are most foundational, so as to ensure they make up the core of the system, while concepts that confuse novices the most need to be identified so as to try to represent them most clearly.

The following concepts and structures were deemed to be within the scope of teaching novices: values, local variables, expressions, conditionals, methods, global variables, simple heap objects, and strings. The reasons behind the content of this list are covered later, as they are fairly specific, and thus do not belong in an overview such as this.

1.3.6 Methodology Used for Deliverable Creation

While not strictly speaking a deliverable in and of itself, the methodology used has a significant impact on the quality of the final deliverables (hence this overview):

Throughout this dissertation a design-based research paradigm was adhered to - this paradigm makes use of an iterative technique where the final deliverables are initially explained in very broad terms, and where multiple iterations make gradual improvements on prior versions[30]. This technique allows one to have a final goal, and achieve said goal, without necessarily knowing the best path to follow in order to achieve it from the start. An example of where design-based research was particularly useful was in improving the understandability of the metaphors created - the first iterations were not easy to understand at all, while the final versions were clean and simple.

1.4 Limitations of the Study

One of the primary limitation of this work is due to its scale - due to the number of deliverables created, and the amount of work that went into each one, it is impossible to perform an object analysis of the system as a whole. Chapters 6, 7, and 8 attempt to set up and utilise a framework one can use to evaluate the system (or individual components) - from both a non-technical and technical perspective, respectively - however the application of such an evaluation mechanism on one's own work cannot be done without a certain amount of bias entering into the results. Section 7.4 goes into more detail regarding specific test limitations.

The second key limitation of this dissertation is that the students involved are never actually asked to write code - either before or after being exposed to the game. This makes conclusions regarding the effect of this teaching technique on individuals hard (if not impossible) to measure.

These limitations are overcome to a certain degree by the fact that this work is primarily attempting to lay a technical foundation on which to perform future related work, rather than focusing on the effects such a system actually has.

1.5 Layout of the Dissertation

Rather than adhering to a strictly chronological organisation, this paper is laid out according to a dependence hierarchy. The chapters are organised as follows:

Chapter 2 explains and discusses work from several fields that strongly relate to this project, including education theory, metaphors in computer science, and virtual learning environments, among other things.

Chapter 3 defines the developmental and research methodologies adhered to during the course of this work.

Chapter 4 provides a detailed description of the proposed metaphor set for teaching novices, as well as a summary of the developmental stages it went through.

Chapter 5 elaborates on technical details regarding how various system components were built, the rationale behind certain design choices, and a summary of the system usability tests.

Chapter 6 explains the various ways in which one might evaluate a system such as ours, and proposes a test framework one might adhere to (along with several sample tests).

Chapter 7 applies the less technical of the aforementioned tests to the proposed system, and discusses the results.

Chapter 8 performs a more technical evaluation of the system, and discusses the results.

Chapter 9 details several ways in which the proposed system, and its various components, could be enhanced in the future.

Chapter 10 reviews material covered in the paper, and presents the conclusions drawn from the work.

1.6 Summary

This section laid out the motivation behind this work as having multiple sources, including: the need to explicitly build strong viable mental models, the benefits of visualising data, and an alternative student-as-interpreter perspective for teaching programs.

This paper's deliverables hinge on the creation of a set of metaphors, which in turn can be broken up into developmental and quality evaluation elements:

- Identification of troublesome concepts.
- A brief survey of existing metaphors.
- A brief survey and comparison of existing games that teach programming.
- The creation of a set of high fidelity metaphors, based on the three previous points.
- Development of a modular execution and debugging engine and its API for use in the test game.
- An interactive implementation of the metaphors (the game itself) for execution and metaphor validation purposes.
- Creation of a test framework for use in validating both the metaphors and the illustrative game.

Most programming courses and approaches do not explicitly teach specific mental models. They teach programming, and leave it up to the students to construct their own mental models. Given the evidence that these are often poor or non-viable, this work might be considered a first step towards a teaching approach that makes the mental models more explicit.

Considering the aforementioned deliverables and objectives, this work attempts to contribute to the body of domain knowledge in multiple ways. These range from the proposal of a new programming teaching technique, and metaphors to go with it - to the creation of an adaptable program and API to be used in applying and testing the proposed technique. Other, more technical, contributions are also present, and a complete list of these can be found in Section 10.2.

Chapter 2

Related Work

In the previous chapter, mention was made of the benefits of visualisation, the poor mental models that novices have, and the sheer volume of systems that attempt to teach programming. This chapter elaborates on topics such as these in order to give a more solid background for the work that will be based on them. This chapter gradually moves from more broadly applicable topics, to more specialist work. For example, this chapter begins with a description of some aspects of education theory, and concludes with a summary of a programming game by Gilligan [5], which works via manipulation of in-game analogies (Gilligan's work runs in parallel to, and occasionally overlaps with, this research).

2.1 Education Theory and Learning Styles

2.1.1 Constructivism

Constructivism asserts that all knowledge created by a learner is built on top of prior knowledge (also called schemas), meaning that everything one learns is based off of what one already knows. Constructivism also states that engagement on the part of students is always more beneficial to the learning process than passively acquiring new knowledge (such as through a lecture). And finally, constructivism states that individuals actively construct their personal representations of a shared reality. Most applied constructivist methodologies attempt to guide students to knowledge creation.

Like most theories, constructivism does have its critics [31, 32] (for example, some critics argue that offering minimal guidance to students while also encouraging experimentation,

is less effective than simply offering strong guidance). Despite this, the proposed system builds on constructivist theory by assuming that a student who can liken a new concept to some pre-existing schema is likely to assimilate the new concept more quickly and accurately than one who tries to learn via more traditional abstract means. To alleviate the concerns of constructivism critics, one should note that while the proposed system shares several attributes with constructivism, it also incorporates non-constructivist ideas: for example, it tries to direct learners through gradually more complex levels (occasionally going so far as to simply tell the user what to do, in an objectivist manner[1]) and has an accompanying hint system.

To clarify, objectivism states that all knowledge takes the form of unchangeable non-subjective facts that everyone must perceive in the same way, while constructivism says that all learning is subjective and that any newly learnt concepts will rely (and build) on an individuals prior experiences [33].

Kolb (an advocate of constructivism), has proposed a learning cycle which makes use of what he calls "active experimentation" [34]. This paper attempts to build on the active experimentation explained above. Figure 2.1 shows the Kolb Learning Cycle.

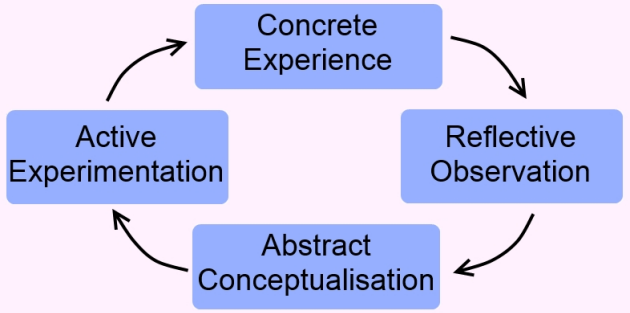


Figure 2.1: The Kolb Learning Cycle

2.1.2 Barriers of Traditional Learning

According to Coyle [35], a person develops and improves their ability to perform a task by adding myelin to circuits in the brain (also referred to as "construction and assembly of component proteins"). This process is most pronounced when people learn from their mistakes (rather than, say, learning by rote). Because this process is best achieved via practice and error-centric feedback, a game that is immediately able to tell the user that

[1] Objectivism and Constructivism are essentially opposing philosophies

they made a mistake - as well as *why* it is a mistake - should in theory greatly increase the rate at which a person is able to learn the subject matter (in this case programming).

Observations made by Carl Wieman [36] demonstrate how people who do not engage with a given topic are unlikely to retain information about it - he found that only around 10% of lecture content is retained. This statistic alone suggests that most students are unlikely to be able to learn to program if they do not engage with real code (and instead simply, say, attend lectures). Another interesting idea derived from this is that a programming environment that offers immediate feedback as well as forced engagement will likely improve learning.

There are examples of fast feedback systems, such as the observable agent behaviour (or misbehaviour) in systems such as the Logo [9] and Python [37] turtles, or the Snap! [15] multi-form sprites. ActionWorld acts as a fast feedback system in a similar way to the aforementioned systems.

Another interesting idea that can be taken from Wieman's observations is that "students do not fail or succeed on the basis of what facts they can retain, but because of the structure and organization they can bring to the content." [38] which implies that if one can give students the necessary structures and organizational systems, or help them build their own, their ability to learn programming (or anything else) should benefit . Therefore a sub-goal of the proposed system would be to help students build such a scaffold (in this context, a scaffold is a temporary structure put in place in order to help students reach a particular level of understanding, before later being removed).

2.1.3 Abstraction of Complexity

Teaching a subject without using field-specific nomenclature or syntax can be difficult, however when done correctly it can have numerous positive effects on learners. Consider a student who is taught to program using only pseudocode. If that student is able to write all the appropriate logic in pseudocode for a particular problem, learning any specific language would be greatly simplified as their fundamental understanding does not rely on any language specific constraints. This process is often called abstraction of complexity, and it is the process of hiding extraneous details that might otherwise overwhelm a novice.

A prime example of this teaching style can be found in the field of mathematics: Huynh and Marchal [39] created DragonBox, a game that is able to teach the fundamentals of

algebra to children of ages five and up. According to Shapiro [40] "of those students who played at least 1 hour, 83.8% achieved mastery", these numbers alone highlight just how powerful this sort of technique can be.

DragonBox uses metaphors to (initially) simplify and sidestep the syntax of the maths, and gradually replaces the concrete metaphors with more abstract equations as students advance. While the proposed system does not gradually replace metaphors, it still follows a similar pattern of metaphor manipulation. Figure 2.2 shows what a student would see in DragonBox and how it might compare to an actual equation.

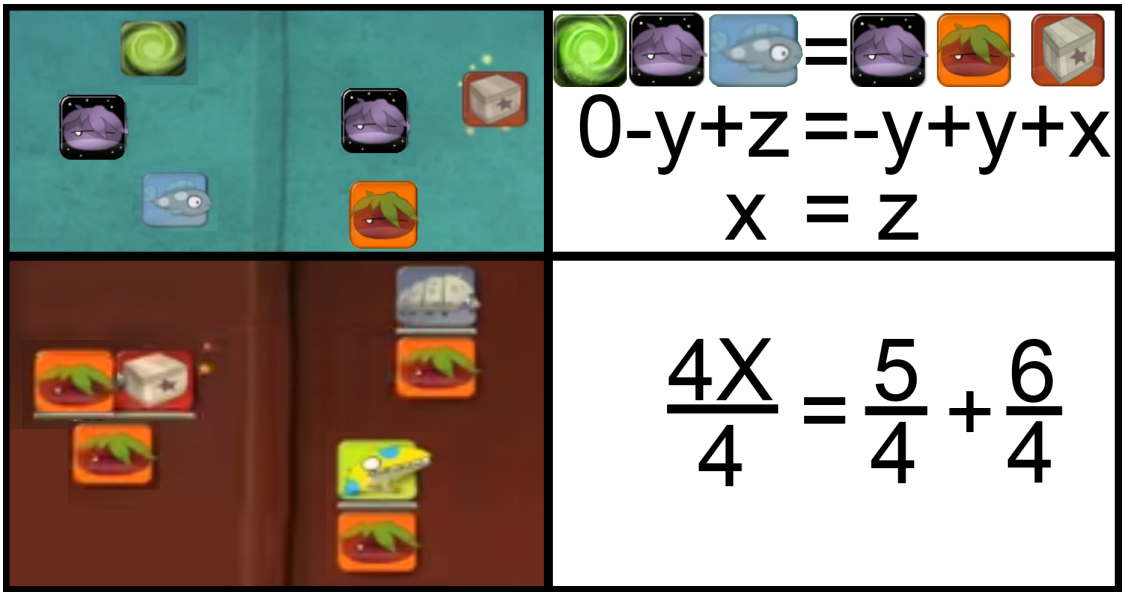


Figure 2.2: Two possible equations presented by DragonBox (Left), and their equivalent equations (Right). Users drag terms around, and add new ones in, in order to cancel out and simplify the equations.

Snap! [15] and Scratch [13, 14] (among others) employ a similar abstraction technique by removing syntax and allowing students to drag and drop code chunks. All of the aforementioned systems give visual (often immediate) feedback to students, which might contribute to their success, and relates to what was explained in Section 2.1.2 regarding error-centric feedback. This visual feedback can be both positive as well as error-focused for example, as soon as the sprites in these systems do something, the users know whether or not their logic was correct, and where it went wrong (if it did).

2.1.4 Threshold Concepts and Cognitive Thresholds

Threshold concepts can be summarised as any concept that a student is likely to struggle with, but which will result in them acquiring a deeper understanding of a subject upon assimilation of the concept. Flanagan [41] demonstrates just how prolific the theory of threshold concepts has become, with applications in fields ranging from journalism to computer science, and engineering to geography. Such widespread acceptance and application lends a great deal of credence to it.

Meyer and Land [42] explain that threshold concepts need to meet five criteria - they need to be: transformative (in the sense that one looks at things differently), irreversible (in the sense that one cannot "undo" the new insight), integrative, troublesome (something that is "conceptually difficult or counter-intuitive"), and often (but not always) bounded. Bounding can refer to the notion of a discipline's boundaries, or it can also mean how the barriers between one threshold concept and the next should not overlap (this second meaning is less commonly applied).

Meyer and Land provide a simpler way to understand threshold concepts by likening them to portals: it may be difficult to pass through them, but once you do you end up in a whole new area (from a cognitive perspective), and there's no going back once you pass through. They pose an interesting problem from an educator's perspective, because once they have been overcome it can be difficult to remember how one thought before the associated cognitive transformation, which makes explaining them to an 'uninitiated' student difficult [43].

Rountree and Rountree [27], who focus more on threshold concepts in computer science, describe concepts in general as being core concepts, threshold concepts, or both. They describe a threshold concept as being an idea or way of thinking that is key to not only understanding a subject, but to beginning to think like a practitioner of the subject. For example, simply understanding how a computer scientist thinks is not the same as being able to think like a computer scientist (this would be almost akin to knowing what logic is, as opposed to being able to apply it).

Zander et al. [44] performed an interview of 36 different computer science educators, and they were unable to reach a consensus on what the key threshold concepts are when teaching programming, although several came up more frequently than others (such as recursion, pointers, and the differences - when present - between classes, objects, and instances). Boustedt et al. [45] performed a study of 33 programming concepts that

could be potential threshold concepts. An amalgamation of the most popular threshold concepts in programming follows, drawing from work by Zander et al., McCartney and Sanders [46], and Boustedt et al.:

- Object orientation
- Abstraction
- Levels of abstraction
- Procedural abstraction
- Pointers *
- Differences between classes, objects and instances *
- Polymorphism
- Recursion and induction *

Marked concepts can be represented in the proposed system. An example of how one might represent one of these in this system follows: the difference between value and reference types (i.e. pointers) can be made less troublesome by allowing students to visualise the difference. One way that this can be done is to show the state of the computer's memory, and allow students to then experiment with value and reference variables in order to better understand their underlying differences.

Of Meyer and Land's [42] five criteria for a threshold concept, the proposed system aims to be able to help students with at least three of them:

- Assistance with the transformative requirement by giving them an alternative perspective to see things from (which is almost exactly what the transformation is meant to do in the first place).
- Irreversibility will hopefully occur when a student assimilates one of the metaphors into their own mental models (whether pre-existing or not).
- And finally, by carefully crafting the set of analogies to cooperate with one another (by design rather than brute force or coincidence), the integrative criterion will hopefully be addressed.

This system attempts to address the criterion of troublesome knowledge by giving it a context in which it can be understood, as well as a clear representation - however, this does not mean that the system can address all troublesome concepts. This paper does not address the final 'requirement' of threshold concepts (that of bounding) primarily because it is a theoretical division between fields, but also because it is not a hard and fast requirement for threshold concepts. Even if one is not an advocate of the Threshold Concept model, the aforementioned reasoning still applies to problems that are transformative/troublesome.

More traditional education systems favour an objectives-based approach, however Rountree and Rountree [27] explain that threshold concepts are an alternative way of perceiving education. The proposed system can be used by educators to undertake computer science education using either model.

Thomas et al. [47] have proposed a complementary practical-oriented idea, to go along with the more theory-oriented threshold concepts: the idea of threshold skills. They state that mastery of a subject requires both a theoretical understanding and the ability to apply that understanding. For example, it is all well and good to *understand* recursion, but being able to write a recursive method is a skill (and in context it is a threshold skill). One of the primary requirements of a threshold skill (that it does not already share with its associated threshold concept) is that it requires practice. There is more to say about the differences between concepts and skills, however for the moment it is enough to say that the system seems to almost bridge the two. The only unique requirement of threshold skills is addressed by offering students a platform to practice on.

2.1.5 Multi-Modality

Glasser [48] and a number of other notable academics are often incorrectly quoted as giving percentages relating the amount of information people retain based on how they learn certain material, for example, "10% of read material is remembered while 90% of taught material is retained". It is believed that an unsubstantiated article by Treichler [49] in the magazine *Film and Audio-Visual Communication* is the origin of these unsubstantiated percentages [48]. While the validity of the quotes may be in question, the underlying theory behind them is not: multi-modality encourages greater retention. In other words, the more ways individuals learn a particular topic, the more likely those individuals are to understand and remember it.

Multi-modality is relevant to this work because while the system does not offer a multitude of different representations for what it is trying to teach, it is not monomodal as both images and text are used to encourage students to engage. Furthermore, multimodal learning does not have be done using just one system, which means that if this system were used in conjunction with more traditional teaching techniques, and conventional programming exercises, it would result in a more varied (and thus hopefully more effective) learning experience.

2.2 Importance of Visualisation

Victor [18] makes several compelling arguments about what constitutes a good programming system, and concludes by referencing Tufte's [20] primary rule for understandable environments. In short, this rule says that you must always show the data. The longer (more amusing) explanation by Victor goes like this:

> "If you are serious about creating a programming environment for learning, the number one thing you can do - more important than live coding or adjustable constants, more important than narrated lessons or discussion forums, more important than badges or points or ultra-points or anything else - is to show the data."

Before concluding with Tufte's rule, Victor also explains that there are two major features that any good programming system needs. The first of these is that the program needs to "encourage powerful ways of thinking" in its users; this point is subject to individual opinions about what a powerful way of thinking is - for example, is it better to be able to think both iteratively and recursively, or is being able to abstract away extraneous detail more 'powerful'...what about being able to see how the individual components of a program come together to make a whole? With regard to this point one could also say that having high-fidelity mental models of various programming concepts is a powerful mental tool, and therefore that this work is in fact attempting to support this first point.

According to Victor the second goal of any good programming system should be to let users see the execution of their code, as an aid to understanding it. This point is less ambiguous than the first and relates directly to Tufte's rule about showing the data. While the proposed system does not show users the execution of their code, it is still visualising

program execution one step at a time - the major difference is that users are asked to mentally perform the execution for themselves (and then demonstrate their understanding of the resulting program state by manipulating the metaphors). With all of this in mind one might restate Victor and Tufte's rules as "any good system for teaching programming needs a step-by-step debugger and inspector that shows the data".

Unfortunately, along with several other suggestions by Victor, most of the teaching programs examined during the course of this paper ignore several simple yet essential paradigms that could arguably aid in the learning process. Most importantly only a minority show the user the state of the data that they are trying to manipulate via code. This need for data visualisation may stem from the dynamic nature of program data (as opposed to the static nature of data in, say, mathematics), which changes during execution and is therefore harder to track.

However, one should carefully consider the long term implications of constant visualisation. The potential danger is that while one solves a short-term problem (that learners cannot visualize the data or imagine what is going on), some would argue that this approach is sidestepping the real issue: how is one to build good abstraction skills if everything is always visualized or explicit?

Bostock [1] demonstrates the benefits of algorithm visualisation, and while his examples are for very specific algorithms and are generally non-interactive, such visualisations do an excellent job of illustrating macroscopic meanings *and* difficult-to-spot output differences. In short, while Bostock's focus is not on visualising general code, it would be difficult to dispute the usefulness of his proposed visualisations. Figure 2.3 shows an example of one of his static visualisations (most are either animated or too large to include here, hence the use of his easiest-to-understand static example).

One can see that the aforementioned benefits extend beyond just code visualisation, by considering a simple image, juxtaposed against the image's serialized representation: one can see that the image (which abstracts away the detail) is the less overwhelming representation (despite actually having exactly the same amount of information). It was proposed that this concept could be applied to how one might represent the potentially overwhelming details of what happens behind the scenes during code execution. A debugger is the ultimate means of representation in terms of both accuracy and fidelity, and allows users to inspect and accurately understand the low-level execution details. But that level of detail can also be overwhelming (leaning more towards the serialized representation, than the holistic one), therefore approach taken was to combine this idea with

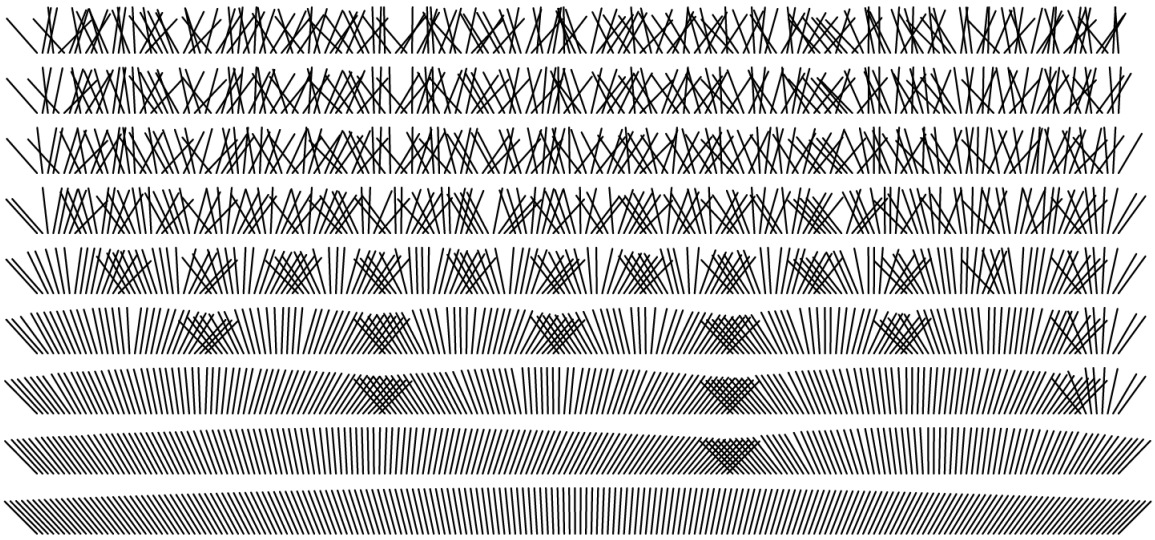


Figure 2.3: Bostock's[1] static visualisation of a mergesort, where each line's angle represents the corresponding number's relative size, and each row represents the state of the list after a new pass.

the importance of visualisation (as explained by Victor [18], Tufte [20] and Bostock [1]), and the value of high-fidelity mental models.

2.3 Metaphor Background

Of the 80 programs and games that Kelleher and Pausch [4] surveyed, 20 used physical metaphors to assist with understanding in some way. Of those 20, only five were able to incorporate all (or almost all) of the more fundamental programming constructs (conditionals, loops, methods etc.), and not one was able to assist with code that was both procedural and object oriented. While this does highlight something of a niche in the area of educational programming games, it also shows that of the established metaphors available, there does not appear to be any unified sets that work across the board.

The breadth and diversity of metaphors for teaching programming is immense, this section serves to illustrate just how varied metaphors can be by examining those used by two very different programming systems, as well as explain why caution should be observed when teaching via analogy. The two systems in question vary both in teaching and programming style, where Gilligan's [5] system is procedural (Pascal), and Hackety Hack [16] is object oriented (as is Ruby).

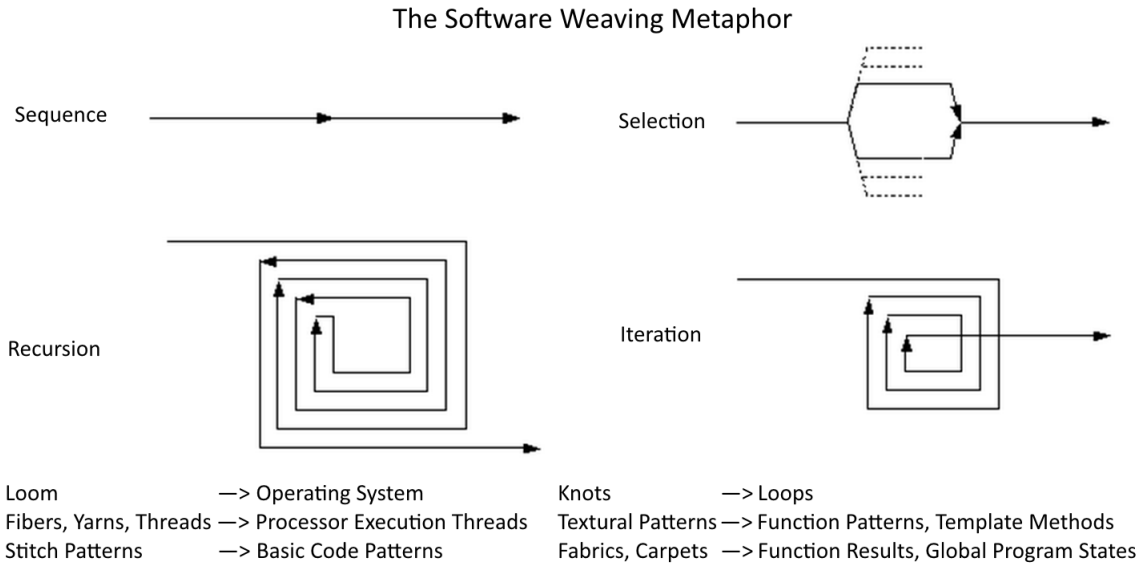


Figure 2.4: Program patterns as sewing patterns[2]

As a demonstration of this diversity, here are three examples of very different stand-alone metaphors:

1. The first and only mention of analogies in TurtleAcademy [9] is made in their lesson on loops, likening nested loops to bird's nests within larger bird's nests. They compare a loop inside another loop to a bird's nest inside of a larger bird's nest ("Command[mama loop[baby loop]]"). Considering that this is the only place they use metaphors for explanation, it may well have been done simply to explain the term 'nested', rather than to clarify the students' mental models. Regardless, this is an interesting - if uninformative analogy - in that it can be applied to any nestable structures.

2. Jiménez-Peris [24] propose that memory be explained by likening it to a locker room: each locker represents a cell in memory with its number and matching key taking on the roles of memory address and pointer. They explain how this analogy is robust enough to explain why copying a pointer is not the same as copying its value, memory leaks, uninitiated pointers, and more.

3. Figure 2.4 is a good illustration of just how far reaching (and often bizarre) some analogies for programming are - this particular one likens different programming concepts to sewing patterns [2].

2.3.1 System 1 - Gilligan's Analogies

Of all the systems surveyed during the course of this work, Gilligan's [5] system is the most closely related. Sections 2.6 and 7.3.1 elaborate more on his system in general, for now the focus is on describing some of his system-specific analogies. Gilligan has approximately 18 different *implemented*[2] metaphors in his system, with a one to one relationship between metaphors and the concepts that they describe. This is far too many to detail in their entirety (Gilligan's already-succinct description covers 16 pages), and so in the interest of brevity this summary only includes a brief explanation of some of the more relevant analogies (a good number of those not included are conditional-structure specific variants of the same basic analogy).

Gilligan's analogies place the user in control of an office clerk (who represents the computer), and gives the clerk a worksheet (which represents a method) to work through. The clerk follows the instructions to the letter until either the end of the worksheet (at which point it is discarded, and any previously incomplete worksheets are returned to) or until reaching an instruction that requires more details on how to complete it (at which point another worksheet is printed and placed over the current one.) This broad description already illustrates an analogy for methods as stackable and disposable worksheets, but also presents a *potential* problem: the clerk does not change rooms (or make any other obvious scope changes) upon starting or finishing a worksheet, thus the end user might be mislead into thinking that the scope had in fact not changed (along with all the ramifications of such a misunderstanding).

In his system different expression types (string, arithmetic, and boolean) each have their own calculators, this arrangements has advantages and disadvantages: the biggest advantage is that users get a strong sense of variable type distinction, the biggest disadvantage is that expressions comprised of more than a single operation type become complicated, and maybe even impossible (for example: "x + 1 >= y/2" would require at least two calculators). Another risk that this multi-calculator approach poses is that students might misunderstand and think that no expression can use more than one form of arithmetic. ActionWorld also had to take into consideration the different benefits and drawbacks of alternative calculator implementations - and concluded that offering two alternatives, instead of forcing one, would be beneficial (more on this in Section 4.1.3).

As ActionWorld purposely avoids in-code input/output (I/O) operations (such as the presence of console operations *in the code to be interpreted*), only a brief description of

[2]Gilligan also describes potential analogies for more advanced concepts that did not get included in the final system. Examples include: objects, pointers, abstraction, and files.

the three forms of I/O that Gilligan's system caters for follows: the computer-user divide is likened to the walls of the clerk's office, and the I/O devices (one each for text input, text output and mouse operations) are computers inside the office that display and then print data (for input), and accept variables then display their content for output. This is a fairly simple metaphor, however it requires that the user not ask questions such as "but why is there a computer *inside* another one?" On the plus side, the associated analogies appear to be fairly high fidelity. In this context the avoided forms of input are those that would alter the behaviour of a program, for example, ActionWorld does not allow for programs that ask the user to enter a number.

As mentioned above, the analogy for methods (or subroutines as Gilligan calls them in his procedural language) is that of a worksheet on a clipboard, what this analogy also takes into consideration is the finite memory of a computer: all discarded worksheets are recycled in order to replenish the stack of blank pages on which they are printed. There is an upper bound to the amount of paper available, and when the paper runs out, the program crashes. ActionWorld did not take such a limit into consideration, primarily because memory limits are less of a concern on modern machines than they used to be - however, one could include a limit such as this by placing a finite number of 'bookshelf-blocks' in the global scope (more on these in Section 4.1.5).

Many of Gilligan's [5] analogies are simple descriptions of how a particular programming structure (such as a loop or conditional) is represented on the 'work sheet' and how the clerk has to handle them. These are not truly stand-alone metaphors, and are more descriptions of the interactions that the clerk needs to perform with the actual metaphors. ActionWorld avoids that issue by automating flow-of-control, and hopefully focusing the user on what happens with the data at each point in a program's execution.

Attempts to extend Gilligan's metaphors occasionally lead to them breaking down: for example, if you want more than the three basic types, you somehow have to cater for more than the three already present calculators (this 'break' can be overlooked because his system was designed to cater for a simplified Pascal Dialect). Gilligan sidesteps passing by reference by saying that his dialect simply does not allow for it. What this sort of thing indicates, is that while his analogies are robust enough to handle his specific dialect, they often cannot be applied to more general cases (which is something that this work attempts to address). No mention is made of global variables, but including them in his metaphor set appears unlikely to break anything.

2.3.2 System 2 - The Hackety Hack Metaphors

Hackety Hack [16] is a system designed to teach Ruby, it follows the more traditional approach of explaining and then presenting users with problems to solve. While the focus of this particular system is not on metaphors, they are used frequently nevertheless.

Hackety Hack likens an algorithm to a "really big to-do list" that the computer needs to follow in order. They make this analogy so as to drive home how unintelligent computers are, and how they will blindly follow their to-do list without regard for the consequences. This analogy is broad enough to almost never break down: one's to-do list can be as short and simple or as long and contrived as needed, so long as the user follows it in order. Conditional structures such as 'ifs' can be easily accommodated into this analogy, where a conditional contains items that can be skipped or done multiple times depending on what the to-do list says about the current state.

As it stands, ActionWorld does not give the user a metaphor for whole programs, this is something that can be achieved fairly easily using this analogy: the user could be told that code to a computer is like a to-do list is to a person, and that it is their job to learn how to read the computer's to-do lists (and eventually write their own).

Hackety Hack also makes a point of saying that programs are *not* like shopping lists, which can be done in any order; this is an important differentiation as it could prevent some users from executing code out of order (the idea of teaching via inapplicable analogies is interesting).

Types are differentiated by relating them to one another and the concepts they represent, rather than something more concrete. Their example compares the integer 2 to the string "2": they explain that the integer 2 is the same as the mathematical idea behind the number, while the string "2" is the number written down on paper. This comparison could both clarify and confuse things: if the user understands the difference they are likely to equate strings with output and integers with theoretical numbers, however novice users (especially young ones) might not see the difference between an idea and the representation of the idea on paper (thus potentially confusing them).

Hackety Hack [16] explains types, and then immediately goes on to variables, seemingly skipping out values (potentially blurring the distinction between values and types). They use the common metaphor of variables being like boxes. There is nothing inherent in the metaphor that indicates that a box can only contain one thing at any time.

They explain method calls by comparing an expression to a sentence. Their example is that of "Turtle.draw": where Turtle is the noun (or object), and draw is the verb (or method call), and while they do not mention it explicitly the '.' connector is equated to a connecting word like 'please'. The result is a sentence that reads as "Turtle please draw something". Continuing with their analogy of programs as to-do lists, they also describe methods as sub-lists that need to also be done in order. Both of these metaphors appear to be high-fidelity, though the 'expressions as sentences' analogy is of limited help to novices who might not realise how carefully and precisely they need to structure their 'sentences'.

2.3.3 Potential Dangers when using Metaphors

Metaphors can be either beneficial or detrimental to novices' mental models depending on how they are taught, as well as the fidelity of the metaphor in question. A poor metaphor runs the risk of confusing matters, rather than clarifying the issues - this can be due to the student having certain pre-existing ideas about the metaphor you are using. This 'baggage' runs the risk of confusing students and actually damaging their mental model of a concept.

For example, methods in programming are often compared with functions in maths:

```
f(x) = 4x + 2;
```

The above would be compared to code such as:

```
private int f(int x){return 4*x + 2;}
```

This comparison seems simple, valid, and unlikely to confuse, provided the student already understands the maths. However, an easy to miss misconception that might creep in as baggage is the implicit multiplication sign between two terms separated only by brackets in a maths equation. A student might think that f(x) means f * x, and try to transcode that into the programming concept. (This case was observed with a struggling student.)

Another example of a metaphor that seems valid is one that describes scope and nested structures as Russian nesting dolls - it's fair to say that one doll or structure fits inside

another, however this kind of metaphor runs the risk of saying that a certain doll or structure cannot 'see' things (such as variables) that belong to other structures or dolls, when in fact scope rules often do allow this sort of thing. Another 'baggage' induced flaw in this analogy is that at some point you cannot make a doll any smaller (in the real world), whereas code constructs can be nested arbitrarily deeply.

A different type of limitation that metaphors need to take into consideration is that they can and do break down under certain conditions. To continue with the Russian nesting dolls metaphor: what happens when you need to consider how the dolls contain variables... adjustments would have to be made to the metaphor so that a doll can contain more than just another doll (for example, specialist mini-dolls to take the place of variables).

This brings the discussion to the final danger that metaphors need to take into consideration: the creation of artificial limitations. If a student is taught a concept using a metaphor that imposes certain constraints when followed literally, or without a certain degree of understanding that the metaphor is limited, that student runs the risk of thinking that those same limitations apply to the programming concept that they are being taught. For example, if one were to re-use the teaching of methods and functions via mathematics equations, when moving on to recursion a student might try to construct something like this for calculating a factorial:

```
f(x) = f(x-1)x;
```

There are multiple issues present with an equation like that, including lack of a base case for the recursion, a lack of type limitations, and no bounds checking. This sort of issue might not happen to a stronger student, who might know to represent the function instead as:

$$f(x) = \begin{cases} f(x-1)x & x > 0 \\ 1 & x = 0 \end{cases} \quad where\, x \in \mathbb{Z}^+$$

However, as not all novice programmers know more advanced formulaic representations, this sort of issue could cause students to think that recursion is not possible, or behaves in hard to predict ways.

These three general dangers (extra baggage, false equivalence, and false limitations) need to be taken into consideration if one is to create a set of metaphors that can be applied to multiple concepts rather than just to one concept at a time.

2.4 Common Misunderstandings and Troublesome Concepts

This section touches briefly upon what concepts cause problems for novice programmers. It can be shown that, in fact, anything can be misunderstood regardless of complexity - and therefore only some of the most common troublesome concepts are listed below.

A study done by Bayman and Mayer [22] found that a group of self taught BASIC students had trouble with even very simple statements: most of the test statements were understood by less than 43% of the students in the study. Further examples include: only 3% understood what “input A” meant, while 27% understood "IF A < B GOTO 99". While BASIC might be less verbose about the meaning of each line when compared to something like C# or Java, the difference is not so extreme as to render this work irrelevant. The major idea that one can take away from this study is that without the guidance of an experienced programmer, students will often have little to zero understanding of the language constructs, not to mention more complicated concepts such as mutable vs immutable, or passing by reference rather than value.

A secondary point that can be taken away from Bayman and Mayer is that even the simplest of statements could benefit from more detailed explanations (which is where the metaphors come in).

In a more recent study, which focused on students’ understanding of recursion, Götschi et al. [23] found that on average less than 50% of first year students held viable mental models of recursion. Relating this to self-taught students (such as those in Bayman and Mayer’s study), shows (unsurprisingly) that there are benefits to having someone help you learn, however it also shows that even after a year of tuition students will often still have poor mental models.

Personal experience has demonstrated that some novice programmers will treat a series of assignment statements more like simultaneous equations than a series of ordered commands. For example, students see line 2 in Algorithm 2.1 and assume that variables x and y are now one and the same, thus their final output is “x -> 7, y -> 7” rather than “x -> 7, y -> 2”. Götschi et al. also observed mental models such as this, which they refer to as an “algebraic model”.

What all of the above examples demonstrate is that students can misunderstand almost anything, regardless of complexity. While some misconceptions are likely to be more common than others, it seems safe to say that by addressing the most fundamental concepts

Algorithm 2.1 Simple assignment statements

```
int x = 2;
int y = x;
x = x + 5;
Console.WriteLine(“x -> {0}, y -> {1}”, x, y);
```

first (by creating high-fidelity mental models for them), the later, more complex concepts are less likely to cause problems because they have a firm foundation to build upon. This concept is used later in order to determine the developmental hierarchy that the metaphors follow (i.e. get the fundamentals well-established before adding anything more that might depend on them).

The start of this section promises a list of some of the more common troublesome concepts, however before providing one it is necessary to compare troublesome concepts to threshold concepts. As explained in Section 2.1.4 all threshold concepts are troublesome concepts (but not the other way around), therefore a list of the most common threshold concepts is likely to also qualify as a list of the most common troublesome concepts as well - additionally, the number of troublesome concepts has to be larger than the number of threshold concepts. Therefore the amalgamated list of most-common threshold concepts in Section 2.1.4, also qualifies as a list of some of the most common troublesome concepts. As a reminder, they are: object orientation; abstraction; levels of abstraction; procedural abstraction; pointers; polymorphism; recursion and induction; and finally, the differences between classes, objects and instances.

2.5 Virtual Learning Environments.

A virtual learning environment (VLE) is any program that strives to put the user in control of a simulated system so that they can experiment with how certain changes affect the environment. For example, an environment aimed to teach students the ideal gas law (PV = nRT), might present users with a virtual box containing a simulated gas, users would then be able to alter various properties of the box (its size, temperature, the amount of gas it contains) in order to see how one change affects the others. Another example is that presented by Trindade et al. [50], which presents users with a virtual reality environment for understanding how water behaves at a molecular level.

As shown above, several subjects have had VLEs created for them, however there is a shortage of environments like this that might illustrate the inner working of a program,

or what happens behind the scenes in a simulated computer. It is not hard to imagine why there is a shortage of this sort of programming VLE: it would only serve to assist novices, but would be difficult to present in a novice friendly manner without presenting users with an overwhelming amount of detail.

ActionWorld is not quite a VLE such as those described above (where the user changes something and the environment adapts to reflect the change), but it instead puts the user in control of the simulation and makes them change the environment based on the code for a particular program.

According to Rutten et al. [51], VLEs can be useful tools, especially for laying solid conceptual foundations for complex or abstract topics. Therefore it seems reasonable to assume that a VLE which incorporates Victor [18] and Tufte's [20] ideas about how to learn and teach programming (as explained in Section 2.2), and which places the role of the computer in the hands of the learner, should be a viable alternative to the more traditional approach of saying "this is the input, I want you to write code that will transform it into this output". Henceforth this alternative technique will be referred to as the student-as-interpreter model.

2.6 Gilligan's System and Programming by Demonstration

Gilligan's [5] dissertation was only found and read near the end of this work - nevertheless, despite not basing any prior research or developmental paradigms on their work, there are several notable similarities. The most important is that their model aims to give learners strong mental models of programming concepts by asking them to demonstrate code, and attempts to do so via metaphors. Taking this broad statement in isolation one might say that both projects are the same, this is in fact not the case. There are several key differences that will be highlighted before going on to further explain Gilligan's work, the most crucial difference is in how code demonstration is defined in the two systems.

2.6.1 An Overview

In order to lend context to the remainder of this section, it is necessary to briefly describe both Gilligan [5] and ActionWorld. More detailed descriptions of various components of

each system can be found elsewhere, Sections 2.3.1 and 4.1 describe Gilligan's and this work's metaphors respectively, while the remainder of the dissertation elaborates on the details of the proposed system.

The proposed system presents users with ready-made code that they need to interpret: in order to advance in the code users must interact with the appropriate metaphor in the game world. Users receive immediate feedback about whether their action was correct or not, and at no stage does the system attempt to generate code based on users' metaphor-interactions. Figure 2.5 shows what a user would see upon starting a new level in this system, metaphors not relevant to the current level can be hidden based on user preference (for example, the memory robot, communication terminal, and the double notepad).

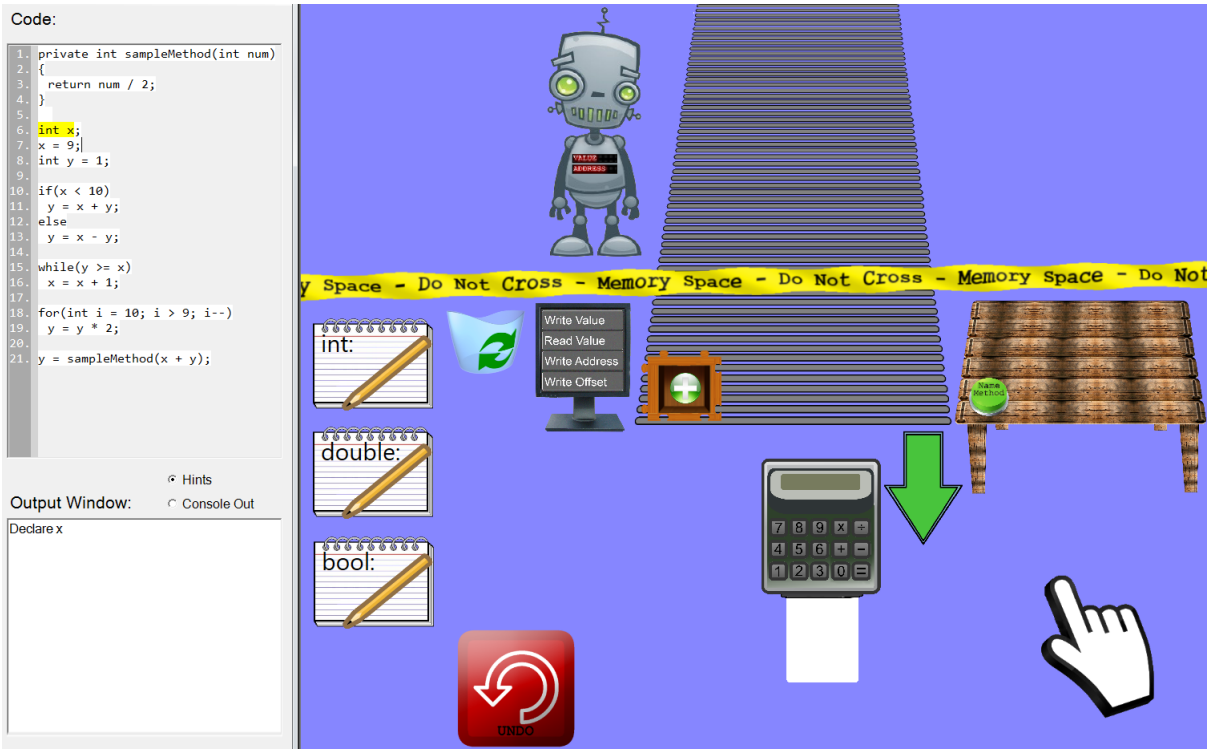


Figure 2.5: The WPF implementation of the proposed system at the start of a level. Left: the code that the user needs to interpret. Right: the metaphor playground that users must interact with. The Image has been cropped and compressed somewhat in order to accommodate spatial limitations.

Gilligan's system also works via demonstration, however rather than presenting users with pre-made code and asking them to explain it, the user is presented with several interactive metaphors which they can combine however they choose (i.e. demonstrate the desired program behaviour). Based on how the users interact with and arrange the various metaphors, the computer then infers and displays the PASCAL equivalent (or its

best guess). Figure 2.6 shows what users would be presented with when starting Gilligan's system.

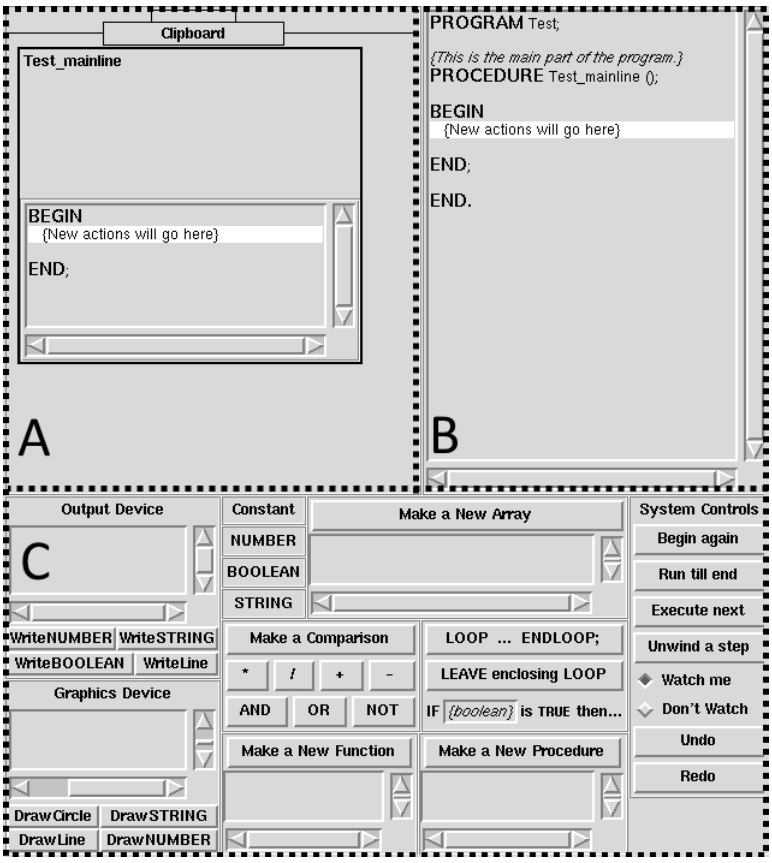


Figure 2.6: A screenshot of Gilligan's second prototype. Users drag objects from area C onto the clipboard in area A, the corresponding code is then inferred and displayed in area B.

2.6.2 Key System Distinctions

Both ActionWorld and Gilligan's system ask users to 'demonstrate' through interaction with metaphors or analogies, but what is being demonstrated and what results from this demonstration differs significantly. Gilligan's [5] system presents users with a form-like environment and lets them experiment using assorted graphical user interface (GUI) components. Upon interacting with an object, matching code is creating in their simplified PASCAL language. What this means is that users have no goals or limits, and the output of their actions is actual code. ActionWorld takes the reverse approach. Users are given a piece of code, and asked to demonstrate its meaning line by line via metaphor interactions (except when this system is in sandbox mode, where no code is displayed at all, and

users can simply experiment). One might be forgiven for thinking that this difference is unimportant, however it more than just differentiates the two systems: it also shows that both systems are taking a fundamentally different approach to achieving the same goal, and therefore cater to different learning styles. This system's primary teaching style is the acquisition of automaticity via linear learning (with sandbox mode, again, being the exception), while Gilligan appears to focus entirely on exploratory learning.

In much the same way as this work, Gilligan also aimed to create something of a complementary set of analogies (i.e. metaphors that work well together). Section 2.3.1 explains these in more detail, along with other established metaphors. The short version of his analogy is that the computer is portrayed as being a clerk at an office desk, and the user takes on the role of the clerk.

Before actually reading Gilligan's dissertation a similar enhancement had already been considered for ActionWorld: one to allow for reverse-mode activity, that is, code generation in response to the manipulations performed during sandbox play - however after considering the various implications the idea was rejected. There were two main reasons why the idea was rejected: the most important one was the risk that students who learn to program in such a fashion (without conditional structures being presented to them somewhere) might be more prone to write problem specific solutions rather than general solutions. The second reason that the inclusion of reverse-mode code generation was decided against, was that the metaphors and implementations had not been created with that sort of functionality in mind, and thus such behaviour would likely appear out of place or poorly catered for. Section 9.6.1 goes into more detail regarding reverse-mode code generation as a potential future extension to the system.

If one were to judge Gilligan's system based entirely on what Kelleher and Pausch [4] say about it, one might unfairly condemn the whole system as having no support for variables. However, what Kelleher and Pausch fail to mention is that it was only one of Gilligan's two implementations that did not support variables, and this was due to time constraints preventing implementation completion. Their model as a whole in fact supports three distinct variable types: numbers, strings, and Booleans. One might still perceive this as something of a limitation because most modern languages have many more fundamental types, a simple example would be the difference between integer and floating point numbers (which would have to be treated the same in Gilligan's system). This work takes a different approach, and allows users to declare variables and values of any of the fundamental .Net types.

2.6.3 String Type Implications

Gilligan's [5] handling of string types is a potentially interesting illustrative feature, depending on whether their PASCAL dialect treats strings as mutable or immutable. Pascal's original definition had a number of severe limitations with string types (a maximum of 255 characters being just one of them). In modern dialects (such as Delphi) this has led to the growth of a number of differently optimized string types to cater for short strings, long strings, and strings that are concatenatable without copying-on-write overheads. Because of these limitations, there are currently moves being made to rationalize and perhaps introduce immutable strings [52].

What this means is that if Gilligan [5] adopts a representation that treats strings like the simpler number and Boolean types, it will almost certainly build an internal mental model that may not be a good match for what the student subsequently encounters. While if he intends to treat them as immutable, students are likely to be better equipped, provided his in-game representation is sufficiently different from those of the simpler types. Considering the date of publication it is unlikely that Gilligan intended to represent strings in an immutable form, which is a more modern (potential) development. This concern might not have been relevant when Gilligan's [5] system was being made as programming paradigms change so much over time, as Gilligan makes no mention of mutability.

2.6.4 The Interface

The only *relevant* weakness of Gilligan's system that has not yet been discussed, is that it does not use images to enrich the user experience - instead it uses a GUI made up of standard buttons and generic components (Figure 2.6 illustrates this). So even though the metaphor for a variable is a box, there is no box to help the user construct the mental imagery. Gilligan notes that the interface is something of a weakness, but goes on to say that he wanted to focus on the functionality of the system rather than the aesthetics (a design choice also adhered to in ActionWorld, as explained in Section 3.5). This could also have been done partly due to technical limitations at the time (after all the program is nearly 18 years old), nevertheless it is a point upon which ActionWorld is able to improve.

2.6.5 The API and Gilligan's Virtual Machine

Gilligan refers to his underlying model as an abstract machine that is not concerned with syntax, scope, and most other things that regular compilers need to take into consideration. He goes on to say that his machine assumes that it is passed a valid program for every run. Without explicitly intending to, the virtual machine created for this system does something similar - it assumes that a program (or level) is valid and complete, and then links the user to the virtual machine via the assorted metaphors. It appears as if the key difference between Gilligan's [5] abstract machine and the one created for ActionWorld, is that his interprets a PASCAL dialect, while the new one interprets what amounts to an intermediate language created specially for each level (in theory somewhat like what C# code gets compiled down to, though not comparable at a functional level).

2.7 Summary

This section covered a wide range of topics, ranging from Constructivism theory to a survey of existing games, and it highlights a number of important ideas:

- Education theory in general is a wide reaching topic with occasionally conflicting theories (for example, creation of automaticity is best done through repetition, while learning through exploration is believed to improve retention). Regardless of these conflicts, the relevant theories can still be used to improve on ActionWorld's design in order to increase its effectiveness.
- There are proven benefits to the visualisation of data and flow of control.
- Learning to program is hard, often because foundational mental models are flawed and need to be re-factored. One potential source of flawed concepts is the adoption of 'weak' metaphors as mechanisms to explain programming concepts.
- There is a huge variety of material aimed specifically at teaching programming, and yet no single system has been able to conclusively say that they have found the best way to teach. This means that an alternative system to the more common ones may be beneficial (in the very least because it adds variety).

This section showed that while ActionWorld is similar to Gilligan's [5] system, they have several fundamental differences. ActionWorld appears to build on Gilligan's in much the

same way that Snap! [15] builds on Scratch [14], which in turn builds on a system proposed in 1986 by Soloway [53] where novices learn by combining algorithm components through a simplified medium.

Gilligan performed a brief analysis of their system based on programming fallacies presented by Eisenstadt et al. [19], and in order to compare the two systems using a similar standard the same thing was done here. This is only elaborated on later, in Section 7.3.1.

The next chapter elaborates on the methodology used to develop and improve upon the various attributes of this work.

Chapter 3

Methodology

The previous chapter provided some background from the field of education theory, as well as several examples of systems which use metaphors to teach programming. This chapter explains how the last chapter's theory influences development, along with all the details regarding the design process followed during development of ActionWorld and its constituent components. A description of the design based development paradigm used is included, along with several design patterns aimed at improving quality (such as the Model-View-Controller (MVC) architectural pattern).

3.1 Defining Artefacts and Deliverables

This research will output at least two distinct artefacts for assisting in the education process. The metaphor set is intended to be usable by both instructors and learners, across multiple educational media including live lectures, videos, textbooks, and interactive games. The second distinct artefact is that of the API, which will make the task of creating educational systems such as ours easier. These two primary artefacts create a need for several secondary ones: at least one implementation of the proposed game so as to demonstrate the quality of the metaphors and the API, as well as a level editor which allows educators to craft curriculum specific levels.

Due to time and scale constraints, thorough testing of all the aforementioned deliverables is not feasible, therefore this work includes another secondary deliverable that one might not consider a requirement, but does serve to improve quality: a testing and evaluation framework for the primary artefacts.

One could go further, and argue that this dissertation delivers a total of four separate secondary artefacts (the two different implementations of the game, the level editor, and the test framework), and that they are usable by both teachers and students: the actual game component is aimed more at learners, while the level editor can be used by anyone to create custom lessons, and the test framework is geared towards academics who wish to expand on this work.

The next few sections elaborate on the design and development process used during the creation of the aforementioned deliverables.

3.2 Iterative Process

This work places a great deal of emphasis on iterative development, in fact one could write several sections detailing all the iterations this work went through. For now it will suffice to detail the steps undertaken during the design and development process.

There are two separate iterative procedures that were followed, one for creating the set of metaphors and one for developing the API and games. The latter process is broken up into distinct phases.

3.2.1 Metaphor Development Methodology

The steps involved in creating the metaphors are as follows:

1. First one needs to list and explain the various concepts that one wants to be able to represent and why each one is necessary. The list is then ordered according to the necessity of each concept (for example, being able to show at least simple methods should always be more important than being able to demonstrate recursive methods).
2. Select a structure or concept that needs representation, generally starting with the most fundamental of those which do not yet have representations.
3. Propose a metaphor or metaphor refinement for the current structure or concept.

4. Identify potential limitations of the proposed metaphor (without concern for complications that the existing metaphors might introduce, see next step). Problems or limitations here require that one goes back a step.

5. Consider how various existing metaphors would fare with the newest addition to the set (and vice versa). If issues are found at this stage one must go back several steps to identify whether there is a weakness in the existing metaphors or in the newest one. A common issue at this stage is that a metaphor might be sound in isolation, but a meaningful relationship could not be established with the other metaphors, thus inclusion is not an option as it would not result in a unified set of metaphors.

6. Once happy that all the metaphors *so far* mesh together (and are accurate enough), the process is repeated from step one to include the concepts or structures that aren't represented yet.

3.2.2 API Development Methodology

The process used for API development was a little more concrete, as it was known what kinds of structures needed to be included almost from the outset. For example, a variable class was essential, which in turn needed a value class. Bear in mind that these steps are for a set of classes that are non-graphical, and essentially make up an abstract virtual machine. Here are the basic steps followed during the development of the API:

1. Following a similar order to that used in developing the metaphors, the next structure that needs an API representation needs to be selected.

2. One would then create a class to represent the concept or structure in question.

3. Once the rough class is created, the question "what sort of non-basic functionality would one expect from this class?" needs to be asked and answered. The answer dictates the functionality that must now be added to the new class.

4. This step involves linking the new class to the existing ones wherever a link is necessary. If at this stage two classes do not 'mix well' for whatever reason, one or both of them would have its structure revised. This only happened once or twice during these iterations (more specifically with the calculator), and issues were more to do with the metaphors being represented.

5. Provided one does not yet have abstract representations for all the desired constructs, one would then go back to step two and repeat the process for those that remain.

6. Once one gets to this point the API is nearing completion, and so it needs to have simple (generally textual) tests run on it as a whole, to ensure that everything works as expected. Any problems here would mean going back through previous steps of this process (depending on the severity of the issue the number of repeated steps and revised classes would change). Fortunately, during implementation, this step seldom showed up any major issues.

If at any point in the development of the API a concept is encountered that has been left out during metaphor development, or if anything else is discovered that has not previously been considered regarding the metaphors, development goes back as many steps as required to address the issue. In practice, this sort of alteration generally did not mean that the API needed to be changed significantly, as it is mostly independent of the metaphors. To clarify this distinction, one can imagine the boundaries between a virtual machine, and the front end one uses to communicate with it (not unlike the Java virtual machine and the assorted develop environments that communicate with it).

3.2.3 Game Development Methodology

Once the API was up to standard the next step was to create a functional game based on both the API and metaphor set combined. If during the process of game development it was found that something was missing from the API, development would go back to the API phase to address the issue. If the issue was serious enough, development would go back to the metaphor phase (this did actually happen with regard to calculators vs. notepads, as described in Section 5.4).

The game development phase is only considered complete when one is no longer able to significantly improve the experience. This last stage is almost made up of sub-phases: if there was a failure in any one of these sub-phases one could either go to an earlier sub-phase or back to an earlier main phase (API or metaphor development). These are the steps involved in the game development process:

- Add a visual representation of the next metaphor in the list of concepts to represent.

- Test that it works correctly.
- Test that it interacts sensibly with the other metaphors.
- Criticise the layout and metaphor representation in order to try to improve the experience that it might give a user.
- The process is then repeated from step one for the next concept.

3.2.4 Iterative Hallway Testing

During implementation, getting to this stage was an indicator that development had reached the limit of what improvements could be made using 'in-house' testing and development. Therefore the very last set of development steps centre around user testing. Rather than perform a full-scale user test that might be appropriate in an HCI project, this phase used a relatively cheap and lightweight approach advocated by Joel, called corridor or hallway testing [54]. The game prototype was set up in the undergraduate computer laboratory during CS1 practicals, and users were invited to give the game a try. Section 5.4 gives a more detailed account of the results of the corridor tests, and is a good demonstration of the iterative methodology described above. These are the broad testing steps undertaken (deviating from these steps is not a major issue):

1. The game would be presented to users, who would then be asked to play it.
2. Details of their experience would be recorded as they interacted with the game.
3. A set of question would then be posed, in order to get their opinions.
4. Finally if there was a significant improvement to be made based off of user feedback, development would go back to the appropriate stage in order to implement it. This could be as simple as re-arranging sprites in the game development phase, or altering a metaphor to clarify its meaning (all the way back at step one).

By this stage the design and implementations of the game and the metaphors should have stabilised to a point where development could comfortably move onto the next phase (that of validation).

3.2.5 Validation and Evaluation

The next phase is about validating the quality of the game, the metaphors, and the work in general. The quality assessment stage is not as iterative as the other development stages; instead it involves taking some form of quality assessment criteria and grading the system accordingly.

Several quality evaluation and validation techniques have been proposed, including: anonymous qualitative surveys of novice and experienced programmers (both asking for opinions on the metaphors and asking them to use the metaphors to demonstrate how a piece of code should work); quantitative surveys of students based on their marks and use of the game (this would require an amount of time that would exceed the scope of this work); questioning focus groups of programmers to try and elicit responses that might not be obtainable through surveys; one-on-one interviews with programmers of various levels, as well as several more quantitative measures such as checklists, heuristics, and system comparisons.

As there are so many potential measures of quality, of both quantitative and qualitative natures, two separate chapters were created: the first (Chapter 6) gives an overview of the most common evaluation techniques along with sample tests for each technique, while the second (Chapter 7) elaborates on the results of the applied tests.

3.3 Design Based Methodology

So far two key design issues that this system adheres to have been explained: deliverable artefacts, and an iterative approach. Several established methodologies rely on these two ideas. Of these, this work's methodology aligns best with the so-called *Design Based Research (DBR)* advocated by Juuti and Lavonen [30], and adheres to the majority of their criteria. According to Amiel and Reeves [55], when creating an educational technology it is important to not only consider the final artefact, but to also take the process of design and refinement into consideration. This process is something that the iterative DBR approach addresses with a great deal of success: end users are consulted for feedback throughout the process, it merges design principles with technological advances in order to better serve the end user, and finally it relies on a rigorous iterative cycle of design-test-reflect-refine in order to ensure quality and fidelity [56].

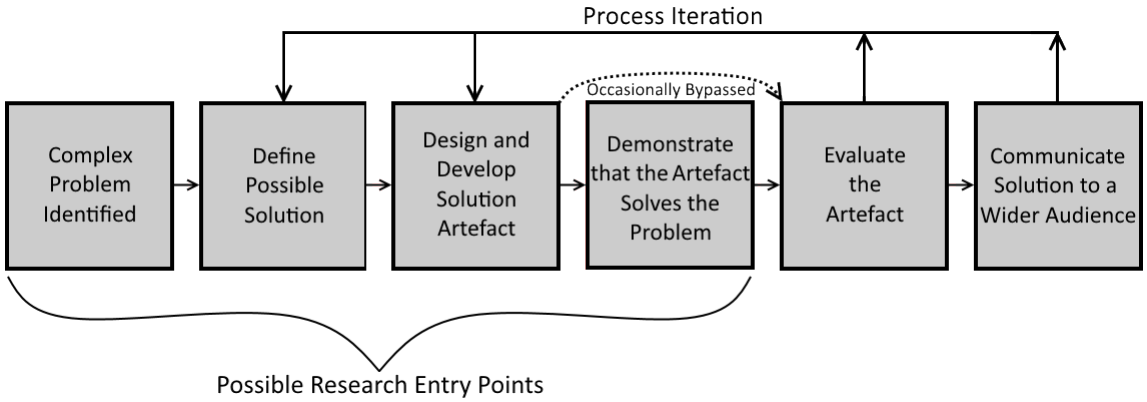


Figure 3.1: A block-based flow diagram illustrating the iterative steps involved when using a DBR approach [3]. De Villiers and Harpur did not state that the demonstration phase is optional - this was included to demonstrate one of the steps that was occasionally bypassed during the course of this work.

Figure 3.1 shows a general block-based flow diagram, typical of most DBR projects, as explained by de Villiers and Harpur[3]. Note the annotation regarding the optional nature of the demonstration stage - it is the authors experience that after the design and development stage, one often does not need to demonstrate the behaviour of the artefact - perhaps because the outcome is already obvious. An example of this may occur near the start of a project, where the researcher already knows that the artefact is not yet able to solve the problem - in this situation one can go directly to evaluating the artefact's shortcomings before returning to an earlier stage to rectify them.

During the course of this research, multiple different artefact were created - each one with its own flow diagram. Due to the scope of the project, providing a comprehensive set of flow diagrams is infeasible. However, as there is very little variability between diagrams, one has been included in Figure 3.2. It shows how most artefacts had at least three different types of testing and quality assurance applied to them (fidelity checks, understandability checks, and Hallway Usability checks), these were occasionally performed in an informal manner, while the majority of evaluation results were obtained through the applications of the test framework explained in Chapter 6.

This figure also demonstrates that there is occasionally deviance from the more generic version shown in Figure 3.1. The primary difference shown is how the different artefacts' development flows merge together, and often influence each other, through the forward and backward propagation of changes made to rectify identified issues (regardless of what stage those issues were identified in). A secondary difference between Figures 3.1 and 3.2 is that iterations could begin at almost any stage - rather than just the final two.

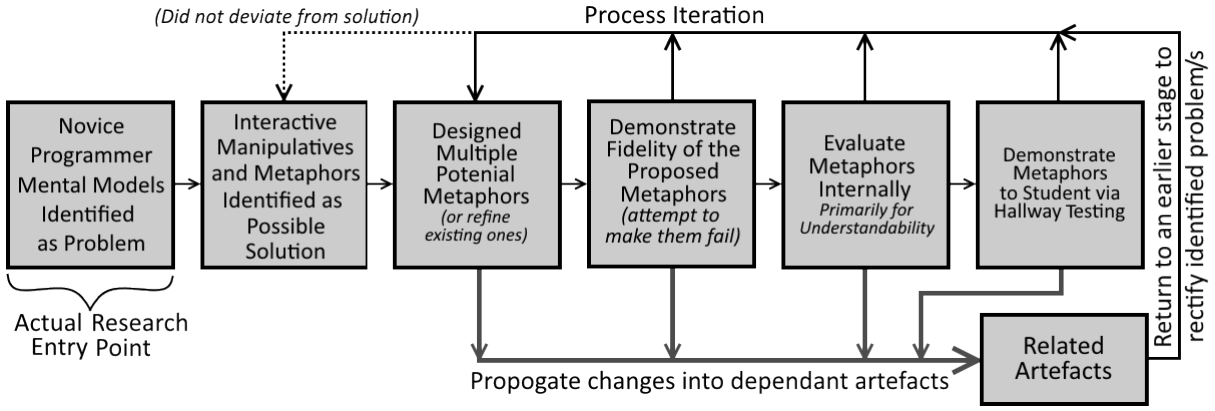


Figure 3.2: The DBR flow diagram used during the design and creation of the metaphors. Notice that later stages in the development process feed into this diagram (or are fed by it) - this can include the game development stage, test framework application, or even the alteration of a related metaphor.

This cycle of problem identification, rectification, and proliferation only ends when no changes are made on a particular iteration - at which point the project shifts focus onto the related (dependant) artefacts. As an example of how often this cycle occurred, consider that no less than fifty different potential metaphors were considered, built, and ultimately rejected, before settling on a set of nine for use in the final program. The in-game representations of those nine metaphors were then altered multiple times based on the Hallway Usability Testing results (explained more in Section 5.4) before reaching their final state.

3.4 MVC Build

Due to initial uncertainty around the final form that the metaphor set would take (in turn based on which metaphors complement each other best, while retaining fidelity and encouraging consistent mental models) the popular Model-View-Controller (MVC) design approach was used when implementing the game. An MVC approach allowed development of the underlying API for tracking state and fundamental logic (the Model), to be done separately from the visual metaphors and manipulatives that the end user is presented with (the View). The model and the view then come together through the Controller. Figure 3.3 illustrates how the MVC paradigm was applied to the system, and gives a good idea of how interchangeability is encouraged via MVC.

This design and development methodology allows one to easily switch out components on either the front end or the back end without any side effects. During the course of devel-

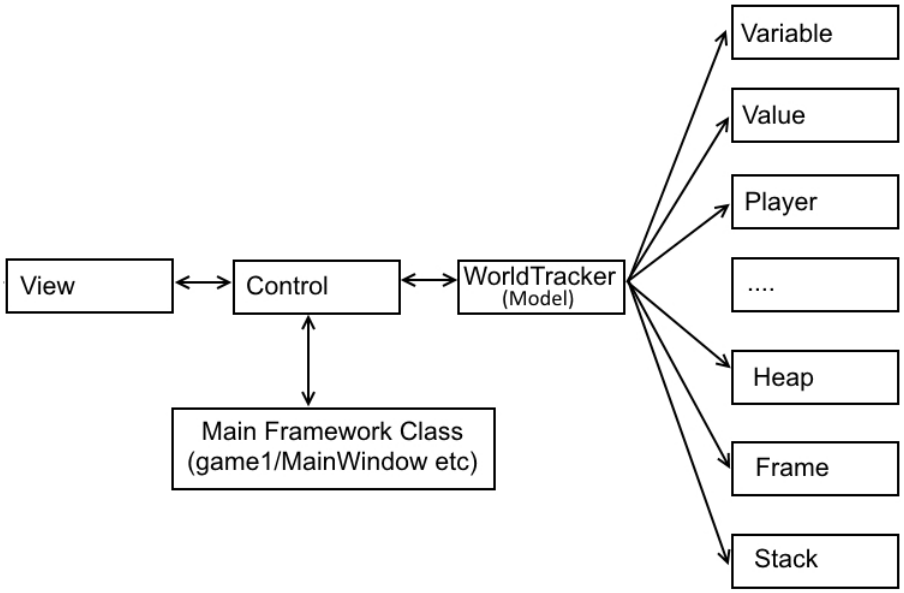


Figure 3.3: An illustration of the MVC design the system used. View keeps track of components such as sprites (and their associated details), the Model keeps track of the state of the virtual machine, while Control manages user interaction and facilitates communication between the View and the Model.

opment several components of the underlying API, as well as the displayed metaphors, needed to be switched out and improved upon - especially once usability testing began (as explained in Section 5.4) - this need for frequent easy switching proved the value of the MVC approach that was used. A simple example of a change to the Model, would be to alter the order in which actions needed to be performed (an example of this was in regard to variable declaration and assignment as explained in Section 4.3). View refinements were generally made when a user did not understand something as expected, or when users provided feedback about the interface lacking something or being unintuitive.

Most of the time changes to the Model component seldom affected the View (and vice versa), however this was not always the case: when the decision was made to include an alternative means of getting expression values from the user via notepads, both the Model and the View needed to be altered.

Using the MVC paradigm not only makes development easier, it also makes the underlying API more portable, thus allowing others to use the API as it is without making any changes. The MVC decoupling makes it particularly easy to respond to feedback from the users.

3.5 Functionality over Aesthetics

As mentioned before, one of the primary goals of this project is to create a set of unified high-fidelity metaphors, which can be used in various environments. With this in mind the most important part of the metaphors described is not whether they look good in any single implementation, but rather that they can be used in an assortment of different environments (for example, real-world class rooms, images in textbooks, XNA games, or GUI applications) and more importantly that they accurately portray the concept in question in a manner that is unlikely to confuse novices.

A good deal of effort went into the appearance of the two game implementations (including images and layout), however without the resources to acquire the help of professional graphic designers, occasional trade-offs between functionality and aesthetics had to be made. In these situations functionality was prioritized. For example, it is more important that the user has a stream-lined understandable experience when assigning to a variable, rather than one that looks good but risks causing distractions or misunderstandings. Continuing with the variables example, a representation was created that *looked* better than the final one, however it confused students and so it was replaced. This project is not the first to put function over aesthetics, as shown by Gilligan [5], who acknowledges his sub-optimal interface and justifies it in much the same way.

Section 4.3, while being focused on metaphor optimisations, also has several good examples of visual changes that vastly improved the user experience without altering the underlying metaphors.

3.6 Summary

This chapter explained the iterative methodology followed during the various stages of this work: where any identified issues or improvements would result in development returning to an earlier stage, in order to implement the required changes. Two secondary design and development paradigms were also explained: MVC and Functionality over Aesthetics.

The next chapter describes the proposed metaphor set, and then goes on to explain earlier stages in the design of the metaphors. The description of earlier stages relates directly to the iterative approach described in this chapter.

Chapter 4

The Metaphor Set

This chapter describes the final metaphor set after optimisations were made, some examples of the earlier metaphors before optimisations, alternative representations, potential enhancements, and finally a look at the early developmental representations that the system as a whole went through. Some of the final analogies did not change at all, even after being presented to students (values on paper are a prime example), while others underwent drastic alterations. When students seemed to not grasp a particular metaphor during usability testing (see Section 5.4 for examples), the troublesome concept and associated metaphor were considered carefully and altered to address the issue, and it is primarily the improved metaphors that are described here. Section 3.2.1 describes the iterative development of the metaphors in more detail, while Section 5.4 contains more details about what students struggled with and how the issues were addressed.

In general, all metaphors that indirectly represent a concept must eventually constrain or misrepresent reality. Thus the metaphor design boundaries need to be informed by the overall goal of the work: to support an introductory programming course. In this work, elements like expression evaluation, variable declaration, assignment, strong typing, reference types, value types, method calls, parameter passing, and values returned from methods were considered core notions. Other more advanced ideas such as multi-threaded programs, object instantiation, garbage collection, inheritance, method overriding, and nested scopes were not prioritized, and the metaphors no longer need to hold in those advanced situations.

This chapter is not arranged chronologically - if it were, it would start with various early metaphor proposals, move on to the final metaphor set, and end with potential future enhancements. Instead, it opens with the more important information (the finished

set), and everything that follows can be considered additional detail regarding stages of metaphor development (as explained in the previous chapter) and future work.

4.1 The Finished Metaphors

4.1.1 Values

The most fundamental thing in any program is an individual value, either as actual value or a reference to a location in memory (this would include integers, doubles, Booleans, chars, and all other primitives). This particular concept is so fundamental that it applies to almost all languages (including functional ones). As this is such a key concept, its associated metaphor needed to be one of the most reliable and easy to understand: it was concluded that a simple piece of paper with the value written on it would be suitable, provided the paper was given some special attributes to cater for special cases (such as value copying, and immutability).

Everyone can relate to pen and paper, you cannot erase pen from paper (meaning values, not variables, become immutable), and if one were to imagine solving an expression in their head the most sensible thing to do with the answer would be to write it down. This representation of values laid the foundation for the remaining metaphors. Figure 4.1 shows an example of a value notepad, the user inputting a value, and finally the user holding the value[1]. The paper used for this metaphor is volatile - once it has been written on and placed in a variable box it cannot be removed without being destroyed (this is to ensure that learners do not make the mistake of trying to move a value around without creating a copy).

4.1.2 Variables

Variables are represented by boxes with transparent lids that contain a single piece of paper (in the same way that a simple variable can contain a single value). Reading of a variable is done courtesy of the transparent lid: you simply look into the variable box and copy the value off the paper without changing the content of the box. Assigning to a variable involves opening the box, disposing of the old value-paper, and then placing

[1] The writing utensil in this figure might appear to be a pencil. It is, in fact, a pen!

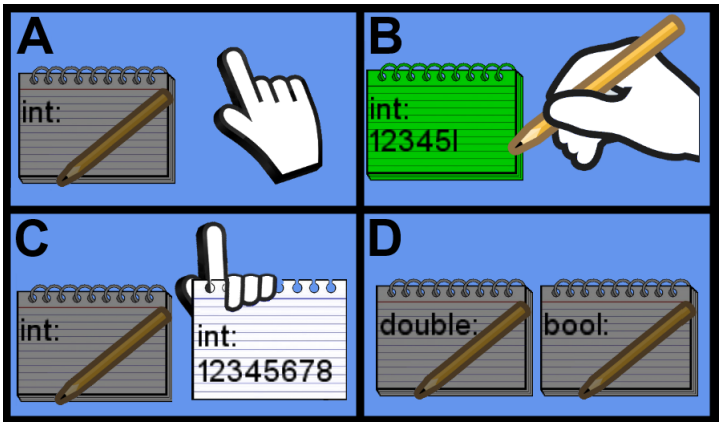


Figure 4.1: A: An int value-notepad. B: User inputting a value. C: The user holding the final value. D: Alternative value-notepads.

Figure 4.2: A variable box called counter, containing the integer value 12345678.

the new value (which you would be holding) into the box. The metaphors for both values and variables can be extended to include type limitations: one potential way to do this would be to use different coloured paper and boxes for each type, or alternatively you could have boxes and pieces of paper that are of different shapes and sizes, so that one type cannot fit into another type, or finally you could make users use different pens to write down differently typed values. These potential extensions do not interfere with the more advanced metaphors. Figure 4.2 shows one potential version of the variable box metaphor. The representation of variables on the stack comes later, up to this point it is enough for users to know that variables exist and how to imagine them.

4.1.3 Expressions, Arithmetic and Calculation.

Expressions of all kinds have two viable metaphors in the unified set: users can either be presented with an explicit calculator (which then requires special interactions and operations) - or they can have expressions explained as being transient, and thus they need to work the result out for themselves and only store the result. Good arguments can be made for either case, and thus both metaphors have been included so that anyone who

wants to expand on this work can make up their own mind - Section 4.3.4 goes into the pros and cons of the two representations. Originally the explicit calculator metaphor was used to explain expressions and evaluation - it worked much like a real-world scientific calculator, where users could enter an expression and then go back and alter terms once they had the required values. For example, a user might enter "x + 5" into their calculator, they would then look at the local variables and copy the value from x onto a piece of value paper, and that value would be fed into the calculator (almost like a fax machine, except the paper is destroyed).

When the user asks to substitute into the place holder 'x' in the expression, the calculator would replace the variable in the expression and then wait for further instructions from the user, or for the user to ask it to evaluate the answer. When the expression no longer has unfilled 'variables', the user would hit evaluate and the answer would be printed out from the calculator onto a piece of paper. Figure 4.3 illustrates the sequence of events when using the calculator metaphor.

One of the major downsides to this method is the number of steps involved in evaluating an expression, for example, 'x + y + z' would require: inputting the expression, three explicit variable reads, three substitutions, and finally evaluation. This number of steps might not sound like much, but for a beginner every extra step runs the risk of distracting from the code or confusing them. A secondary downside to using the calculator as an expression evaluator is that the user would not need **any** understanding of types as the calculator would simply output the correct type along with the answer. This issue could be addressed by altering the metaphor so that users would have to be explicit about the type returned upon evaluation (almost like a compulsory casting step). One should note, however, that these drawbacks only occur when using the metaphor as a manipulative (such as in a game) - if one is simply explaining expression through the analogy of a calculator, these concerns are removed by the fact that the user no longer needs to perform the numerous steps, and can instead simply imagine the procedure or have it explained.

The multiple calculators option used by Gilligan [5] is impractical, as there are so many types in ActionWorld (whereas Gilligan's system has just three types).

The second, easier, technique for expression evaluation in ActionWorld removes extra metaphors entirely by just asking the user to evaluate the expression outside of ActionWorld - in their heads (or on paper, or on their real-world calculator) - in much the same way as when users debug a piece of code. This second method poses fewer problems (for example in requires only one in-game step, as opposed to several) but it is not perfect,

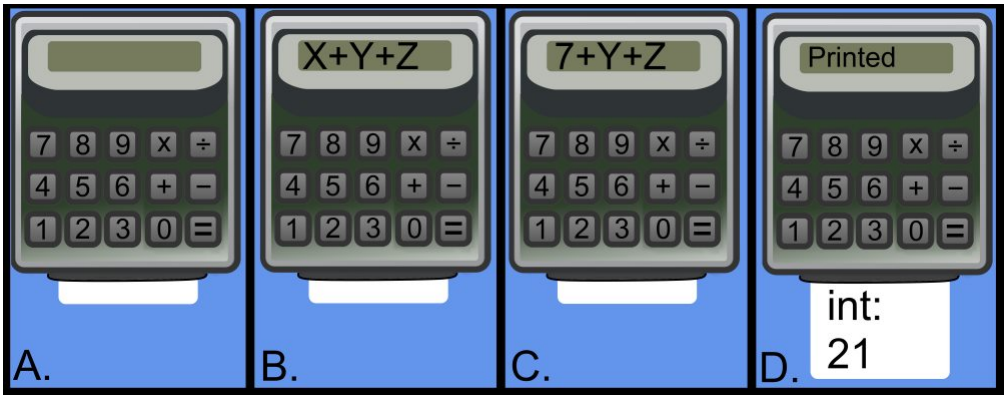


Figure 4.3: The steps required to evaluate an expression using the calculator. A: the calculator is blank and requires an expression. B: An expression has been entered and now requires value-substitutions for every variable. C: A value has been substituted (this requires a variable read first), this step is repeated 3 times. D: All values have been inserted and the user has asked for the answer (which was printed onto a piece of value-paper).

the biggest flaw is in fact one that it shares with the calculator metaphor: how to deal with expressions that contain method calls, Section 4.4.1 goes into more detail regarding this issue.

One might note that having multiple notepads is similar to Gilligan's calculators, and it is therefore subject to the issue of too many types. This issue was overcome by only presenting users with the more common types as notepads, and anything more complex requires the use of the calculator instead - thus the two representations can be used side by side.

4.1.4 Conditionals

In what has been described so far, values always end up being stored in a variable. But in a conditional statement the Boolean test expression serves only to direct the flow of control, and the value is "consumed" in the process. Some mechanism had to be created to allow for value "consumption", when the value was not to be stored in any way.

While ActionWorld attempts to remove the responsibility of program flow from the user, it is still necessary to include some way of handling, or at least demarcating the locations of, conditionals. This need is based on the fact that the result of a boolean expression does not have anywhere to go or be put yet, thus leading to behavioural ambiguity about

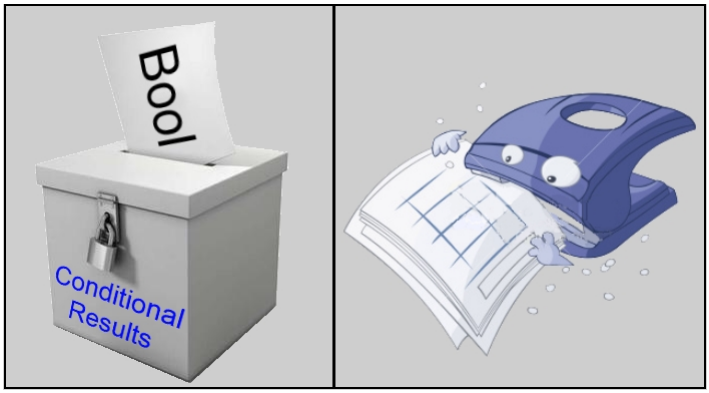


Figure 4.4: Two possible representations of the BoolEater conditionals mechanism.

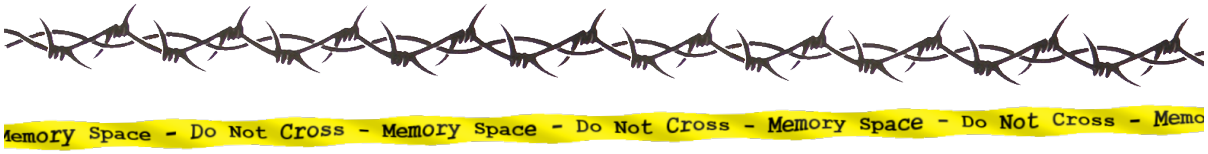


Figure 4.5: Top: Barbed wire memory barrier. Bottom: Police tape memory barrier.

what needs to be done at a conditional. The answer to this simple need is an equally simple mechanism: a Boolean value eater.

After a user has evaluated a boolean expression that is part of a conditional structure, the resulting user-held paper-value is disposed of by feeding it to the BoolEater, whose responsibility it is to direct conditional flow. Figure 4.4 shows two example representations of the BoolEater.

4.1.5 Local Variables and the Current Stack Frame.

The next step up the abstraction ladder is to delimit local and global scope using some kind of uncrossable barrier (for example, a line of barbed wire, police tape, or even a fence - Figure 4.5 shows two possible barriers), this barrier serves to separate the user space (the local scope) from the memory space (the global scope, including the stack and the heap). The global area, and everything in it, will be discussed later. Scope-delimitation was mentioned here for clarification of the next concept to be described: how to represent the current frame and all the variables that are stored within it, while still being relatable, and without complicating the pushing of stack frames into the global scope.

Several different possibilities for the local frame were discussed, and finally something akin to a jigsaw or Lego bookshelf was settled upon: users would start out without a bookshelf

(provided there are no local variables yet), and a mechanism which dispenses bookshelf 'pieces' is always present. Whenever the user needs to declare a new variable they would first have to extend the bookshelf so there would be enough space for it. Each space on the bookshelf is able to contain one variable box. This metaphor might be slightly less relatable than using a normal bookshelf, however a simple bookshelf runs the risk of making the user think the current frame has a fixed size (which is not the case). Another advantage to the expandable bookshelf metaphor is that if you want to teach students about how variables can go out of scope (for example, when they are declared inside a loop) the out-of-scope variables (and their associated bookshelf sections) can be removed entirely.

If one is concerned about representing stack limitation (such as those imposed by memory limitations), this metaphor can be adapted slightly to allow for their inclusion: the bookshelf-block dispenser can be made to have a finite number of blocks, that way if it runs out of blocks a memory error can then be shown. When returning from a frame, or deallocating local variables, the used pieces could then be returned to the dispenser.

The metaphorical bookshelf sits on top of a conveyor belt, this only becomes important when the user is able to call methods, thus it is discussed in more detail in the next section. Figure 4.6 shows what the user might be presented with depending on the current stack frame state. Alternative methods of representing the stack and individual frames were discussed and almost uniformly rejected for various reasons, Section 4.2 goes into more detail on these.

4.1.6 The Stack and Methods

Once one understands how to interpret the representation of the current frame, one is then also able to interpret the stack as a whole: just like a library usually has more than one bookshelf, a stack usually has more that one frame - therefore one can cross the two ideas and represent the stack as rows of bookshelves. This is also where the division between local and global scope becomes important, as the stack is primarily located in memory space rather than user space. When a new frame is created from a method call, the conveyor belt moves the current (not new) frame-bookshelf across the memory barrier divisor so that it now exists in memory space (thus allowing for reference variables to be accessed in the same way as objects located on the heap, which is explained later). Figure 4.7 shows how the stack would expand into memory space after each method call, and shrink after each return statement.

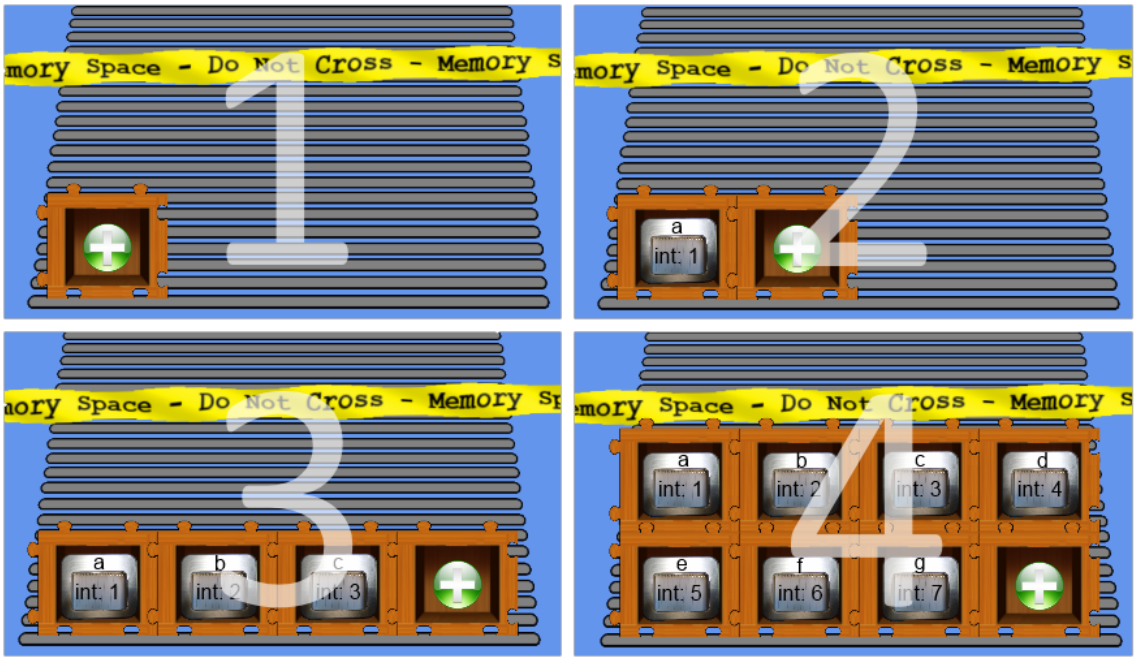


Figure 4.6: Four snapshots of the current frame and local variables bookshelf - displayed on top of the frame conveyor belt, and inside the user space. Demonstrating the various states during the declaration of local variables 'a' to 'g'.

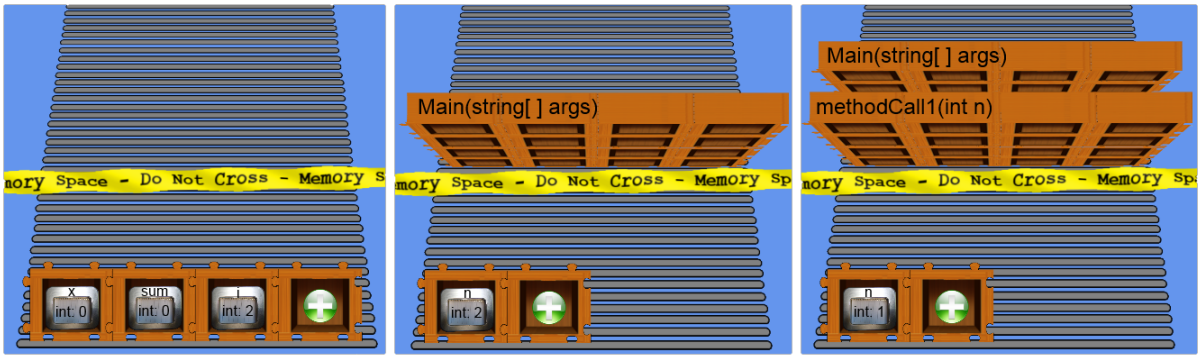


Figure 4.7: Changes in state and visualisation for the stack and the frames that correspond to each method call. Read from left to right this shows the calling of each method, while read from right to left it demonstrates returning back down the stack (pushing and popping respectively).

As mentioned in the previous section, the current frame's associated bookshelf is positioned on top of a conveyor belt; this allows bookshelves to be moved across the memory barrier and into or out of the current user scope. This mechanism is important for method calls and returns: the conveyor belt has two control buttons, the call button, and the return button. These two buttons move everything on top of the conveyor belt either toward or away from the user space. So far it is not hard to imagine the moving of the conveyor belt as matching up with the actual process of pushing and popping frames to and from the stack; the tricky part of designing this section of the metaphor set is considering what happens to frames that have been popped off the stack, and where new frames come from. Both situations can be explained in a similar way to overwritten variable values: For popped frames and overwritten values the object in question no longer belongs anywhere and so must be destroyed, the paper value would be torn up or burnt, and a similar thing would be done to the popped off frame.

In the same way that paper values are prepared using notepads, it was decided that a workbench could represent the creation of a soon-to-be-used frame:

After getting the student to name the method they are preparing to call, one has several options for presenting the steps of method preparation., The most feasible two techniques are as follows: the most initially-intuitive way is to ask users to name and assign parameters and arguments in much the same way as local variables (one at a time, with types, name, and values all the explicit responsibility of the user). The second possible presentation method provides more scaffolding for the user: they are given a selection of available method signatures which they need to pick from, and then are provided with all the parameters ready to receive values. The user simply slots argument values into each place. Figure 4.8 compares these two method preparation techniques side-by-side. Section 4.3.7 explains how and why technique one was originally used, but was later replaced by technique two.

For a void method no further explanation is really required for how the user leaves the method (they simply press the conveyor belt return button). However, it might need saying that a value-returning method works by simply making the user hold the return value in their hand, that way the return value is in hand when they get back to the previous frame, and thus they can use the value straight away. Student understanding of this particular subset of metaphors seemed particularly good, as discussed in further detail in the usability test results in Section 5.4.

Figure 4.8: Left: Metaphor one for method preparation, where the user is responsible for everything. Right: Metaphor two (signature-picking metaphor) for method preparation, where the user is responsible for arguments and calling but nothing more. The stages from top to bottom are: no method named, a method named but no parameters assigned or declared, one parameter (a) named and assigned, all the parameters named and assigned with the method ready for calling.

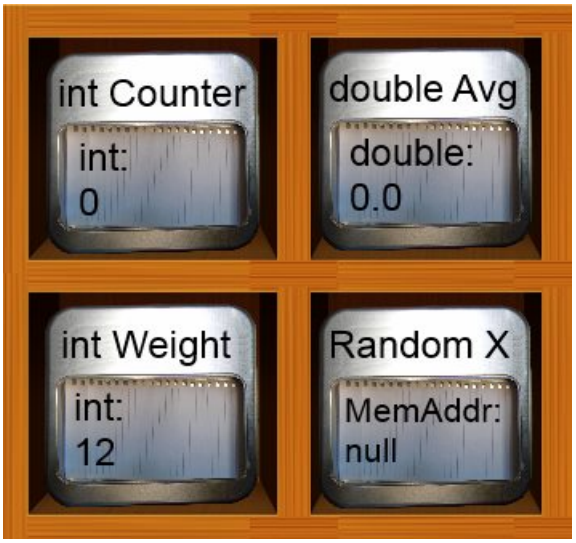


Figure 4.9: A simple fixed-size bookshelf that holds the global variables. It is directly accessible to the user.

4.1.7 Global Variables

Many of the more modern programming languages do not, strictly speaking, have *truly* global variables any longer - instead they have class fields. So in this context global variables refer to class fields of whatever context the user is currently in (although this metaphor would still be valid when applied to more legacy languages). Firstly, because global variables are always accessible, they need a representation inside the user accessible space. Secondly, because they (often) exist almost from the very start of the program the user need not be able to create more of them (unlike local variables with short lifetimes). These two facts can be brought together with another bookshelf metaphor: a fixed size bookshelf that has all the necessary variables already declared, but which is not part of the conveyor belt stack. Aesthetically one can represent this sort of bookshelf in more than one way, the proposed representations are shown in Figures 4.9 and 4.10.

A potential concern for some might be that globals should be created and initialized explicitly much like locals, but the disadvantage to this is that code execution would have to begin outside of any methods (something one might prefer a beginner not to do). This is another example of an implementation specific preference: if one does use globals, they are automatically provided for the student as scaffolding.

4.1.8 Representing the Heap

The easiest thing to represent about reference types is what exists in the local scope: a simple memory address written down as a value. Reference types were represented this way to make it perfectly clear to end users that whatever they do to objects or referenced variables goes through a memory address because the thing in question is not local. Additionally it makes explaining aliasing much easier, for example, if the user is presented with code such as:

```
object x = new object();
object y = new object();
x = y;
x.changeSomething();
```

When using the visualiser, users can see clearly that x and y are initially different, then y became just another name for x because their values are now the same memory address (and its original value was lost), and then finally, changes to one change the other. The more challenging thing to represent about reference types is what gets stored in memory (which the user can partially see over the memory-barrier). Without introducing a middle-man or some other go-between mechanism there is no way for players to affect what exists outside of the user space. To solve this issue a robot to represent the memory manager was introduced, which is elaborated on in Section 4.1.9. For now it is sufficient to say that the memory manager robot obeys instructions from the user, and carries them out in memory space - taking values from the user and writing them to the heap, or vice versa, depending on the instruction being carried out.

With the memory manager robot ready to interact with the global space on the user's behalf, the next requirement is a way to represent the heap and access to the stack. Access to the stack is fairly simple as a concrete metaphor for the stack itself has been established (the rows of bookshelves): the memory manager robot simply takes the address he has been given and goes between the bookshelves to interact with the appropriate variable box.

The heap requires more thought. After considering that the size of the heap is in fact finite (determined by the available memory) in an actual computer, it was decided that the heap metaphor could also be represented by a finite, fixed-size structure. The easiest

Figure 4.10: The heap bookshelf with two objects in it. If they were array objects they would be of length 4 and 6 respectively. This heap can only contain 36 simple values, and thus demonstrates how a compressed representation would be desirable.

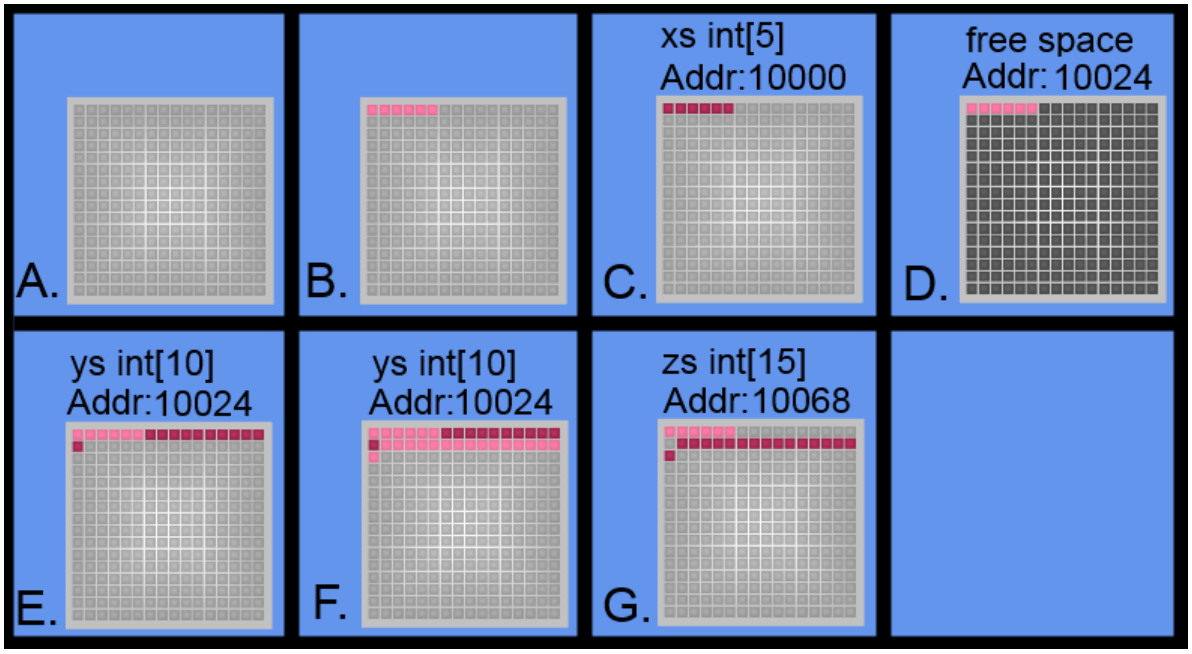


Figure 4.11: A compressed representation of the heap. A: an empty heap. B and C: one array present on the heap, with details above. D: highlighting the free space. E: three arrays on the heap, with ys being highlighted. G: a demonstration of what might happen after a garbage collection when ys is no longer being used.

way to represent it without deviating from the existing metaphors is to have one super-sized bookshelf, possibly with movable dividers, where unused space is represented by the lack of a variable box. Figure 4.10 shows an example of this extra large bookshelf, and also illustrates that in certain media the bookshelf representation falls slightly short due to its space requirements. For this reason, in-game, the heap can instead be shown as a compact series of squares (as shown in Figure 4.11). This alternative representation can just be thought of as a compact version of the bookshelf shown in Figure 4.10. Objects on the heap do not, strictly speaking, need labels other than their memory addresses, however they do make it more clear to the user what is where.

There are some disadvantages to the compact representation, the primary one being that the representation is not based in the real-world and therefore is likely to be less relatable. If one is concerned about distinguishing between different objects, colour can be used to emphasise the area used by a certain object.

Representing objects that are more complex than arrays of values is where reference types become particularly difficult to represent: one cause of this is the wide variety of 'shapes' that objects can take, a secondary issue is that of constructors and when instantiation actually occurs, and a final hurdle is that of objects having their own methods

Figure 4.12: A small section of the heap bookshelf showing potential representations of two non-array objects.

and associated code.

A simple non-array object such as a Random generator could be represented on the heap as two numbers and something to point to the objects associated method code: in this example the numbers would be the start Seed and the previous output (as pseudo random numbers usually rely on previous output), however the instance's methods would have to be seen only in the code that the students are following (thus risking a disassociation between instance and method code). Another example would be a bank account with a name, number and balance.

Figure 4.12 shows a potential representation of the heap with an instance of the Random and Account classes, unlike Figure 4.10 the objects on this heap have their properties labelled so as to make it clear to users that they are not just arrays of numbers. The 'name' property of the Account class would be a string, its contents on the heap would depend on how one chooses to represent strings, as explained in Section 4.1.10.

The compressed representation of the heap is not particularly conducive to anything more complex than simple arrays (simply due to labelling limitations), while the more explicit bookshelf representation would allow for objects to have other objects as properties: the object property would simply contain a memory address that points to another object on the heap (however this could be included in the compressed version, the resulting visuals would just not convey everything clearly).

4.1.9 Interacting with Objects and other Reference Types

Once one understands how the heap and various reference types are represented, communication with them becomes fairly simple to understand as well. The memory manager

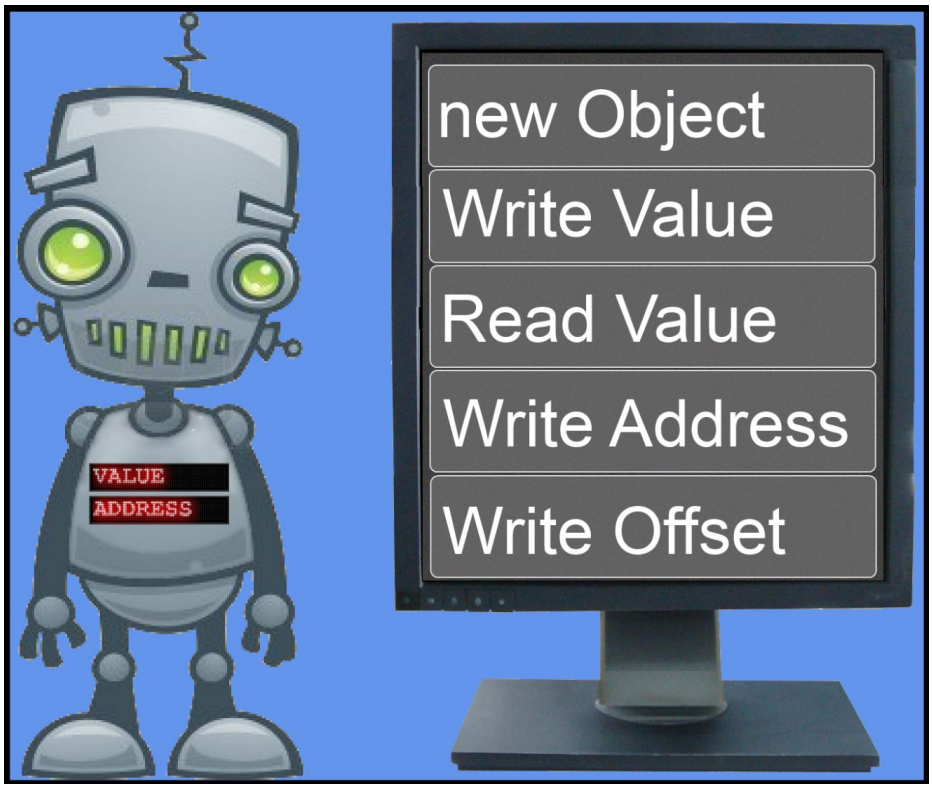


Figure 4.13: The memory manager robot (left) and the terminal used to communicate with him (right). These images of memory space communication metaphors were geared towards array communication, in the event of more complex object "Offset" might be replaced by "Field name".

robot is responsible for all interactions in the memory space, he receives his instruction via the terminal (which is located in user space). For writing to the heap values go from user to terminal to robot to memory address (and the other way for reading from the heap). In order to access data on the heap the user needs to provide the memory manager robot with two things: the object's base address, and the offset of the specific object-element of interest. For non-array objects the 'offset' would instead be the name of the field or property in question. Reference access to variables on the stack is done in much the same way, except there is no need to provide an offset - an address alone is sufficient.

In order to facilitate easier state recognition the memory manager robot can have several ways of showing whether it currently has an address, offset or value: either lights on its torso can show up saying what it has (as shown in Figure 4.13), or it could hold visible values. The robot and terminal do not require much explaining, however coming up with them in the first place took several iterations. The same is true of the stack and the heap, Section 4.2 contains more details about earlier unacceptable metaphor versions and why they were rejected.

4.1.10 Strings

Strings deserve a special mention regarding their representation, this is because while they are technically objects, their immutability means that treating them as value types is unlikely to cause problems. For this reason it was decided that strings could have two representations in the metaphor set, depending on the preference of the teacher: one can either use the more accurate (but more bulky) option of treating them as objects on the heap, or they can be treated as value types that reside on the stack. It is important to note that while the system allows for multiple representations, neither one threatens the fidelity of the metaphor set as a whole.

If one chooses to represent strings as objects on the heap then they would be treated as arrays of characters, with the special property of being read only after their initial declaration (due to their immutability). When they are treated as value types the metaphors do not break down: when one assigns one string variable to another one can treat them as though they are copies rather than aliases. The following simple piece of code illustrates how, while they are objects, their assignment to each other does not link them when alterations are made later. It also shows how, just like normal value types, one cannot change an isolated part of the value - if one wanted to make a change, one would have to replace the whole thing (the last line shows this).

```
string x = "Hello";
string y = "world";
x = y;
y = "hello " + y; //notice how x does not change, despite the alias
y[0] = ‘H’; //illegal as it’s read only
```

As briefly mentioned in Section 4.1.8, objects on the heap that have string properties would have different representations based on the choice of string representation: if one uses the more accurate model with strings as objects, then string properties would contain memory addresses, and the individual character arrays would exist elsewhere on the heap as seemingly separate objects. If one uses the strings-as-value representation then string properties become simple, with the strings clearly being inside the parent object.

4.2 Early Developmental Stages - how to Represent the Game World?

This section describes the earliest forms and representations that were proposed (and almost uniformly rejected) for visualisation of the system as a whole. The simplest concept one might try to represent, before even considering persistent data, is how to demonstrate flow of control. As flow of control structures are the most rudimentary concepts in code, this is the first thing that this work attempted to visualise. Early examples of the proposed visualisations were built around representing programs as mazes, and included:

- Representing a Switch-block as a corridor with a certain number of doors, of which the user can only enter one.
- Simple conditionals that rely on a boolean (such as loops and ifs) could all be shown as forks in the maze.
- Method calls required ladders that took the user up or down the stack into a new scope (and resulted in the maze becoming 3D).

Representing the system using the aforementioned maze structures seemed simple to begin with, however all possible representations require that users have some way that they can access local and global variables (which complicated matters when representing the

program as a maze). The author attempted to address this by making users carry local variables in their backpacks, which are put down when any method-ladders need to be climbed. Globals were more of a challenge: even when control enters a new method or scope, the same globals still need to be present. A possibility was to represent globals by a second backpack which users never leave behind, but this presents a problem when trying to carry around objects that belong on the heap... one could carry references to the objects, but how does one access and alter them? A portal-gun/teleport mechanism was proposed, that took the user into 'global space' and allowed them to handle objects and reference types on the heap... but then one must ask what happens to primitives that have been passed by reference (their actual values end up sitting in a backpack near a ladder somewhere and are inaccessible).

The issue of changing scope is further compounded when one considers that methods always have a single entry point, but can have an arbitrary number of exits (returns): how does one represent a ladder that has only one path up but can have any number of paths back down? Even if one can imagine something like this, one must consider how congested and confused the maze around the return ladder (or ladders) would have to be. It was at this point that the maze representation was scrapped entirely, and instead the previously proposed idea for global space was examined and altered to see if it could be used as a replacement for user space.

The result of altering the aforementioned global-space idea, was an open-plan area that the user can move around freely in, which contains:

- Visible code.
- A section dedicated to global values.
- Another for locals.
- And several control structures that are used to govern flow of control and data alteration.

This new arrangement removed the need to represent code in a 'walkable' structure (such as a maze). It instead allowed flow of control structures (such as conditionals) to be represented as value-consuming mechanisms which the user can interact with, the BoolEater, which is present in the final system.

With this 'playground' established in its most basic form, experimenting with certain behaviours began, the first of which was what happens during a scope change (this was

the first thing tested because it caused the most problems in the maze representation). Without some sort of enhancement the simple playground could not handle scope at all; to address this an 'elevator' mechanism that shifts the playground up and down while leaving certain things behind, such as local variables, was included. This meant that the user could change scope and *see* non-local variables, however accessing them as references was still troublesome.

To address the values-as-references issue, the arrangement of the playground's content was altered, resulting in a section dedicated to the stack, which could only be accessed via a communication robot. By now one might be able to see some similarities with the final arrangement.

The stack robot resolved the reference issue and allowed the work to progress, however before moving away from issues regarding scope, consideration was given to how globals (and the heap) should relate to the stack...after all, they all reside in memory. This led to the realisation that all objects that belong on the stack should be accessed in much the same way as references to stack variables. This was addressed by creating a heap structure in the memory space that is accessed in much the same way as references to the stack. At this point the robot stops being a stack robot and instead becomes the memory manager robot.

Now there is a clear segregation between the memory and the user space. At this point alterations became more about how best to represent a particular concept, rather than "how should the system be represented as a whole?" Discussion of some of the more interesting concept-specific alternatives can be found in the next two sections.

4.3 Pre-Optimisation Metaphors and Functionality

This section explains the state that certain metaphors were in when presented to students for hallway-testing (Section 5.4), and the improvements made based on the results. The metaphors usually appeared perfectly fine during development (to the experienced programmers involved in alpha development) but fell short for some reason or other once presented to novices. These short falls have various reasons, including: the fact that the developers were no longer able to see things from an 'uninitiated' student's perspective, and that simple aesthetic issues were making things unclear. The perspective-induced issues lend further credence to what the literature in Section 2.1.4 says about troublesome concepts being irreversibly transformative.

4.3.1 "What am I supposed to do next?"

The most common observation during testing was that student did not know where to start, or what portion of a line of code had to be done next. This issue was not due to a fault with the metaphors, but rather because novices often simply do not know where to start on a given line of code. The response was to implement a more powerful code highlighting mechanism; the original simply highlighted the entire line, while the enhanced version highlighted exactly what had to be done next. For example, the line "`int x = meth(2 + y)`" would have sections highlighted that correspond to:

- `int x`
- `meth(`
- `2 + y`
- `meth(x+y); //call`
- `x = //on return`

4.3.2 The In-Game Hand's Function is Unclear

When representing the metaphors in test games as manipulatives, it is necessary to replace the student's real-world hand with a virtual one. This in-game hand is responsible for letting students know exactly what they are currently holding (if anything), and is essentially a stateful cursor. The first representation of the users in-game hand caused issues because its state changes were not obvious: originally it always had a notepad in hand, and it was up to users to decide if there was a value on the pad. To address this, three hand states were introduced: empty, holding value, or interacting with another metaphor that requires more than just a simple click (for example, user input). These three states (as shown above in Figure 4.1) helped users realise when they were carrying (or inputting) a value.

4.3.3 Never-Empty Variables

The very first representation of the variable box presented to students attempted to avoid the issue of variables never being truly empty, by hiding the content until the user assigned

to the variable. This led to issues when comparing the order of operations for "`int x = 1`" vs. "`int x; x = 1`": in order to hide the content of x in the former case users were expected to evaluate the expression before declaring and assigning in a single step.

This issue was addressed by automatically giving variable boxes a default value for their content, two alternatives for this content exist: either "???" to represent that it hasn't been assigned to, or whatever the actual default value would be (for example, 0, 0.0, "", or null). Use of the "???" representation was decided upon, so as to avoid risking the misconception that a local variable can be read from before it is assigned to. The default values of object on the heap are also hidden, but that is more to do with heap communication being represented through the memory manager robot.

4.3.4 Expressions Overwhelming Students

In Section 4.1.3 above, it was mentioned that there are two ways users can handle expressions using the metaphor set: either via an in-game expression calculator, or as a simple answer notepad, with calculations done manually outside of the game. This section expands on the experiences with the calculator during the testing phase and how it was improved upon.

Generally when users were presented with an expression that had an operator in it, they seemed unable to decide what needed to be done with it. Once given a hint that the calculator should be used for expression evaluation, users got stuck with the order of metaphor interactions that they needed to undertake:

- Some would start by trying to mark a variable for substitution, before reading the value into the hand from the stack.
- Some would attempt to substitute values in an arbitrary order.
- Others still couldn't grasp what they had to do in order to substitute the value they had read (and which they were holding) into the expression.

Despite hints and attempted optimisations, the majority of users still had trouble with the calculator, thus it was decided that the whole mechanism should be replaced by a simpler one: moving the expression evaluation outside of the game.

Near the end of development an improved hint-system was implemented that highlighted the next object users needed to interact with, rather than just giving a textual hint. In fact it could go so far as to hide or dim objects that were not relevant to the current level in order to minimise clutter and information overload. While this final hint mechanism was not tested on users, its blatantness may be enough to overcome the aforementioned confusion around calculator interactions for any future work regarding this metaphor.

4.3.5 Conditional Structures, and Boolean Value Disposal

Because the metaphors are more concerned with concept representation than flow-of-control-structure representation, the final metaphor set has a mechanism which accepts a Boolean value from the user and then controls the code based on the resulting value.

However in an earlier version of the metaphor set, before introducing the BoolEater mechanism, users were having trouble knowing what to do when they reached a conditional statement. The original way conditionals were handled in the test games, was that the user needed to evaluate the Boolean expression, and then ignore the now-held value (the code would automatically advance based on whether the acquired Boolean was correct) - users almost uniformly failed to realise this.

By combining the aforementioned code highlighting with a new metaphor, that only showed up when the user needed to give the computer information regarding the outcome of a conditional, the intuitive nature of conditional structure operations thus was improved. This particular issue was a combination of both aesthetics, and a missing element in the metaphor set.

4.3.6 Unclear Scope Division

When using the original barrier between user and memory space (the barbed wire fence), two concerns were raised: the first (less important) one was that the barbed wire was unnecessarily intimidating; the second concern was that its purpose was not intuitively clear. By altering the barriers image to one of police tape, with a visual message stating clearly what it is and why it's there, users stopped trying to interact with things in the global space. This issue was further addressed by the inclusion of a mouse movement limiter: no matter what the user did, the in-game hand could never cross the barrier.

These two alterations together, one to the representation of a metaphor and one to the game itself, solved the issue of unclear scope. It also serves as an illustration that a theoretically sound metaphor might not live up to its potential if implemented poorly; in short, appearances can and do matter when working with actual students. However, as explained in the Section 3.5, the primary focus of this work is on the theory, while the more implementation specific details are left to future testers. These simple aesthetic differences, and their implications, are similar to those explained above regarding the states of the in-game hand.

4.3.7 The Method Mechanism

The metaphor which benefited most from optimisation through user testing was that of the method calling mechanism (specific user test results can be found in Appendix C, while a summary of the process and results can be found in Section 5.4). As mentioned above there are two primary ways in which one can present users with the metaphor for method calls: one can either make users responsible for *everything*, or just for assigning parameter values. While the metaphor itself was not flawed, the pre-optimisation presentation mechanism tended to be at too-low a level of abstraction, and tended to overwhelm students: either they had no idea what needed to be done, or they would try to perform operations out of order. This is understandable as there were no clear instructions saying "declare the first parameter, then find and assign its value, and now move on to the next one".

Not only did the original representation stymie most testers, but leaving this mechanism as it was would run the risk of giving students an inaccurate mental model, where method signatures are allowed to be arbitrary or dynamic. One should note that the aforementioned confusion was caused by the way the metaphor was presented, rather than the metaphor being unclear - and after improving the presentation, users understood the metaphor much faster.

4.4 Alternatives and Potential Enhancements

This section details some ways that the metaphors could be improved upon, or reasonable alternative representations. As explained in Section 3.5, the focus of this work is on functionality and theory rather than aesthetics, therefore no purely visual enhancements will be elaborated on here (after all, almost every one of the metaphors could have a nicer drawing made for them).

4.4.1 Expression Enhancements

Expressions that contain method calls should be the first thing one enhances: as the system stands, an expression such as "`int x = 4 + nthPerfectNumber(6);`" is representable using the calculator metaphor, though it isn't as intuitive (plus it would not work at all in the games themselves, because of the expression-substitution mechanism the API uses). Doing this sort of thing with the notepads instead of the calculator would not be possible either using just the metaphors or in the actual games. This shortcoming has been well addressed and carefully considered, and it was determined that the benefits of trying to include a mechanism for this scenario were outweighed by the difficulties associated with doing so:

- In order to properly represent this situation in a clean and understandable way (in game) one would have to perform several deep alterations to the API.
- If one were to alter the API using the less-complex alternative of asking the user to specify exactly what is replaced, the resulting behaviour would likely confuse users.
- There is no limit to the complexity that one can build into a single expression, and so a line has to be drawn somewhere regarding what the system can and cannot represent. Here are several legal examples that one might encounter, but which novices arguably do not need to understand:
  - `int x = nthTriangle(nthPrime(fact(y)));`
  - `x = f(m(3) + p(y - 2), b(x * Math.Pow(x + 1, 4))`
  - `return n <= 1 ?  1 :  fib(n - 1) + fib(n - 2);`
- One can avoid the issue entirely by breaking more complex expressions up into simpler ones, even when the expressions are recursive (such as the horrendously inefficient recursive Fibonacci number finder above).
- Breaking long expressions down into a sequence of simpler statements has some advantages: debugging and single stepping is easier, and it mirrors what happens in the background of a real compiler (users normally do not see the temporary variables, or the placement of temporary values on the stack).

This situation highlights a potential shortcoming with the use of notepads for expression evaluation, and is one of the main reasons why the final metaphor set includes both the

calculator and notepad analogies - when expressions become too complex for students to work out on paper, they can revert to using the slower but more powerful calculator. Fortunately this shortcoming is more to do with implementation limitations than with the underlying metaphors.

4.4.2 More Distinct Variable and Value Types

One of the simplest extensions to the metaphor set is that of improving type differentiation: at the moment there is no distinguishable difference between the variable boxes and paper types associated with various value types, aside from having a textual label. The textual label might not be enough for some students, and so an easy enhancement would be to include a better differentiator. Option for this enhancement can include the use of distinct colours, shapes, sizes, and more, to help users distinguish types from one another, for example:

- A boolean type might be represented by a piece of paper with true *and* false written on it, but with just one circled.
- A double might be written on a piece of paper divided in two (one half for decimal digits and another half for the integer portion).
- The size of a piece of paper could be made proportional to the number of bytes associated with a type (a short would use a smaller piece of paper than a long, but a larger piece than a byte).

There are such a multitude of potential representations to choose from (that are all still based around writing on a piece of paper) that one cannot claim a single one would be consistently better than the rest, let alone list all the possibilities. For this reason, this potential extension was left up to any future users of this system so that they might choose the representation that best suits their needs (this work simply lays down the foundation).

4.4.3 Non-Metaphor Heap Alterations

The metaphor used for the heap and objects is sound on its own and can in theory handle more complicated objects, however as the API and current implementations stand they

are only able to handle simple array cases. An example of a simple yet non-trivial object on the heap might be a random number generator, which keeps track of a single number (either the seed or the last unaltered number it generated), and pretty much nothing else at a field level. This example object would be stored on the heap and a reference to it would exist on the stack - when trying to access any object fields the user would simply need to let the memory manager robot know what part of the object they want access to (along with a pointer to the object). The proposed analogies stand up in this case, but the API and game implementations do not (they were intended originally as a test framework for the metaphors rather than final deliverables).

4.4.4 Potentially Representable, Non-Novice Concepts

There are several concepts that were deemed too advanced for novices, and thus no attempt to incorporate them into the metaphors was made. With a little tweaking, some of these concepts could still be represented using the proposed analogies. For example, enumerators and IEnumerable are a non-novice concept that the metaphors would be able to represent with a bit of alteration: strictly speaking, enumerators are built around instances of the IEnumerator class and therefore inclusion should (strictly speaking) be done on the heap, however it is easier to explain this concept by relating it to regular methods. One could represent enumerators by allowing users to either dismantle stack frames (for normal method calls), or put them to one side so that they can be returned to later (for enumerators).

This is just an interesting example of how potentially extensible the set of metaphors is: even when including more complex concepts, they can be fitted in without breaking the fidelity of the existing metaphors.

Most users take for granted the way that the computer accesses specific objects in a collection of some kind (whether it's a simple array, a list, a dictionary, or a hashtable) - if one were interested in demonstrating to students the way these collections differ under the hood, one could include an extension that animates the process of accessing memory on the heap (to whatever level of detail they consider appropriate), or a simpler way to do this would be to present the user with the actual code behind these structures and ask them to perform the lookups (whether it's through relative index addresses, binary searches, or linear searches). This sort of thing would give the end user a greater appreciation for the often-subtle differences between collections, however it is not recommended for use on novices.

Another example of a more advanced programming paradigm, that the metaphors could be altered to represent, is that of event driven programming[2]. The user would perform the same operations that they already do, but an additional metaphor could be included that asks them to handle an event: users would have to finish what they are doing, go to the event queue metaphor, and deal with the event as if it were a method call.

One final example that would allow for a multiplayer mode, is that of threads: each user would be on their own thread, but would have a shared global space. This sort of thing would be able to illustrate race conditions, locks, and shared resources. Very few alterations to the metaphors themselves would have to be done for this to work, the hard part is in the implementation of a system like this.

4.4.5 Flow of Control Given to the Users

A most welcome extension to the proposed analogies would be one that allows users to take more direct control of the flow of a program. After various developmental stages and early representations that unsuccessfully attempted to cleanly incorporate something like this (as explained above in Section 4.2), it was eventually decided that the computer would control code progression. This decision was made because all of the proposed visualisations for user-controlled control flow mechanisms became very messy, very quickly, however it would still be nice to extend the metaphors to allow users more control over the code than the small amount they are given by the BoolEater (explained in Sections 4.1.4 and 4.3.5).

4.5 Summary

This chapter explained the proposed metaphor set in its final form, detailed some of earlier stages that the metaphors went through, and finally explained some potential alternatives and enhancements. The biggest takeaway from this chapter is the finished metaphor set - not only is it one of this work's primary deliverables, but those metaphors can be used in a variety of system-independent contexts. For example:

[2]Event driven programming - when explained at a novice level - often glosses over details such as event handler parameters, what happens in the event of simultaneous events, or the main loop event listener and its associated callback functions. It is because of these details that events were deemed too advanced to be represented in this system, and not because they cannot be taught to novices (which is not the case at all).

- Teaching children to program, especially so due to children's responsiveness to visual stimuli [29].
- Assisting novice programmers who have difficulty dealing with abstract concepts (at least until they can overcome those difficulties).
- Any individual attempting to create educational programming media (including books, videos, or games) might use these metaphors to assist with visualising certain concepts.
- Individuals investigating the effectiveness of traditional vs. non-traditional teaching methods - for example, a class taught using these metaphors could be compared against one taught in a more traditional manner.
- Researchers attempting to create their own set of metaphors might take ideas from this set - they might also benefit from being forewarned regarding some of the described potential pitfalls (such as the counter-intuitive complexity of representing expressions).

The metaphors described in this chapter are referred to in the following chapter, which explains how the more technical deliverables (the API and test game) were designed and implemented, and how they incorporate the metaphors as manipulatives.

Chapter 5

Implementation

The previous chapter discussed the various design iterations and choices for the metaphors. This chapter goes into some detail regarding the implementation of the technical aspects of the system: the API, and the XNA game implementation. (The evaluation-oriented Windows Presentation Foundation (WPF) version is discussed in Section 8.1). This chapter starts out with several key design decisions, followed by details of the API. It then goes on to describe implementing the game on top of the API, and finishes with corridor tests and the improvements made based on those tests. Sample screenshots demonstrating sequences of in-game events can be found in appendix X.

5.1 Design Decisions

5.1.1 Register or Stack-Based Architecture

This question relates to how expressions and evaluation are presented to the user. In computer architectures there are essentially two competing execution models: to use an expression evaluation stack, or to use registers. The calculator metaphor in isolation of any implementation details is simple enough that one does not have to consider how users interact with it. But when one asks users to follow a strict sequence of operations in order to evaluate an expression, order begins to matter more.

The simplest solution to this problem is that of notepads and external user evaluation: getting users to do the evaluation externally, on their own calculator or notepad. By doing

things this way almost all the responsibilities of expression handling are sidestepped in this system. However, since the system kept its own calculator as an alternative expression evaluation mechanism, it is necessary to define how it works.

With the exception of the most primitive machines, register based systems require a minimum of two registers [57] - the in-game hand could act as one of them, however there is no space reserved for the second register (one cannot use the calculator as it is there to keep track of the unevaluated expression). A different, yet related, problem arises when trying to define the calculator as a stack-based one: the hand would have to no longer hold a value, but instead point to the top of the stack; this would complicate the representation of the stack by including more than just methods and scope.

An additional issue with attempting to accurately represent the system as purely stack-based, is that method arguments are typically prepared as part of the current frame, and some tricky call logic manipulates the stack so that the frame boundaries are repositioned when the call takes place. This lack of clear separation between the calling frame and the called frame could potentially lead to confusion, although it could be done with the current metaphors.

If the system were built as a purely register based architecture, all the metaphors and visualisations proposed so far for the stack and the local frame, run the risk of being invalid due to the change in abstraction level: registers force the user to perform all operation at an extremely low level (basically at an assembler level). For example, evaluating "x + y * 2 - z" on a register only system would require users to perform an over-long series of operations such as:

```
read reg1 y
read reg2 2
mul         //overwriting reg1
read reg2 x
add
read reg2 z
sub
```

The calculator metaphor is a useful and familiar abstraction that can sidestep the architectural complexities of registers and stacks, and simplify the user's notion of expression evaluation. By representing the calculator as an abstract hybrid one is able to overcome

the granularity with which users would have to interact otherwise. For example, if one were to try and adhere to either of the aforementioned architectures, every operation in an expression would require at least two distinct user interactions, making evaluation of something as simple as "1+2+3" a four step process (at least).

5.1.2 Level Generation and Storage

A game level must encapsulate two key things: the program code for the level, and some representation of what the code should do in the metaphor world. There are various ways one can design this:

1. Fully compile and run the code so as to compare final results with just the expected end state of the user's actions.
2. One could accept any valid program code, use reflection to compile it down to the .Net intermediate language (IL), and then interpret the IL to decide on the corresponding metaphor operations on the fly.
3. Create a C# interpreter, that directly interprets the original program code (not unlike Python, and similar in some ways to the previous option).
4. Manually annotate the code with some markup that describes what should happen in the metaphor world. The annotated code can be stored in a simple Extensible Markup Language (XML) format.

All of these options have potential drawbacks, some more than others. Using just the output of fully compiled and run code is the easiest possibility to both implement and eliminate, as it prevents users from knowing if or when they have gone wrong until the end of the level (which they may not even get to). The two interpreter options are highly complex, and would involve a great deal of additional work, plus they may not be conducive to offering constructive feedback to users. An ancillary concern associated with the first three options is that they limit the target audience to only C# users, while pre-made levels allow for different languages.

This leaves only one option, the creation of pre-made level files. The advantage of this option is that it could, in the future, be combined with one of the other options: the interpreters would simply output a level file in the already defined format, meaning the

game could work with that level object and not be altered to accommodate a new format. The disadvantage of the manual annotations is that level creation becomes more tedious, but the advantage is that the author can determine exactly what level of detail to expose the student to. In ActionWorld a a level editor tool removes some of the effort of level creation.

5.1.3 Text, 2D, and 3D Visuals

Originally development was to be divided into three phases: a textual game, a 2D game, and finally, a fully 3D game. The textual stage was implemented without any trouble (its primary goal was that of checking that the assorted API components were functioning correctly). The 2D version, which was implemented next, was meant as a way of testing both the metaphors and the API, as making changes to sprites (based on user feedback) is far easier than changing 3D models. Due to the number of changes made to the sprites' appearance and layout, the second stage lasted longer than anticipated. It was decided that the extra effort involved in creating the necessary 3D models and animations was not justified by the potential benefits.

5.2 The API

5.2.1 Calculator Component Functionality

Expressions are a large part of programming, and thus the API's calculator component needs to be appropriately powerful and accurate. This section details how and why it was made the way it was, as it is one of the most complex components of the API (one can see the actual 'position' the calculator takes in the API as a whole, by referring to Figure 5.1).

At times it is necessary to programmatically infer the type of whatever the user has entered, for example, when the game asks them for a float and they enter an integer. There are two main ways one can do this: the simpler method is to analyse the string they enter and infer the type from its format. This method works, but needs all sorts of bounds checking (a simple example is that of exceeding the maximum size of a given type), also it cannot be applied to unevaluated expressions.

Instead, this was an area where the newer more powerful features of the framework were exploited. The .Net framework provides the ability to generate source code for a C# class, compile the source on-the-fly, and then load the class and instantiate objects. Compilation on-the-fly is also useful for finding errors in fragments of code that the user might submit (although ActionWorld does not make use of this), for evaluating expressions and providing functionality for the calculator, or for determining the type of something that the user enters, as shown here. This powerful use of reflection and the CodeDom compiler allows delegation of the task of complete expression evaluation. This snippet shows a small portion of the system's RuntimeCompiler class, where user input is textually substituted into uncompiled source code (using string formatting). The resulting code is then compiled, and uses reflection to infer the type entered by the user:

```
object result = {0};
return result.GetType().Name;
```

The actual method is significantly larger, but it is these two lines that ensure correct type interference. Without using the run-time compilation, the user's string must be parsed to infer the type - and the associated parsing method with its exception handling and other plumbing is closer to 40 lines. The runtime compiler technique is also used to obtain the result of expressions: the only difference to the above code would be that `GetType().Name` would be replaced with `ToString()`, which would produce the result rather than the type of the expression. Catching and interpreting runtime compilation errors is more difficult when depending on on-the-fly-compilation - this issue was overcome by using an all or nothing approach:

- When users input their expression it is assumed to be correct.
- The compiler attempts to evaluate the expression, if it fails the expression is marked as unevaluatable.
- An unevaluatable expression is either caused by user typos, or from unsubstituted variables. In either case the user is asked to check for these two scenarios (distinguishing the two *should* be easy enough for the user).
- When no error occurs during compilation the result is returned as a string, with a corresponding type.

5.2.2 Final API State

As shown in Figure 5.1, all of the API classes hinge around one central control class: WorldTracker. Anyone making use of the API need not concern themselves with the other classes (unless they are enhancing the API or building levels without the specially made level editor). WorldTracker has too many methods to list here, so instead, only some of the more important ones will be elaborated on:

- PerformOperation is where user opCodes are sent and executed. The opCodes have to be assigned to each sprite by the current control class, but the behaviour for each operation is governed autonomously by WorldTracker. Some example opCodes include: "return", "garbageCollect", and "evaluateCalculator". A in-code example might look like this: `theWorldTracker.performOperation("return", assignCurrentFrameSpritesAsAction, freePlay);`
- A few 'get' and 'is' methods are necessary to display the game state to the user. Examples of 'get' methods include getStackSize, and getLocalVariablesCount, which one would use to decide how many frame and variable sprites to draw, respectively. Examples of 'is' methods include isLevelComplete and isMethodNamed.
- The undoLastOperation method is called if the the user wants to take back an action.
- Most of the remaining methods govern the behaviour of the virtual machine for individual opCodes. For example, the opCode "return" will result in returnFromMethod being called, and "assignvalue" will call assignStackVariableIO (which assigns the players held value to a given variable).

Value is another noteworthy class, it is as central to the virtual machine as WorldTracker is, but in a very different way: all user data in this system is stored as an instance of Value, thus any other class that stores or manipulates data will have to interact with Value instances at some point.

To clarify things, here is a brief summary of the responsibilities and functionality of the remaining classes:

There are only four classes that are ever instantiated more than once - Value, Variable, Frame, and HeapSector (all on the bottom row of Figure 5.1). Of these, the role of heap sector is probably the most obscure: A HeapSector is made up of one or more Values,

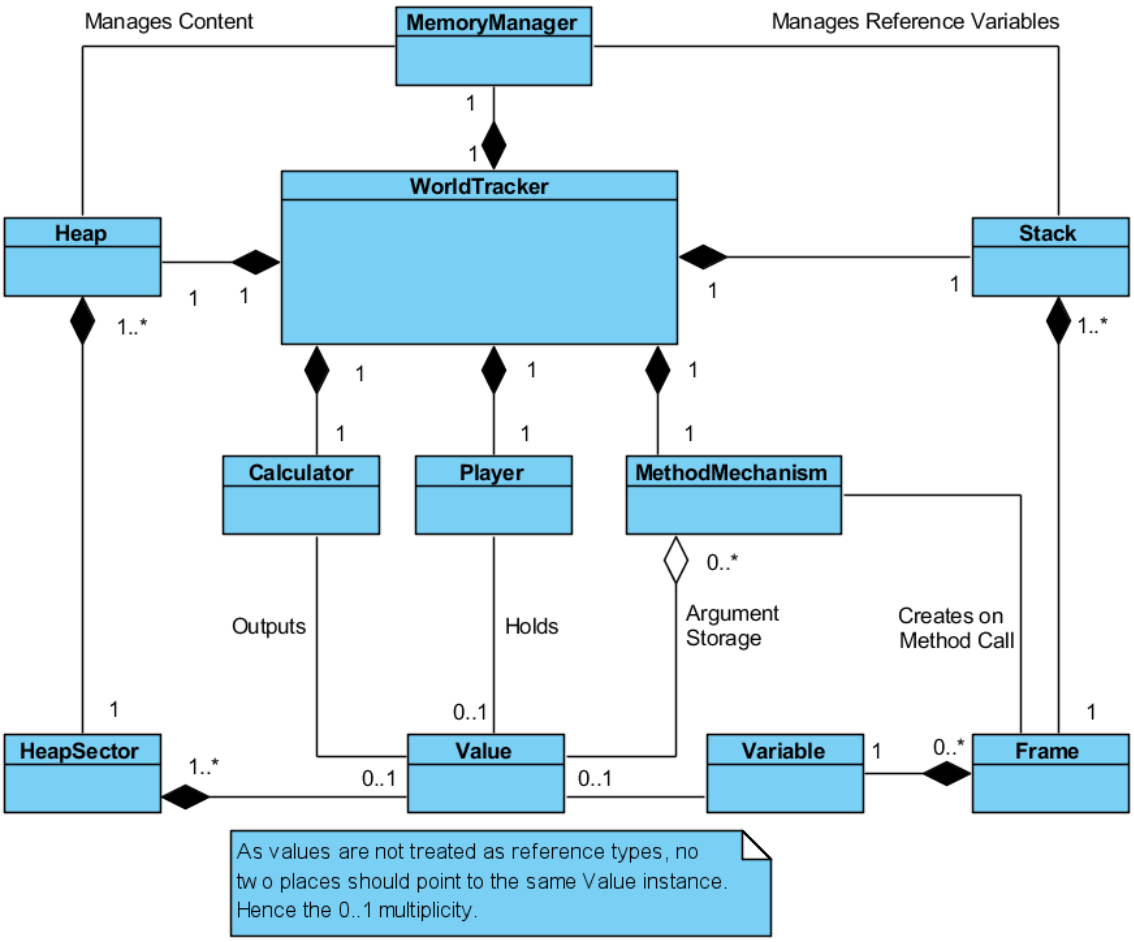


Figure 5.1: A simplified class diagram of the API. Note how almost all interactions go through the WorldTracker class in some way.

and in turn, the Heap is made up of one or more HeapSectors. A new Heap with nothing but free space has a single sector. When the user adds an object to the Heap, that one 'empty' sector is divided into two pieces - one stores the new object, and the other is left as 'free'. The heap includes two space allocation options (first-fit and best-fit), when placing new objects into a fragmented Heap, so as to potentially teach more advanced memory management concepts (not necessarily to novices).

The Heap class was built around the concept of a File Allocation Table (FAT): the Heap knows about the positions and sizes of each HeapSector (or cluster, to use the FAT equivalent term), but does not know what their content is. WorldTracker generally does not interact directly with the Heap, that task falls to the MemoryManager which takes memory addresses, and offsets in order to control how data is written to and read from the Heap.

The system's MemoryManager class keeps track of free and occupied regions in the Heap. Each region is represented by a HeapSector. The memory allocator will allocate heap space for storing objects, and a simple mark-and-sweep garbage collector will free up any heap objects that are no longer accessible from the user's program (this is either done automatically, or on a user instruction, depending on the mode).

The key responsibility of the Frame class is to store local Variables associated with each call of a method.

There are still three undiscussed classes: Calculator, Player, and MethodMechanism. All Value transfers go through the Player (for example, assignments, passing parameters, or return values). Player simply tracks what the user is currently holding, and is essentially a container that can accept any Value type as well as track whether the Value has been used or not. The tracking of Player Value usage is to prevent users from re-using a value, as the hand (a bit like a variable) is never truly empty (although its contents can be hidden from the user). The Calculator object is akin to a complex stateful arithmetic logic unit (ALU): it tracks expressions, performs substitutions, and outputs results.

Finally, a MethodMechanism instance manages the state of partially complete method calls: it tracks the name of the method, the names and values of it's various arguments, and generates a Frame when it comes time complete the call. MethodMechanism has two call modes, one where it is given the stack as an argument, and one which simply returns a Frame so that the WorldTracker can push it onto the stack. There is only ever a single instance of MethodMechanism at any one time, which is shared across different method calls. One could alter the API so that multiple MethodMechanisms can exist at

once - this alteration might serve to better track nested method calls, for example, `x = fun1(fun2(y));`.

5.3 The Game

5.3.1 Transitional Text-2D Stage

This was initially envisaged as a transitional stage between text-only, and the eventual goal of a 3D animated game: in this stage sprites would be visible and would reflect the underlying state of the system, but that was all they could do. If the user wanted to perform some operation it had to be done through what amounted to a debug menu. During this stage new sprite interactions were gradually added, and eventually came to completely replace the text menu interactions.

This stage simplified the process of migrating to a more graphical version, as any later additions could be compared against this stage's textual outputs. This allowed for quick identification of problems in the event that the results of two operations did not match one another.

5.3.2 PIP Magnifier

A picture in picture (PIP) magnification area was created that tracked the cursor, so that users could easily see more sprite details. By default, graphical-rendering targets and draws its output on the screen. By rendering a small region of the 'normal' output to an alternate render target, one can create an enlarged image and then draw the image to a portion of the screen. This magnification area made reading things easier, but possibly also hinted at poor user interface (UI) design (perhaps a well designed UI shouldn't need a magnifier in the first place.) Figure 5.2 demonstrates the magnifier.

5.3.3 Calculator State Machine

The first iterations of the in-game calculator required that the user specify whether they wanted to enter an expression, substitute a value into a variable in the expression, or evaluate the expression. A fairly simple solution to the problem was proposed that utilised

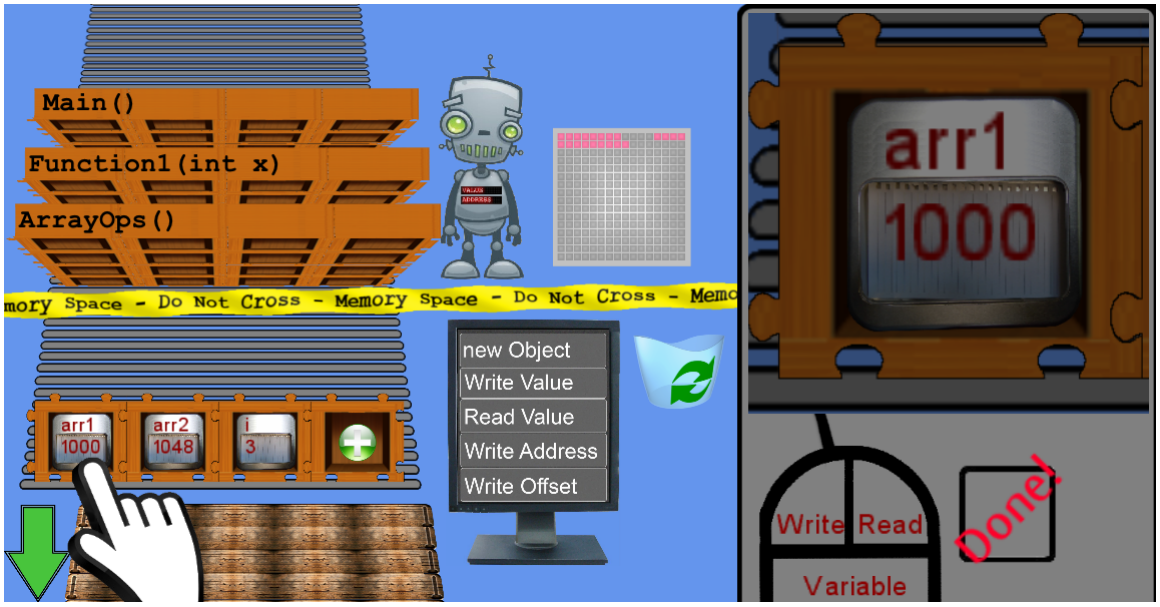


Figure 5.2: A compressed screen shot demonstrating the magnification of the region pointed to by the hand (top right), the call stack (top left), the variables accessible in the current scope (bottom left), and the heap with two arrays present (top centre).

a finite state machine, and 'test' expressions evaluations. Figure 5.3 summarises the calculator's finite state machine (FSM). By examining the state that the logical calculator is in, the calculator sprite can determine what it should display:

- When in "New" state, display the expression verbatim, do not have any printed paper showing, and let the user know that their next click will either substitute a value or evaluate the expression.
- When in "Simplified" state, display result on a printed value paper, inform the user that the next calculator click will move the content to their hand and clear the calculator.
- When in "Unsimplifyable" state, display the expression, and notify the user that they need to substitute variables before evaluation can occur.

A calculator feature that was not implemented, but which would have improved usability, is the ability to select individual components of an expression and perform component specific tasks. This sort of thing would allow for easier substitutions, term-specific hints, and even nested one line method calls. Unfortunately the benefits of implementing such functionality were outweighed by the complexity required to implement it (especially once the alternative of getting the user to do external evaluation was suggested).

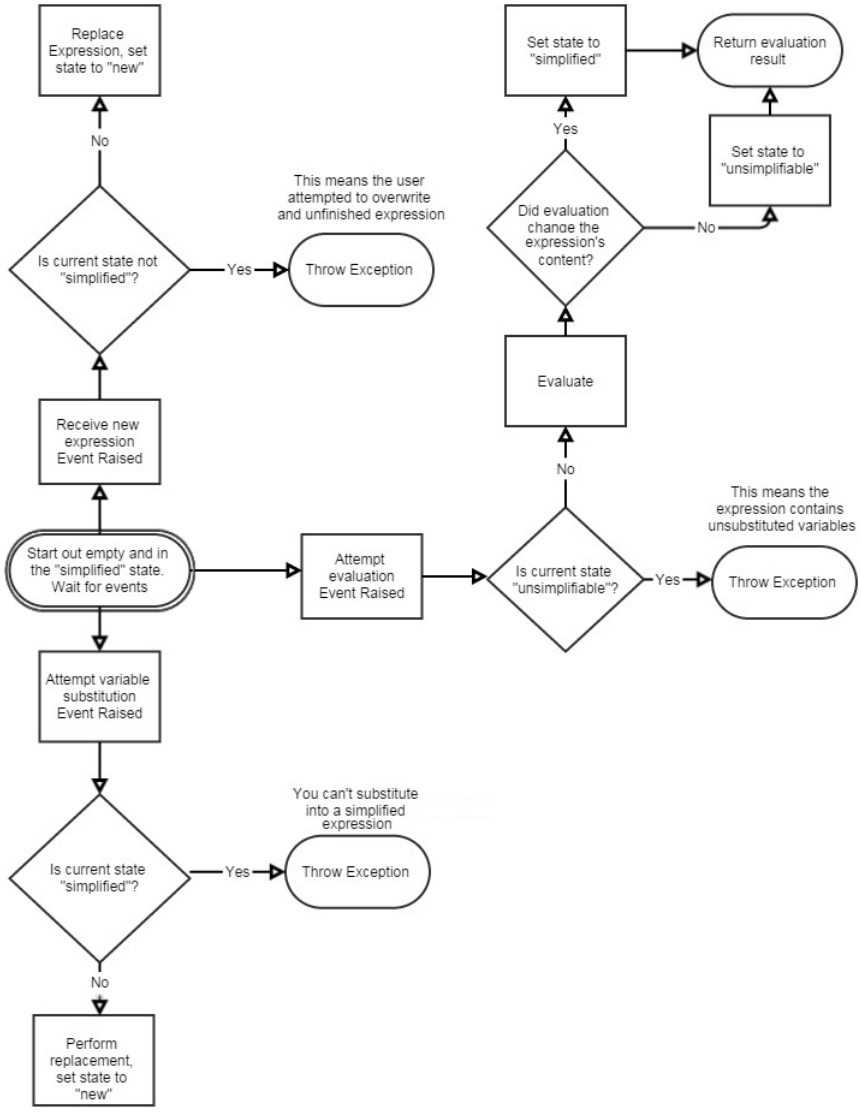


Figure 5.3: A visual representation of the logic used to make the calculator more intelligent, and thus more user friendly.

5.3.4 Bypassing the XNA Guide

The initial implementation was build using the XNA game engine, because this would provide the 3D functionality envisaged for the end product. But XNA does not provide widgets like buttons, menus, text boxes or scroll bars for easy input. Instead, they provide a type of collapsible "console" mechanism called the XNA Guide that can partially remedy this. As will be described in the next section, users found the asynchronous XNA Guide (shown in Figure 5.4) unfamiliar and an intrusive means of obtaining user input. To eliminate use of the XNA Guide, alternative mechanisms were needed for two primary kinds of input: menu selection from a pre-defined list of options, and text entry. For text input, the Guide was bypassed by including a cursor lock (which prevents users from using the mouse, thus forcing them to use the keyboard), and a finite state machine to track and accumulate what keys the user pressed. The input tracking FSM required that the character-case of pressed keys be determined manually (for example, if the user presses shift and 'a' at the same time the stored character should be an 'A').

The other type of input (selecting from a menu) was easier to implement: an auto-generated collection of clickable sprites is displayed, all background sprites are locked, and the clicked sprite calls a delegate with its content as the argument. For example, when selecting a method to load, all sprites would know to call the `prepareMethod` delegate with their content:

```
string[] methods = theWorldTracker.getTheCurrentLevelsMethods()
methodMenu.displayMenu(methods, prepareMethod);
```

Section 8.1.3 elaborates on how much easier obtaining input is when using WPF rather than XNA.

5.4 Hallway Usability Testing

This section details how the program was presented to users, tested by them, and then improved upon based on their critiques. The original plan for an evaluation and feedback mechanism was to simply give the program to anyone who wanted to take part in their free time and then automate the process of counting their mistakes and recording the results. This method was abandoned before it was ever used on students (but not before

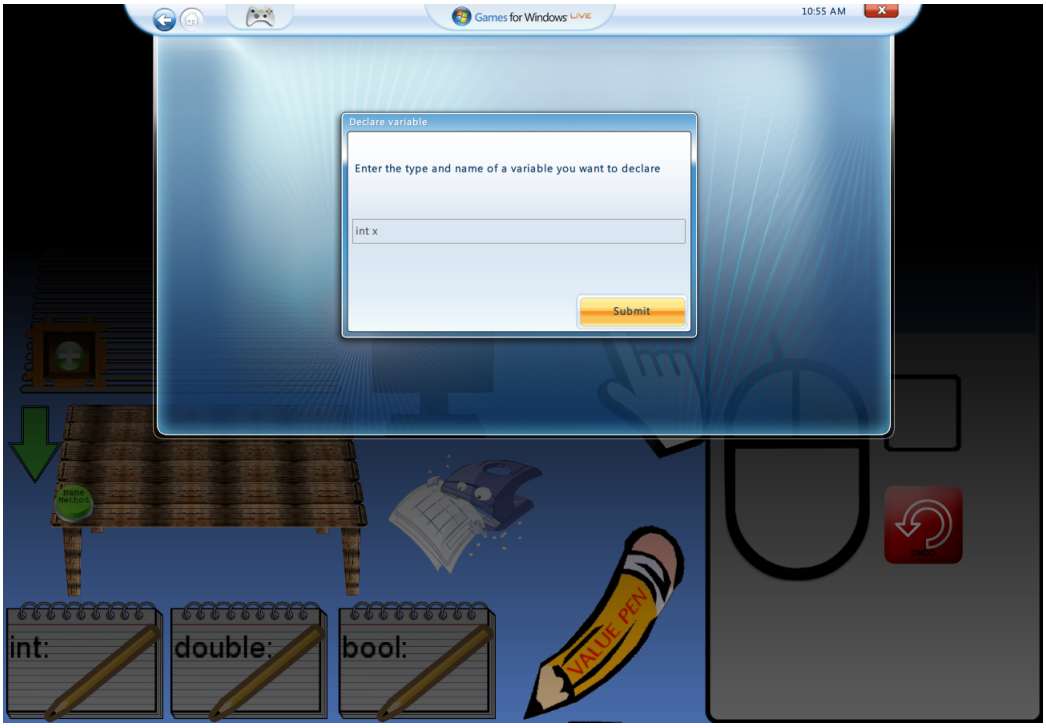


Figure 5.4: The XNA asynchronous Guide: an intrusive tool that obscures the state of the game.

a simple email-based version had been implemented), as it became clear that there were several weaknesses with testing usability in this manner:

- Any students who did not understand something might simply stop, thus rendering their test incomplete.
- Other students who got stuck for whatever reason might simply click around the game experimentally until something happened, thus invalidating any error counts.
- Only mistakes could be counted, and thus no concrete feedback about what tripped up students and why would be gained.

An alternative, less formal, usability evaluation technique is described by Spolsky [54]: "Hallway Usability Testing". This technique's name derives from the idea that one would simply pull a prospective user from the hallway and ask them to test a program quickly. This section elaborates on the process and outcomes of the hallway tests, starting with how test levels were derived, and ending with improvements made based on feedback. Hallway testing, by its very nature, can only be done in a semi-formal fashion, however this does not detract from the usefulness of feedback obtained in this way.

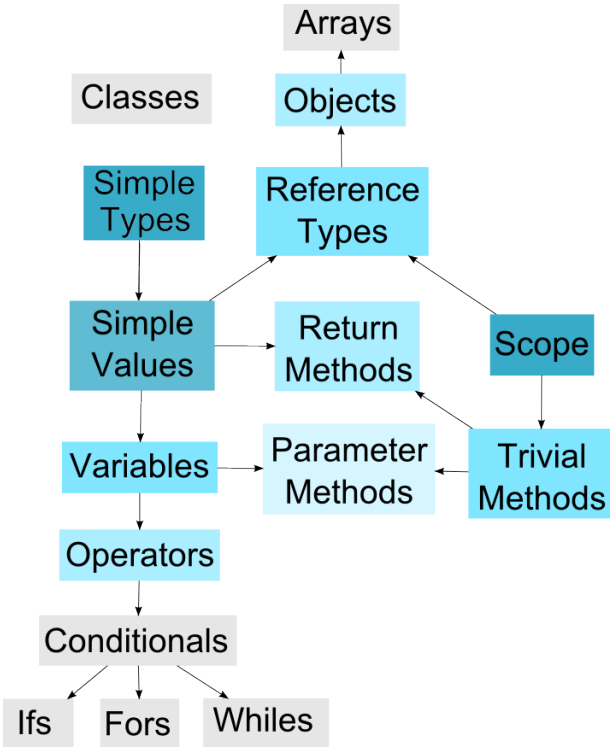


Figure 5.5: Fundamental Concept Dependency Graph. Illustrates what concepts rely on other, simpler, ones. Arrows can be interpreted as saying "This one is depended on by that which it points to". Classes have not had their dependency defined, as it is in fact unclear - this is because almost everything done in C# is within a class, even if the user does not realise it.

5.4.1 Devising Appropriate Test Levels

As the goal of the corridor tests is not to gauge the skill of the user, but rather the quality of the interface, the test levels need to be simple to understand while still having multiple metaphor interactions. It is also necessary to have more than one test level so as to allow for the evaluation of more specific concepts, as well as gradual introduction of metaphors. A simplistic dependency tree was proposed that facilitated decisions regarding the order that metaphors are introduced. The introductory order would always start with more fundamental metaphors and concepts, before moving on to more advanced ones. For example, a user that is being taught Boolean expressions should probably be shown boolean variables and values before conditionals. Figure 5.5 shows the dependency tree.

Using this dependency graph the following test levels were created, which are listed from simplest to most complex:

1. "Variables, Assignment and Operators" - Starts out with simple variable declaration

(type and name), moves on to expressions and assignment of values to the variable, gradually increasing the complexity of the expressions to include multiple operators.

2. "Conditional Branches" - begins with a hard-coded conditional (`if(3 > 2)`) that has no `else` part, and gradually increases the complexity by including `elses`, and more advanced conditionals (up to `x == y || x < 6`).

3. "While Loops" - a very short level with a single while loop. The loop contains two variable incrementing lines so that users can see that `while` loops do not exit as soon as their conditional becomes false (i.e. mid iteration).

4. "For Loops" - similar to the previous level.

5. "Introduction to Methods" - this level presents users with method calls that gradually increase in complexity, starting with trivial parameterless void methods, and ending with a function that requires multiple parameters.

6. "All in one" - combines cases from all of the previous levels, starting with simple variables and ending with method calls.

The final level proved useful for testing users who had prior experience, as there was less level loading required, but the metaphors were still introduced in a sensible order. This work's iterative methodology can be seen in the level designs: each level builds on the last, but starts out with simpler examples than the previous level ended on, resulting in a gradual increase in complexity and repetition of previously covered work.

The exact code for each of these test levels can be found in Appendix B.

5.4.2 The Test Process

This section describes the semi-formal procedure used during hallway tests, and goes on to briefly describe some sample responses. The comprehensive details of every participant's responses can be found in Appendix C.

Participants were first asked what, if any, prior programming experience they had, then the game and its purpose were explained. The tester would then open an appropriate test level - this depended on the participant's experience, as well as which mechanisms were in need of testing the most at the time. An example of this second step could be either

that this is the first set of tests, or that the student has no experience at all, in which case the simplest test level would be loaded (expressions, values, and variables).

During play, participants were encouraged to try and work out issues as they encountered them without help from the tester, while explaining their experience as they do so - this ensured that as soon as an issue appeared the tester could note it down and enquire about the participant's opinion on how it might be improved. Once a level was deemed complete, the tester would ask questions such as these:

- What seemed least intuitive to you?
- What mechanism did you find easiest to understand?
- If you could change one thing, what would it be and why?
- What would your biggest complaint be?

If a participant had no trouble by the end of a level, the next one would be opened and tested as well so that more information could be gleaned from the subject. These questions managed to elicit useful information not just about the metaphors, but about almost all aspects of the game. The next section covers some of the changes made in response to user feedback.

Hallway testing seemed very successful, with one particular example being the method mechanism metaphor: the first set of testers were all unable to call methods, because the interaction mechanism was so unclear at the time. By closely examining what they tried, it was determined that a more informative layout was needed, as well as a more intuitive call-initiation action. This led to a revised mechanism where users select the name of the method to call from a list, and then fill in the empty parameters as they get highlighted by the system. This ultimately led to one of the more intuitive sets of interactions in the whole system.

5.4.3 Important Changes

This section details *some* of the improvements made based on the hallway usability tests:

- The function and state of the hand needed to be made more obvious. This was done by displaying three different sprites based on whether the hand was empty, carrying a value, or getting input.

- Variable creation was originally done in a right to left, bottom to top order. This threw users off and was altered to be more in line with users' experience with reading (top to bottom, left to right).
- The calculator was perceived as *very* unintuitive, and multiple improvements were implemented to address the issue, including (but not limited to) a more intelligent calculator, and the ability to simply work out and write down the answer without using the calculator at all.
- A minimalist layout for new users was implemented to address the recurring issue of users being overwhelmed by the number of objects on screen. This layout came in the form of removing unnecessary sprites. Later iterations of this improvement included dynamic sprite highlighting and hiding based on the next expected operations.
- The initial barbed wire was replaced with a less intimidating memory divider, to reduce the risk that users perceive memory operations as dangerous, scary, or advanced.
- The XNA asynchronous Guide had to be replaced due to its intrusive appearance (it would hide the state that the metaphors were trying to show).
- More specific code highlighting that could identify sub-parts in each line of code was added to help guide users to perform operations in the correct order.
- The visibility of the code was improved, as users would occasionally not notice that it was there until it was pointed out to them.
- When users had to enter the details for a method call themselves, users were almost unanimously confused. However, with additional scaffolding (a menu allowing selection from a list of methods, with placeholders for arguments), users grasped what they had to do faster than with any other metaphor.
- An introductory screen was included to explain the purpose of the game, and the role of the user as interpreter (some users had trouble grasping that there was a connection between the metaphors and the code).

5.5 Summary

This chapter covered various details regarding the implementation of various system components - ranging from initial design decisions, to hallway tests and associated improve-

ments. Several components and concepts described in this chapter have more general applications. The virtual machine API can be used in a variety of contexts that need to track and update the state of a virtual machine (regardless of whether that machine is being used to teach, or makes use of metaphors at all). For example, if someone wanted to develop a genetic algorithm that evolved assembler code for a specific problem, the evolved code could be passed into the API for testing and visualisation.

Another example of a generalisable deliverable from this chapter is in fact a specific component of the API: the calculator class - built around a powerful combination of runtime reflection and dynamic compiling - has the potential to be used in several systems than need to evaluate user expressions as part of their behaviour (bypassing the need to create a custom lexer, parser, and evaluator).

Now that implementation details of the various system components have been explained, the next chapter covers how one might validate the quality of the aforementioned components, and the system as a whole.

Chapter 6

The Test Framework

The previous chapter described the implementation of various system components (such as the API and test game), this chapter focuses on how one might validate these components. Surprisingly, perhaps, there appears to be no consensus regarding the 'best way' to evaluate either educational systems, or programs in general. This chapter reviews the test landscape, and draw out several viable methods one can use to test this system.

Considering how modular this work is, an unusual method of utilising the following tests was proposed: rather than testing a system as a whole, one can instead decompose the system into its constituent components, and then use more targeted testing techniques. For example, surveys might be best suited to evaluating the metaphors, while additional game features or new front-end implementations would give further insight into the quality of the API. Henceforth this testing technique will be referred to as ***constituent evaluation***.

The purpose of the test framework is to take a final deliverable (one where the effort required to improve the system no longer justifies the diminishing returns) and to seek verification of the validity, fidelity, and accuracy of both the deliverable in question and the system as a whole. This work makes use of both quantitative and qualitative tests, and this chapter has been divided based on the nature of the described tests and what they are testing.

As testing and evaluation are such broad topics, no attempt was made to include all the possible ways in which one might evaluate this system. Instead the focus is on a select few (a comprehensive survey and explanation of potential testing techniques would entail writing an additional survey-type paper). As an example of the myriad number of potential techniques, simply consider a survey whose format and content will vary

depending on: what material is actually being tested, the level of detail one hopes to elicit from responses, whether the results are to be quantitative or qualitative, who the survey is targeting, the desired number of responses, and the available testing pool. If each of those factors has just two potential answers, one still ends up with 64 different sets of survey requirements.

The tests and samples explained and provided in this chapter are meant as formal means of evaluation, in other words their primary purpose is to determine quality. For this reason hallway (or corridor) tests have not been included in this chapter (or the next) as their primary use was to quickly and informally identify potential areas for improvement. Put differently, hallway tests aim to improve quality, rather than evaluate it.

6.1 Mark Driven Quantitative Evaluation

In most sciences, quantitative validation is usually preferable to qualitative evaluation. For this system, such an evaluation could be done by taking a group of novice programmers, splitting them into two groups, giving only one of the two groups access to the metaphors and associated game, and then comparing the final marks of both groups against each other.

One might partially alleviate the resource and time requirements of such an evaluation, by allowing all students access to the system and logging the amount each user uses it (as well as the way they use the tool). One might then draw a correlation between marks and usage data. The recorded details would include (among other things): error frequency, changes to that frequency, and what levels users used most. In order to mitigate errors occurring due to students not needing a system like this in order to learn (and therefore not using it as frequently as others, but still getting good marks), one might consider changes in error frequency over several sessions.

Due to time and resource constraints, this particular test was not used: it would take a *minimum* of three months to complete, require a fresh group of novice programmers, and (depending on the manner of implementation) an extra lecturer. Fortunately there are alternative quantitative tests with lesser resource requirements, including: checklists, rubrics, heuristics, and one-on-one comparisons.

Due to the nature of this test technique, it would be very difficult to include a test template one might follow while performing it, nevertheless it is probably one of the most valuable techniques that any future testers might use.

6.2 General Qualitative Tests

This section broadly describes the most commonly used techniques for gauging qualitative quality, in this context a general test is one that can be used on almost any of the system's components (or the system as a whole). The general techniques include: surveys, focus groups, and interviews. Some of the general techniques described here overlap somewhat with the more specific tests that come later.

When done in the right way these techniques can be analysed in a quantitative fashion. One popular mechanism is to have surveys that ask for user scores on a Likert scale. Although the questions are qualitative, the analysis can be quantitative.

6.2.1 Surveys

One of the most tried and tested means of evaluating almost anything is through the use of surveys. The usefulness of surveys varies depending on the survey itself. If one sends out a long, complicated survey to a large group of people and gets maybe four replies, one cannot really draw any general conclusions from that data [58]. However, if one sends out simple, concise surveys the number of respondents may increase, but the amount of information one can glean from the replies is likely to be limited.

There are several different formats that a survey for a system such as ours could take:

- Here are some metaphors, what do you think of them?
- Here is some code, which of these analogies corresponds to the various components of the code?
- Given this sequence of metaphor interactions, what do you think the user was trying to achieve?
- Please match the metaphor to the associated code action?
- When do you think this metaphor stops being accurate?

An in-depth survey was *created* that attempts to elicit detailed opinions regarding specific metaphors, which can be found in Appendix E. This survey was not actually used during the course of this work due to time constraints, regardless it can still be used, either as is or after alterations to better suit the tester's needs.

6.2.2 Focus Groups and Interviews

Surveys have their place, however potential participants will often look at a survey and decide not to take part because it looks too long, too complicated, or they simply have better things to do with their time[58]. Focus groups and individual interviews sacrifice anonymity (and time to a certain degree), in order to ensure users respond to the questions being posed. To clarify, a focus group might use the same questions as an interview, except a group of people are questioned all at once and discussion between individual participants is encouraged[59]. The questions that one uses in this sort of situation are unlikely to be all that different from those used in surveys (such as those explained above), however participants are more likely to answer if there is someone there to guide them. Anyone wishing to evaluate this system or its components using focus groups or interviews can use the questions created for the sample survey evaluation as a starting point.

6.2.3 Evaluation through Experimental Extensions

Hornbæk [60] suggests that one might use usability tests as a method for idea generation - something that Section 5.4.3 (changes made based on hallway testing) demonstrates well. This section elaborates on how some of the concepts and ideas gained through hallway usability tests during the course of this work, can be further used as a means of extensibility evaluation.

This type of test relies on the tested component being in some way incomplete or extensible - the basic idea is that one adds something new to it. For example, one might try to incorporate new concepts into the metaphors, or a new piece of functionality into the API or game. Through careful documentation of the process undertaken, one is able to (subjectively) state that the system in question does or does not adhere to quality criterion X because of design or implementation reason Y.

Throughout this dissertation mention has been made of possible extensions to the system and its constituent components; here is a summary of the potential enhancements one might try to implement as a means of quality evaluation:

- Extend the metaphors themselves to include: specific flow-control structures, enumerators, delegates, generics, reflection, bitwise operations, garbage collection, inheritance, interfaces and aggregation, read only data, threads, or structs. Successful

inclusion of any of these non-novice concepts would demonstrate a high level of extensibility, and fidelity.

- Alter an implementation, or the API, to improve the user's experience - the process of doing this will indicate how modular, extensible, alterable, and understandable the two systems are (or are not).
- Try to incorporate an alternative or new metaphor into an existing system. This is another gauge of modularity and extensibility.

The above examples can be used by future testers. Several extensions were implemented during development, the process and findings for each one is discussed in the next chapter. The extensions included:

- Adding an **undo** button to allow easier user experimentation.
- Inclusion of the BoolEater metaphor, to accommodate issues users had with conditionals.
- Inclusion of the notepads as an alternative to the calculator.

6.3 Metaphor Evaluation

There are several perspectives one can take when evaluating metaphors designed for teaching: one can take the perspective of the student ("does this help to clarify things?"), the teacher ("are these accurate and usable enough for one to teach with"?), or the academic ("what makes these metaphors good, especially when compared to others?"). Each of these perspectives is valid, and each one requires separate consideration. Most of the general techniques explained above can be applied to metaphors; of those techniques, a detailed sample metaphor evaluation survey was created and can be found in Appendix E. The aforementioned survey was designed to be suitable for both novices and more experienced programmers (provided the novices have gotten up to the concepts in question), and it includes both qualitative and quantitative questions.

6.4 API Assessment

API evaluation is particularly troublesome, not least of all because it is an "infrequently researched topic" [61], but also because one must first decide what one is evaluating and what matters most (which may vary based on the API being evaluated). For example, one might need to rank and evaluate: extensibility, modularity, understandability, readability, performance, re-usability, and general usefulness.

As ranking these criteria is completely subjective no attempt was made to do so, however several means by which one *might* evaluate the criteria are proposed later. Most of the tests explained here rely on experimentation and use of the API, usually through an additional from-scratch game implementation. A 'from-scratch' test involves taking the API as it stands, and creating a secondary game implementation around it (preferably using a new framework). During the process of re-implementation the tester is encouraged to avoid looking at any previous code (aside from that of the API) - this is done so as to avoid contaminating the development process with previously made code. The tester should document any problems they had during the process, any alterations that needed to be made to the API, anything about the API that made their task easier, and anything else of note. This information can then be used to 'grade' various aspects of the API.

This particular evaluation technique was used, and not only revealed information about the API, but occasionally it also highlighted other interesting facts (for example, it was found that the code interface required no modification for this exercise). Section 8.1 goes into more detail regarding the implementation of this technique. In short, the exercise proved useful and allowed the author to gauge many different things at once.

Another type of API evaluation technique is one that details the process involved in implementing an entirely new feature into the API - one might call this an extensibility test. The way this works is not all that different from an extra implementation (it is, however, much quicker): one must first establish and define a new feature, and then detail the process undertaken in order to implement the feature in both the API and whichever game implementation it is included in. The steps involved are then analysed and described in order to elicit details about the quality of the API and game. As several such extensions were made to the API, an extension evaluation section was included (Section 8.2.1).

Less experimental evaluation criteria can be used as well: a comprehensive set of unit tests for the assorted components of the API would serve to evaluate the robustness of

the system (the more tests that pass without issue the higher the overall quality), while a series of timed tests could be used to evaluate performance. As performance never presented itself as an issue, no performance tests were done. Unit tests were used during early developmental stages they were used, however the quick prototyping and iterative revision processes used here made them less useful later on.

Almost all of the more technical components (including API classes, interfaces, specific implementations, and even the level editor) can have their quality demonstrated through a simple process of show and tell: one simply shows how the original component was designed, along with how that design fared throughout development, and then if necessary explain why these two things come together as a demonstration of quality. There are several examples of this technique in the next chapter, one of which is a demonstration of the value of continuation style programming.

Ellis et al. [61] state that one of the biggest barriers to the usefulness of APIs is due to their scale (potentially hundreds, or even thousands of classes per API makes them very hard to learn). This can therefore be used as a criterion for measuring API quality: the less interfaces and classes a user has to deal with, the higher the quality of the API (generally). As explained in Section 5.2.2 the ActionWorld virtual machine API only really requires the user to interact with two or three methods of a single centralised class - this is a promising indicator of the API's quality.

6.5 Quantitative Game Tests

6.5.1 Checklists

Checklists have long been used as tools for evaluating the quality of all kinds of systems, including educational programs. They have, however, fallen into disfavour for several reasons, which Squires and Preece [62] summarise nicely:

- Check-lists are unsuitable for evaluating unique or original systems [63].
- They evaluate functionality more than socio-educational concept adherence [64].
- When comparing systems of a similar nature, check-lists tend to highlight similarities, while obscuring differences [65].

- Checklists cannot take into consideration alternative 'off-computer' uses for the software in question [65]. For example, components of this system can be used in textbooks or classrooms (other components can be used in almost unrelated systems, such as the reflective calculator). While other systems might not be usable that way.
- Differentiating the importance of individual points on a checklist can be challenging [66].
- When two systems use different educational strategies, they can become incomparable [66].

Nevertheless they still have their merits, and are easy to apply, while being totally objective: a checkbox is either checked or unchecked, and potentially subjective feelings shouldn't change the result.

Kelleher and Pausch [4] created an exhaustive feature comparison table for 80 different educational programming games. This sort of comparative table can be used on any system aimed at teaching programming: one would simply mark the features of the system in question on the table and compare how that system does when compared to the others. Because of the simplicity and relative objectivity of this particular test, it was applied to ActionWorld. Section 7.1 goes into more details regarding the process and results.

If one wanted to perform further checklist-style evaluations, one could use the list proposed by Victor [18], who concludes his analysis of what makes up a good teaching environment by presenting a 'checklist' of criteria that both the environment and the language need to adhere to.

6.5.2 Rubrics

Squires and Preece [62] briefly mention the use of rubrics and how they are better suited to evaluation than checklists, as they offer a more variable scale. They specifically mention the California Instructional Technology Clearing House's criteria [67] as being well suited to evaluating educational systems, however this resource could not be acquired despite an extensive search. Squires and Preece say a great deal regarding rubrics and ultimately conclude that while they are better, rubrics often boil down to being very granular checklists - they also say how the application of rubrics is occasionally a long, tedious process, and they conclude that heuristics are a suitable substitute.

Allen and Tanner [68] focus more evaluating students through the use of rubrics than evaluating software - but they do state that extending checklists into rubrics is often the easiest way to create them (a statement that lends weight to Squires and Preece's granular-checklist perspective).

6.5.3 Heuristics

Heuristics, also referred to unofficially as rules-of-thumb, can be applied to systems such as ours in a less formal manner than checklists and rubrics. According to Squires and Preece [62] this informality is not a disadvantage, but instead allows for flexibility during evaluation. A well established set of software evaluation heuristics are the ten usability heuristics of Nielsen [69], which could be applied to a system such as ours. However, Nielsen's heuristics were designed for general software, rather than the very specific set of requirements that educational programs have. With this in mind Squires and Preece propose their own set of eight heuristics specific to the evaluation of educational software. Squires and Preece's heuristics are applied in Section 7.2.

Reeves et al. [70] also proposed an extended version of Nielson heuristics, with a focus on evaluating educational software. Their set is composed of fifteen different heuristics (as opposed to Squires and Preece's eight), and was deemed too extensive to include in this dissertation.

6.6 Qualitative Game Tests

A qualitative game test in its simplest form is just user feedback about what they thought of the game. Such an evaluation can be carried out using any of the general qualitative measures explained in Section 6.2. Hallway testing is a valid, if informal, way one might evaluate a program - the fewer problems users find, the better the quality. However, as hallway tests are aimed more at improving quality than evaluating it, they have been kept separate. Section 5.4 goes into detail regarding hallway testing of this system and what it yielded.

6.7 One-on-One System Comparisons

This type of evaluation takes a system similar to the one in question and performs a detailed evaluation of one of the systems based on an already complete evaluation of the older system (optionally both systems can be evaluated using an entirely new technique). One-on-one system comparisons are done by taking the most similar components of two systems and comparing them to one another side by side. The difficulty of performing such a test differs based on the depth and detail that the tester chooses to go into during comparison - this variable difficulty range means that performing such a comparison can be done without going out of scope or exceeding time limitations. Section 7.3.1 compares this system to that of Gilligan [5].

6.8 Summary

This chapter explained an unusual way one might test the quality of a complex system, that of constituent evaluation: the idea behind this technique is that one can use specialist tests on each of a system's individual components in order to evaluate the system as a whole. The alternative would be to try to evaluate everything at once using a single complex test.

Several sample tests for this framework were also included in this chapter (at least one test per component type), for example:

- Surveys for metaphor validation.
- From-scratch game implementations for API validation.
- Checklists for game-feature quantification.
- One-on-one comparisons with similar systems, as a means of whole-system validation.

Some of the tests are suitable for complete system evaluations (such as one-on-one comparisons), however the results of such tests are likely to be more valuable when applied to specific system aspects. The next chapter describes some of the less technical results obtained through the application of this chapter's proposed test framework.

Chapter 7

Non-Technical Test Results

The previous chapter laid the theoretical foundation for the constituent evaluation framework, and gave several sample tests for the various evaluation techniques. This chapter elaborates on the results obtained through the application of some of the aforementioned techniques.

7.1 Checklist Evaluation - Kelleher and Pausch's Attribute Survey

As described in the previous chapter, checklists can be useful, especially when one considers how objective they can be. With this in mind a checklist-style comparison of multiple systems was performed, based on work by Kelleher and Pausch [4], in order to evaluate the system as a whole, relative to other similar systems:

Kelleher and Pausch compared an assortment of programming education systems, as briefly described in Sections 2.3 and 6.5.1 above, which included a comparison of each system's attributes and features. This section compares and contrasts the features of the system to those systems analysed by Kelleher and Pausch. Figure D.1 of Appendix D shows where in Kelleher and Pausch's categorisation model the proposed system belongs, and Figure D.4 of Appendix D shows the more detailed feature analysis of this system next to Kelleher and Pausch's original data (these figures were relegated to appendixes due to their size... as an illustration of their magnitude, the attribute frequency table has over 4000 cells and covers three full pages).

There are several logical divisions that one can use when comparing against the systems in Kelleher and Pausch's survey:

- One can compare against all the systems (a general comparison).
- One can compare against various, more specialist, categories (such as the purpose of the systems, or the way they try to represent things).
- Finally one can perform a one to one comparison.

7.1.1 General Feature Comparison

This sub-section performs a comparison of all the systems in the survey, and tries to identify trends in feature combinations. As the systems analysed by Kelleher and Pausch [4] are somewhat dated (with the most recently published system being from 2002), one needs to consider historic changes in emphasis (such as a shift from procedural to object oriented programming), as well as the fact that technical limitations of the time may have prevented some systems from being implemented in a more modern manner (for example, making a GUI system is much easier nowadays).

The results of comparing the proposed system to all the other systems analysed by Kelleher and Pausch shows that about 56% of systems promote a procedural style of programming, while just 18% allow for object oriented styles - of all the analysed programs by Kelleher and Pausch, only one (out of 80) allows for both procedural and object oriented programming, therefore one can safely say that ActionWorld is contributing to an area of educational games that did not have a lot of support at the time of Kelleher and Pausch's study (ones which support both procedural and object oriented programming assistance).

Kelleher and Pausch's analysis of programming structure frequency shows that the two most commonly included ones are conditionals (73%) and methods (63%). Interestingly, of the systems that support methods, 30% did not allow for parameters. Iteration of most kinds is poorly supported with only 37% of systems allowing even a while loop (for and count loops are even less frequent). ActionWorld allows for all kinds of loops, simply because it was easy to include them, given the dynamic nature of the virtual machine's level interpreter.

A fundamental concept that novices need to come to terms with is that of variables, and yet only 46% of the given systems had some kind of variable mechanism built in. As far

as representation of code goes, 60% of programs used text, with pictures being the runner up at 27%. This work's game belongs with the majority of other systems in this regard, however the extensibility and versatility of the level representation system means that one could (in theory) extend this system to allow multiple representations (no system offers more than two simultaneous code representations).

More than half of the systems analysed by Kelleher and Pausch offered no help for understanding code at all, while ActionWorld offers two different kinds (debugging and physical metaphor) - the focus of such systems was often more on the way code was created, rather than its meaning. This is understandable for the none-education systems, as they often obfuscate code entirely[1].

There is not much to be said about Preventing Syntax Errors, Designing Accessible Languages, and Help Systems (Support Communication) aside from the fact that statistics here averaged at about 10% (meaning very little support from everyone). Kelleher and Pausch's survey shows that a large number of systems (41%) simplify the language in order to improve accessibility to novices. By contrast, ActionWorld can hide detail, but is able to work with unsimplified languages. In the case of this system, the level designer can choose the constructs to expose and exercise in each level.

7.1.2 Specialist Category Comparison

Kelleher and Pausch [4] divide all the systems into two main groups based on their purpose: do the systems aim to teach users to program for its own sake, or is their goal to improve productivity by simplifying the task of programming? This system fits into the first (more academic) category, however the major differences between systems in the two categories are difficult to identify.

The largest differences between the two basic categories can be found in event-based programming and choice of task mechanics: the systems designed for productivity improvements supported event-driven paradigms more than twice as often as those designed for teaching programming fundamentals. Event driven programming is not something that the proposed system supports, however the potential for inclusion of event driven code has been discussed and is described as a potential advanced feature in Section 4.4.4.

[1] The none-education systems catalogued by Kelleher and Pausch are usually intended as tools for simplifying certain tasks that would normally require programming knowledge to address (such as making games).

The other major difference between categories, is that the games which aimed to teach programming for its own sake, prioritized education as the reason to choose a task 98% of the time, while the tool categories' priority was usefulness (67% of the time). This is not particularly surprising, and ActionWorld conforms to these numbers, reflecting this work's primary interest in teaching fundamental concepts.

7.1.3 Brief Comparison with Gilligan's System.

Using Kelleher and Pausch's [4] categorisation system, the game most similar to ActionWorld is Gilligan's [5] 'Prototype 2' (this is the second, more refined, implementation of Gilligan's system). If one compares the two systems (just based on Kelleher and Pausch's feature table), ActionWorld generally has more functionality. One of the key feature differences is that Gilligan's system allows for reverse-mode code generation while ours does not. Another example would be the language support of each system: this system can work with most non-functional languages, while Gilligan's can only work with a simplified Pascal dialect.

The emphasis of ActionWorld is to make students understand operational aspects of the individual components of the code that they are presented with. By the end of a level they should understand what the code does (at various scales) simply because they have run through it themselves. One of the overarching aims was to present a game that took an alternative from the traditional perspective of "here is a problem, please solve it" and instead takes on the goal of "here is some code, explain it" - therefore, in this regard, ActionWorld is not actually lacking.

Kelleher and Pausch incorrectly classify Gilligan's system as having no support for variables. They were omitted from one of the prototypes due to time constraints, but were catered for in the system as a whole.

7.1.4 Statistical Comparison

As the number of systems in question are so varied, both in style and feature sets, a statistical analysis of feature frequency only has limited value. However, a rudimentary one was performed nevertheless and demonstrated that on average the systems in question had only eleven of Kelleher and Pausch's [4] features per system. Figure 7.1 shows a graph plotting system feature count frequency. What can be seen from this data is that without

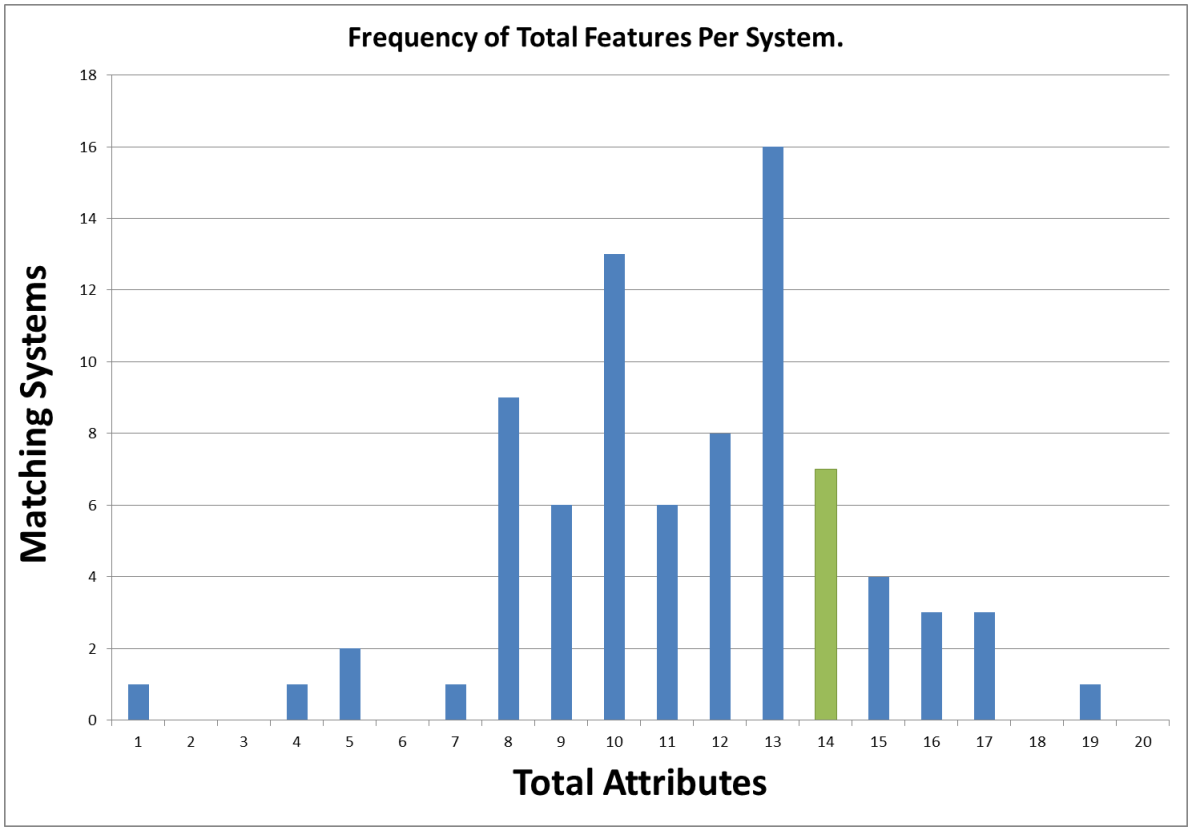


Figure 7.1: Frequency of total features per system. The proposed system qualifies for 14 of Kelleher and Pausch's [4] features, while Gilligan's [5] Prototype 2 qualifies for 10.

taking specifics into consideration this system has, on average, three more features than other educational games in this survey.

When comparing against the systems that share a fundamental category, one will find only two others (1996 ToonTalk and 1998 Prototype 2). The proposed system has two and three more features than these two systems respectively. When continuing to compare against 1996 ToonTalk and 1998 Prototype 2 the two biggest advantages are that this system can support object oriented programming, and almost all of the programming constructs of modern languages, whereas the other two cannot.

7.2 Evaluation via Specialist Heuristics

Squires and Preece [62] describe a number of different ways one might evaluate software, and explain the need to examine educational programs from a more learner-oriented perspective, rather than an expert or technical perspective. This section applies Squires and

Preece's eight heuristics - mentioned in Section 6.5.3 - to the system as a whole, in order to try and evaluate it from a more user-oriented perspective, and as an alternative to the rigid checklist analysis described above. One should note that heuristic evaluation is typically performed by an objective third party with extensive experience in such evaluation techniques. Unfortunately an individual such as this could not be found in time, and thus the author performed the analyses himself.

7.2.1 Heuristic 1 - Matching Mental Models

The first requirement put forth by Squires and Preece [62] says that there must be "a match between designer and learner models". In short, models portrayed in the software need to strike a balance between accuracy and understandability - they need to be simple enough to understand, but not so simple as to result in a superficial mental model. This has aligned closely to the philosophy and goals of this work from the outset: to give the learner robust mental models that were consistent with those of an expert, while not overwhelming the learner with details.

7.2.2 Heuristic 2 - Navigational Fidelity

Squires and Preece [62] go on say that there is a need for "navigational fidelity": A system with good usability will often sacrifice the accuracy and complexity of its underlying system in favour of a simpler interface. It is hard to say whether this has been done: the system (as it was) did not seem highly usable, and yet a certain degree of underlying accuracy was sacrificed in favour of promoting understanding. One could almost interpret this point as saying that just because a system hides some complexity, it does not make it a bad system - and vice versa. A system that makes gratuitous use of multimedia for navigation can distract users from the actual learning outcomes of the system.

As the proposed system changed more and more navigational complexity was hidden in favour of usability, and this did occasionally result in better user interaction. An example of this is the calculator which was removed in favour of simple notepads - this removed a degree of accuracy (such as variable reading and substitution), however it compressed a three or more step interaction into just a single step - it also had the added bonus of forcing users to decide on the resulting values type.

A second good example of where this guideline came in, is in the use of the method calling mechanism. It originally required that users type all the details of the method that they

want to call themselves (often resulting in confusion regarding what was required). This complex mechanism was replaced with a simple selection menu, which provided more scaffolding for the learner, and drastically improved user navigation of method calls.

7.2.3 Heuristic 3 - Appropriate Learner Control

Once it was decided that the proposed system was at a standard where it could be analysed in detail, it was felt that there was only partial conformity with the third heuristic proposed by Squires and Preece [62]: good educational programs require an "appropriate level of learner control", which means that a user needs to feel in control of a system in order to feel in control of their learning. Existing features that seemed to allow some degree of learner control included a sandbox level where learners could experiment, as well as being allowed to choose whichever level they wanted to practice with.

These features alone did not seem to constitute a high enough level of learner control, and so two extensions were proposed that could improve on this: the inclusion of an undo button, and additionally (or alternatively) an enhancement of the Sandbox mode so that sprite interactions would create code that learners could observe. It was decided that the inclusion of an undo button would serve to improve the system in this area more than the code generator - the implementation of the undo button is explained in Section 8.2.1.2, and Section 9.6.1 gives a brief analysis of what implementing a reverse-mode code generator in the system would imply.

Implementing the undo mechanism was not entirely trivial. Reversing code execution is difficult, and keeping state snapshots at each step is costly. To move backwards by a single step, the whole program was re-executed from initial conditions up until the previous step - and then the state of the Model was re-displayed.

The resulting improvement in control and playability was quite dramatic: all of a sudden experimentation and exploration became easier, simply because an action could be re-done until it was understood. This is a good example of the iterative development methodology: despite nearing the end of development, when a potential weakness was identified, development went all the way back to the Model/API level in order to address it.

7.2.4 Heuristic 4 - Avoiding Tangential Complexity

The fourth proposed heuristic is that of “prevention of peripheral cognitive errors". In short, systems need to minimise the occurrence of less important errors so that users can focus on more important mistakes that they might be making. For example, a cluttered or unclear interface might result in users accidentally clicking the wrong button by mistake, despite knowing what the correct action was. Squires and Preece [62] elaborate on this point by saying that, when possible, there should be a "novice version of an application".

The proposed system addresses this in multiple ways: the option is given for levels to hide unnecessary sprites (minimising clutter, or information overload), and an optional hints/tutorial mode where the next appropriate sprite interaction is highlighted is also available. These two features can be combined as the designer chooses and result in 4 basic levels of complexity (but potentially more if one considers the various number of shown sprites to be their own complexity levels): both, just one, or neither. What this means is that the system can vary in complexity according to user needs, thus minimising “peripheral cognitive errors".

The various hint features can be customised by the level designer as well as by the end user, thus adding to their feeling of control, which pertains to the previous heuristic about users needing to feel in control of their learning.

7.2.5 Heuristic 5 - Meaningful Metaphors

The next point (or at least some of it) is in fact one that every aspect of the proposed system aimed to address from the start: the need for “understandable and meaningful symbolic representation”. The first requirement of this point is consistency of interaction, and intuitive icons and behaviour - consistency was addressed by ensuring that metaphors change as little as possible between related concepts. Of course, the author acknowledges that others may come up with more intuitive representations. Certainly for some concepts (such as the method mechanism), it seems as if the proposed system meets the criteria for this concern. The issue of intuitive interfaces is partially addressed during the usability tests (which are explained in Section 5.4), which revealed gradually improving usability after each interface improvement.

7.2.6 Heuristic 6 - Personal Approaches

Squires and Preece's [62] heuristics have a point that is very broad, and which could be interpreted in multiple ways: the need to "support personally significant approaches to learning". This heuristic can be interpreted as being about offering support for multiple learning styles (this refers to styles such exploratory vs. linear learning, or visual vs. abstract learning).

Fortunately this particular point has one requirement which is not subject to interpretation: one must be able to identify the learning styles supported by the system, and how they relate to one another. For this system this is fairly easy: support for visual learners is offered throughout, and then either exploratory or linear learning depending on the game mode being used (i.e. Sandbox or level mode respectively). The focus on visual learners is not a drawback for the system as a whole, as being highly specialised should help learners who do not have the abstract learning style which is arguably most conducive to learning to program.

Some have suggested that abstraction abilities (especially in South Africa [71]) appear to be on the decline, mainly evidenced by a decreasing quality of maths and sciences in schools, therefore there are likely to be an increasing number of students who lack abstraction skills and who would therefore benefit from a visual style of learning (such as the one this system aims to provide). Arguably one can reverse the cause and effect regarding maths and abstraction skills to say that lack of abstraction skills causes trouble with mathematics - if this is the case, the cause of the problem might be more deeply seated [72]. Regardless of the initial cause, this system still aims to help users who have abstraction problems.

There is a great deal more that can be said about supporting personally significant approaches to learning, including:

- How the proposed system (and its constituent components) relate to Vygotsky's mediation. In short, this is the idea that one's higher mental functions and perception of the world are likely to vary greatly between individuals, based on their upbringing (for example, the culture they grew up in, or the concepts presented to the person as important during childhood development).
- To what degree the metaphors can be considered personally significant to students (this includes how they relate to the motivation of the user, and how relatable they are).

- Whether or not the attempt to limit concepts to a single representation could be perceived as limiting rather than empowering.
- What degree of automaticity can be achieved through the more linear teaching style.

A thorough discussion of these points would be enough to fill an entire chapter, thus such a discussion has been deemed as being out of scope, however that is not to say that someone couldn't perform such an analysis in the future.

7.2.7 Heuristic 7 - Fast Feedback

Squires and Preece's [62] second last point is deceptively simple. All it states is that a good system will allow users to realise when they have made a mistake, figure out what they did wrong, why it was wrong, and finally recover from the mistake by figuring out the correct course of action. This system originally had two mechanisms in place to assist with this cycle: a textual hint system to give them guidance about what needed to be done next, and a bold feedback sprite that clearly notified users about whether their action was right or wrong.

One might argue that the sprite highlighting system and undo button offer a certain degree of support for this kind of thing, however the undo button offers far more in the field of user control than it does for error recognition and recovery, while the highlighting system is there to lower the learning curve of the tool. With these features in mind it appears as if the proposed system offers a moderate amount of support for this point, but would benefit from more powerful mechanisms: the first potential enhancement would be to show the user an action specific message that would tell them what their incorrect operation would have done and what they actually wanted to do next. A second, more difficult to implement, enhancement would be to let the user take a single illegal step, lock everything except the undo button, and ask the user to identify what about the current state is wrong. Both of these enhancements have the potential to increase the score for this particular heuristic, however neither one would be trivial to implement.

One possible down-side of providing instant right/wrong feedback, is that students are not required to think ahead, which in turn can lead to them simply proceeding through a level via trial and error. Therefore, when implementing a feedback system, it needs to be carefully designed so as to avoid this sort of issue.

7.2.8 Heuristic 8 - Custom Curricula

The final requirement proposed by Squires and Preece [62] is that of a "match with the curriculum". This point is somewhat vague regarding its requirements, but its emphasis seems to be on teacher customisation and adherence to a particular curriculum. This system allows for a great deal of customisation, especially with the inclusion of a level editor, as teachers can customise every aspect of a level, including: the amount of code highlighting used, the sprites shown, the hints presented, the programmatic structures used in a level, and even the programming language used (provided it's within certain reasonable boundaries, such as being a non-functional language).

7.2.9 Summary of Heuristic Adherence

To summarise the analysis based off of Squires and Preece's [62] heuristics: they propose eight requirements, of these the proposed system adheres well to six of them, while one of them receives a reasonable degree of compliance. As the remaining point ("understandable and meaningful symbolic representation") is one of the major goals of the proposed system, it is subject to debate and further analysis.

On the basis of the above criteria the author believes that the system scores well on Squires and Preece's heuristics test.

7.3 A One-on-One Comparison of Systems

As explained in Section 6.7, a valid means of gauging quality is to compare against an already established system of a similar nature, using whatever means of evaluation the system used on itself, which is what is done in this section. This sort of comparison serves to both compare quality and distinguish between systems (on the surface two systems might seem almost identical, while a closer inspection might show several important differences).

7.3.1 Comparing against Gilligan's System

As briefly mentioned back in Section 2.6, Gilligan [5] performed a quick evaluation of his system by comparing against common fallacies inherent in teaching systems, as originally

presented by Eisenstadt et al. [19]. The proposed system was analysed based on the same criteria, in order to create a more quantitative comparison between the two seemingly similar systems. Eisenstadt et al. uses the term 'fallacies', however the author believes that they are more akin to potential pitfalls, or criteria that need to be met (or avoided, when appropriate).

The first issue presented by Eisenstadt et al. is that of educational systems attempting to create a complete bug catalogue, and corresponding suggestion database, for every 'solution' that a user might attempt to create for a given problem. Gilligan says that this is not an issue for his system as users do not create code themselves (and thus cannot create syntactic errors), this same reasoning applies to the proposed system, and thus both score equally for point one.

The next point is that of poor interfaces being a barrier to learning (a point elaborated on by many other authors). Gilligan admits to not trying to avoid this potential pitfall "because we chose not to focus on the user interface". This work also followed the adage of functionality over aesthetics, however a reasonable interface was still successfully created (the primary drawback with this systems interface is the sprite artwork, because no one involved in the project was a graphics designer). From this one can say that the proposed system probably has a better interface than Gilligan (certainly the gradual introduction of the assorted components tries to minimize the risk of overwhelming users). However, neither system deserves full marks for this point.

Gilligan explains that they did not specifically address the next two issues, but that their system could probably cope with them:

"Systems not scaling up from small toy examples" is the third cautionary note presented by Eisenstadt et al.. This system was never designed to handle programs of 1000 or more lines of code (though everyone's definition of toy examples will differ), therefore, like Gilligan, this work does not explicitly address the third point. One could attempt to address the issue about scaling well as an exercise to further prove the systems quality: one would have to create and test a *large* level that demonstrates various concepts all at once. Regarding this third point, it is unclear whether the cautionary notes presented by Eisenstadt et al. were ever intended for visualisation systems (which have very specific requirements). It is the author's belief that code visualisers designed for novices only need to handle very small examples, and that this point is thus less applicable.

The fourth and final potential pitfall presented by Eisenstadt et al., states that many teaching programs seem to say that they know the best way to learn. This is one that

this work does not claim to overcome, however this work does make a point of saying that the system is a proposed alternative to existing models, but is not necessarily an improvement or replacement. No such claims are made about whether any individual system is the best, Gilligan also admits to not addressing the fourth problem explicitly but suspects that they do not fall prey to it. It might be argued that simply mentioning a point such as this means one is probably not falling prey to it, simply because an awareness of the issue has been demonstrated. Anyone that wanted to make a claim such as this would need an overwhelming amount of evidence obtained over several years using controlled student test results.

The comparison of the proposed systems using these four pitfalls can summarised as follows: the proposed system purposely avoids at least two of them, the third is potentially inapplicable, and the fourth point is addressed by admitting that the system is simply a proposed alternative to established systems. Gilligan states that they only attempt to avoid one of the four, and go so far as to admit to failing at another one, while the remaining two are undecided or unaddressed. It seems fair to say that Gilligan did in fact avoid the pitfall of stating that they had found "the best way to teach and learn", simply because Gilligan consistently refers to his system as a prototype and never as a replacement. This results in a final score of 3/4 vs. 2/4 (not including negative marking for fallacies that are present). The proposed system outperformed Gilligan's in at least one out of four points, and matched them on another one. However, all that can be said about the system based on this is that *for this set of tests* it probably did better than Gilligan. This is not proof of overall quality, but more of a comparison of relative quality guideline adherence.

One might argue that such a comparison is without merit, simply because the work by Eisenstadt et al. is not wholly applicable to either system. As only one point is not applicable (and its inapplicability is arguable) such a comparison is in fact still valid, even if all that it shows is that one system attempts to address more issues than the other.

7.4 Test Result Limitations

Chapter 6 elaborated on a multitude of different test techniques, several types of which could not be applied due to time and scope constraints. Key factors that the partial application of the proposed test framework did not test include:

- Whether the system would have any impact on the learning experience of novices.
- Whether application is likely to create stronger mental models.
- How users respond to the system (or any of its components) - this could include user experiences ranging from UI concerns to confusion regarding certain metaphors.

Fortunately these limitations do not detract from the system, as they are facets of a much larger whole that could not be comprehensively analysed. In fact, such a comprehensive analysis of one's own system may lead to concerns regarding objectivity. The user interactions documented during hallway testing (Section 5.4), while not strictly speaking a part of the test framework, can be referred to in order to at least comment on system usability: over time the interface and usability of the system was improved, and could probably be further improved with only minimal alterations to the metaphors, API, or implementations (as demonstrated by the feature extension examples in Section 8.2).

7.5 Summary

In this chapter several evaluation techniques were applied to the proposed system, and their results were elaborated upon. In general the results were fairly positive. Despite some limitations to the scope of the tests, this chapter demonstrates that the proposed constituent-evaluation framework is a valid means of determining quality (if it were in fact invalid, none of the applied tests would have yielded usable results).

This chapter did not go into the more technical evaluation mechanisms, which are instead elaborated on in the next chapter.

Chapter 8

Technical Evaluation and Enhancement

The previous chapter covered subjective validation techniques, while this chapter contains more technical measures of quality, with a major focus on the extra WPF implementation. It also includes the details involved in adding certain extra features to the API and game. Readers with less technical interest can skip this chapter without loss of continuity.

8.1 A WPF Implementation and what it Demonstrates

As explained in Chapter 6, one can evaluate the quality of the API and metaphors via a detailed analysis of the process one must undertake in order to create a secondary implementation of the game: the original game was written using the XNA framework, and the 'evaluation implemention' was written using WPF.

This section takes a more technical (code-centric) approach to the evaluation of both the API, and the implementation details of the prototype games. Originally it was planned that there would be only one implementation of the proposed system, created in XNA. Up to a point a single implementation is fine, however when one wants to try and evaluate a system's constituent components, having a single implementation limits comparability. Therefore the first step taken toward a more technical analysis of the system was to re-implement it using another framework, more specifically as a WPF program.

The WPF version does a good job of illustrating several critical design details, however it also had a secondary effect of highlighting additional areas of interest that already existed in the system (either in the XNA implementation or the underlying API).

The originally goal of creating a WPF version of the system was to probe the resilience of the underlying system-model API. By the end of the process the extra implementation had demonstrated more that just MVC compliance: concepts covered ranged from the merits of WPF over XNA, to minimising the effects of cross cutting concerns, and even illustrating how a complex abstract system of interaction can actually make development easier (this is in regards to passing delegates as event handlers, which themselves contain delayed Model-method calls - Section 8.1.3 goes into more detail). This summary serves to explain key examples established through the creation of the WPF version, as well as some closely related technical features of the API that were simply brought into focus during the WPF development.

As described in the methodology chapter, an iterative approach was used to create the WPF version of the system. The development process was slightly altered from the original XNA process due to fewer design uncertainties (which was due to the fact that many of the uncertainties had already been addressed during the XNA implementation).
+

8.1.1 MVC meets WPF

Implementing an MVC compliant GUI application can be slightly tricky at design time: the API forms a solid Model section, but it also has a class (WorldTracker) that can masquerade or take on responsibilities of the MVC Control class if one is not careful when using it. The WorldTracker class is primarily meant for telling all the sub-classes of the API how they need to interact with each other, as well as allowing for easy communication with a *proper* control class - in essence, WorldTracker is the main control class of the Model classes, and acts as something of a communication interface when used in an MVC context.

This observation regarding separation of concerns forces one to think very carefully about exactly where all the responsibilities belong, especially so when creating the WPF version, as its front-end (the WPF form) can already subsume much of the functionality of the MVC View class, or alternatively take on a mix of Control and View responsibilities (not unlike the main Game class in the XNA version). In order to separate responsibilities and to keep the WPF layer as "thin" as possible, two new classes, named View and Control, were created. They take on as much of their respective MVC responsibilities as possible while using the WPF framework: The WorldTracker-Model class and View class only ever communicate with Control, which itself acts as a central communication hub

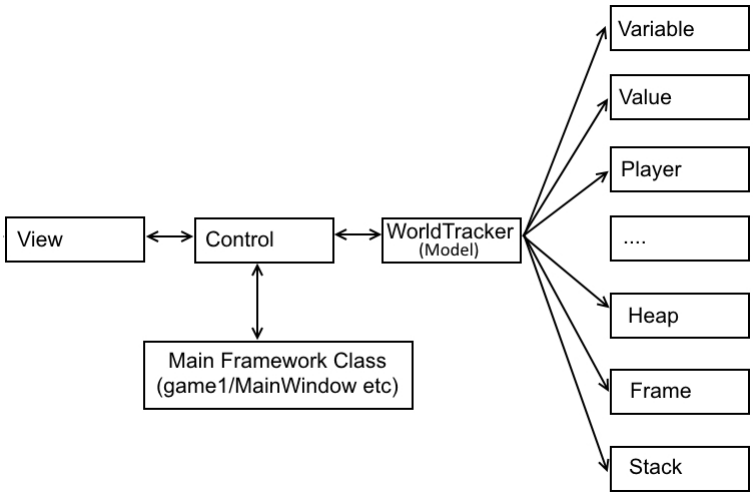


Figure 8.1: The assorted classes and their lines of communication in the WPF implementation.

- Control has pointers to the WPF Form, the View, the WorldTracker (Model), and the Level-Code interface classes. The WPF form also has a pointer to Control as everything begins in the form (one could have changed the startup options so as to begin in Control, and then create a child WPF instance, however this seemed unnecessary and would have complicated things, such as testing and whether the form actually belongs to the View class). Figure 8.1 shows a rough representation of the relationships between the MVC compliant classes.

The relationship between the more abstract View responsibilities and the more concrete WPF MainForm is interesting, as ordinarily one would create the assorted GUI components inside the form before running the code, but the proposed system does things differently: Control knows the startup details of the various uncreated sprites, it then requests that the View class create and return the required sprites, which are then given to the form by Control. This unusual chain of command means that any single component can be replaced, without affecting any of the other components (provided the replacement adheres to certain communication rules). While it wasn't necessary in the implementations, one could use interfaces and aggregation to make such interchangeability even safer and easier.

8.1.2 MVC Compliance

After going through the various design considerations and distribution of responsibilities issue - and then implementing it - the most important thing demonstrated was just how

platform independent the underlying *Model* API was: during the whole exercise of creating a WPF version, not a single alteration to the API had to be made in order to make some feature work (provided those features where already present in the XNA version). This is a key ideal of MVC: that the three main classes should be replaceable without any alterations to the other two (or at most, very minimal alterations).

Animations and MVC do not always seem to fit well together. In MVC, an event typically updates the model, leading to a new view. But when animations are involved, an event might trigger an animation (something that essentially belongs in the view) which should only later lead to a change in the model, and to any other views that depend on the model. (For example, care is required if the Memory Manager Robot has a lengthy animation sequence to place a value in the Heap.) A good demonstration of the modularity in this system, is the updating of the visual components when (and only when) the operation attempted by a user is *completed*: in a less powerful or less 'convenient' implementation of the API one might need to poll the Model on a timer (regardless of its state), or make the Model raise a special 'action complete' flag or event so that the controller would know exactly what visual updates need to be made. In this implementation the Model is passed an Action (a void delegate) parameter from the sprite being interacted with when the user performs an operation, which is called upon completion of the Model's operation logic. This same behaviour can be applied in reverse, where a lengthy animation notifies the Model when it is completed. Several such interactions can be strung together in order to facilitate complex Model-View behaviours.

This design feature means that the Model class does not need to know anything about the Control or View classes, instead all it needs to know is what Action to undertake upon completing an operation (the Action is provided by the Control class). This also means that the only object which needs to know about the action-to-perform is the sprite in question, thus eliminating cross-cutting concerns while keeping things modular. As an interesting point of note, this feature allows every interactive object to have a different finishing-action to call when an operation is performed, meaning the potential for sprite specific visual updates and other similar fine grained control. This idea of asking a component to do something, and also passing it a parameter that says "here is what to do when you are done" is called *Continuation-Passing* [73] and has become a key part of web-based programming, where developers often want to say "Fetch this resource, and here is the code you need to execute when you have got it".

Some might argue that this MVC-"Continuation-Passing" style has the potential to preclude certain ways of working, for example triggering a Model update halfway through

an animation might be difficult. However this is in fact not the case: if one wanted to, for example, add an animation to the players hand when he attempts to write a value to a variable, it might take three steps: move the hand and paper towards the variable box (first animation), update the model, and finally start a concluding animation that tears up the old value. Since, in this example, the call to update the model happens after the first animation begins, subsequent calls and checks become slightly more complicated. However one can work around this issue - continuing with the above example, one could set things up like this: the user interaction triggers an animation call, the animation (once complete) triggers an event to say that it is complete, this event would be where one attempts to update the model (and possibly check for user error) which also has a `finishingAction` passed to it, this particular `finishingAction` would be a trigger to the start of the second animation. Here is a pseudocode example:

```
void animate(string animName, string opDetails, Action finishingAction)
{
    //run the assorted things necessary for the animation
    ..
    //Conclude with a call to the model
    Model.performOperation(opDetails, finishingAction, freePlay);
}
animate(“openBox”, “assignvalue x”, ()=>{animate(“closebox”);})
```

If one wanted to perform Model checks before the first of the aforementioned animations began, the API has features for querying the correctness of actions. One does not have to limit oneself to using just Actions, and instead one could use more complex delegates or even events to trigger this sequence of calls. Therefore the model of operation should impose no limitations on a non-event-driven implementation such as one made in XNA, due to the game loop constantly polling, which would allow for the simple raising of a boolean flag to indicate that some further operation needs to begin.

As the finishingAction ‘feature’ is there for convenience more than anything else, one can safely say that it imposes no more limitations than any other more ‘standard’ way of implementing an MVC system. Instead of automatically calling a continuation method, one would simply have to monitor for a raised boolean flag, or some other signal, to begin the next step. This applies to more than just the finishingAction example: wherever

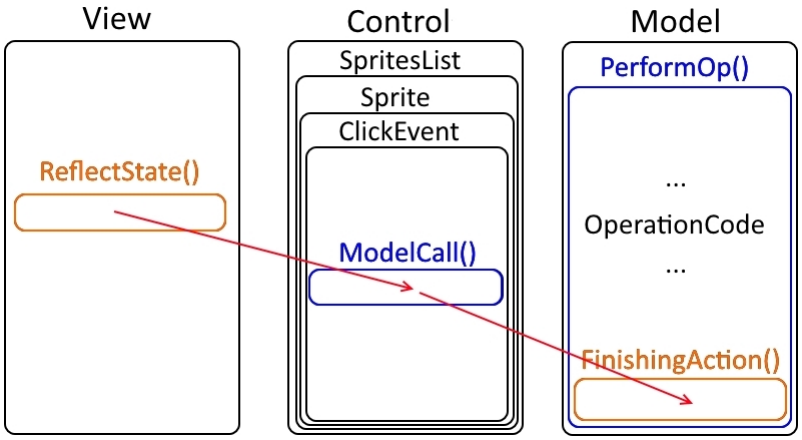


Figure 8.2: A pictographic representation of how the existing Model API simplifies View method calls. Where ReflectState is a general visual update method which belongs to View, and is passed as a delegate down to the FinishingAction associated with a specific sprite.

the underlying model uses continuation methods, they can be safely set to null without affecting the functionality of the model.

Additionally the API also allows for two special Actions that one can specify, and which will be called depending on whether an attempted operation was correct or not. When present, one of these two actions is called after finishingAction, and would allow for more granular control if needed. One might set finishingAction to null, and then specify a sequence of events that needs to take place based on whether the user was right or wrong.

One might even consider going further and leveraging the Model classes to make use of aggregation, reflection, and generics - thus allowing almost anything to be passed as a completion action (or continuation). This was not implemented, as it seemed like it added far to much complexity for any potential benefits it might offer.

Figure 8.2 gives a pictographic representation of how the event-action pairs example works, and Algorithm 8.1 gives a snippet of code demonstrating an actual sprite declaration along with its single call to the Model class. By using anonymous delegates (primarily through lambda functions) code clutter was kept to a minimum, as the system does not need an explicit global method declaration for every sprite's operations. This continuation-style functionality was already, strictly speaking, present in the XNA version. However, it was obfuscated somewhat by the fact that XNA is built on a game loop, and thus the alterations that a polling-API-variant would have required would not have seemed significant.

One final example of modularity and MVC compliance is the implementation independ-

Algorithm 8.1 An example of delegate usage to simplify control flow. This is the declaration and inclusion of a Return Button sprite.

```
//theMainSpritesList is of type Dictionary<string, Sprite>
theMainSpritesList.Add("returnButton",

    //parameters to do with positioning:
    new Sprite("unpresseddownarrow.png", 100, 722, 528,
    //This is the lambda to associate with the sprite click event:
    (object senderx, EventArgs ex) =>
        {
        //reflectModelState will be called by Model at the
        //end of the performOperation method call
        Model.performOperation("return",reflectModelState, freePlay);
        //'freeplay' determines whether or not sandbox mode is on
        }));
```

ence of the code interface - and while it is not strictly speaking a result of the WPF implementation, it was highlighted through the WPF development process. In short, total integration of the code interface into the WPF version took under five lines of code, making it a highly modular component - Section 8.2.2 gives more details.

8.1.3 XNA vs. WPF

After spending a great deal of time developing the original XNA version, the switch to WPF seemed (at least initially) as if it would be a rather daunting task that would take up far too much time and effort to implement anything more than the bare essentials. After just two weeks of programming the WPF version, the new implementation was as interactive as the original XNA version; the only things missing from the two-week version were the ability to load and follow levels, and the ability to guide the user with a hint system - implementing these two extra features took an additional week. The time-frame alone might be enough to convince some that WPF is better suited to the development of the 2D game, however one could argue that by implementing the XNA version first the author already knew what to look out for and what was needed. Therefore more detailed technical examples will be used to demonstrate how the WPF framework made implementation easier.

As both XNA and WPF are both built on the same underlying framework they both have access to the same features and functionality. However, there is a difference in the

relative convenience of flow of control for the programmer: XNA's game loop forces a 'flatter' approach, with more global variables and inverted logic encapsulated in classes - while WPF seems to be able to benefit more from the advanced power of delegation and continuation styles. These are powerful control flow and scope mechanisms, but they are somewhat ineffective in the 'flat' control structures that XNA imposes via its game loop.

There is one particularly good example that illustrates how convenient developing in WPF was: how more complex user input was handled via yet another application of continuation-passing [73](such as calculator equation entry).

Creating a textbox-style input mechanism in XNA (which is driven by a game loop rather than events) involves creating a new class and incorporating it into what quickly starts amounting to a very complicated finite state machine (especially when there is more than one type of input behaviour that needs to be monitored). In the XNA implementation, getting the user to input typed data without using the asynchronous XNA Guide requires, among other things:

- 15 lines of specialist code in the Update method.
- A global Cursor-Lock-Check enumeration that gets referred to in 14 different places (in everything from anonymous delegates to the primary Update class). This is to stop the user from interacting with other objects during input.
- A global string that keeps track of the user's input, which is also used in no less than 14 different places.
- A special method for checking the character-casing of user input (as XNA does not do a good job of checking for multiple pressed keys at once *automatically*. For example, `Keys.Shift&&Keys.A` needs to be interpreted as 'A', not 'a').
- And almost any other functionality one wants, which would already be built into the WPF TextBox class (such as interpreting a backspace).

For an example of how much goes into handling live textual user input in XNA, one can take a closer look at the `buildUpValueInputString` method, included in Appendix F (it is approximately 27 lines of code, just to append the correct input character). One *might* go so far as to say that this lack of UI components could be considered a weakness in the XNA framework when used to develop an application such as ActionWorld.

In the WPF implementation a pair of methods, called performOperationThroughTextBox and performOperationThroughMenu, were created to handle advanced input. They work by taking in positional data as well as the base information of the operation they needed to eventually perform, for example one might say:

`performOperationThroughTextBox("declareVariable ",200,0,0)` which would create a text box in the top left corner that waited for users to enter data, and when users pressed enter to complete the input process their final input would be appended to the base operation string and passed on to the Model. The steps involved from the presentation and execution perspectives are as follows:

1. The user clicks a sprite.
2. The Sprite's clickEvent is handled.
3. The clickEvent code calls the `performOperationThroughTextBox` method, using the details stored by the sprite's event handler as parameters (the text box does not exist yet).
4. `performOperationThroughTextBox` uses the View class to create a text box, and then associates an event handler with the new text box's keyPressed event (which will only trigger later).
5. The user focus is then placed on the newly created text box, and now the system has to wait for the user to enter details and press enter.
6. When the user presses enter, the text box's event handler reads the information inside the textbox, appends it to the base operation, and passes the operation details to the Models `performOperation` method.
7. Finally the event handler disposes of its parent object (the text box).

All of these steps are accomplished in about 14 lines of code, as shown in Algorithm 8.2 which gives the actual code for the performOperationThroughTextBox method, as well as a sample call to it. Once again, this makes use of the newer C# features of advanced abstraction and delegation, and strengthens the case that these features can make user interaction clean and almost effortless for both the programmer and the user.

Using the aforementioned numbers and code details, it appears as if the WPF version requires approximately five times less code to implement 'specialist' user input. This

Algorithm 8.2 An example of delegate usage to simplify control flow. This is the declaration and inclusion of a Return Button sprite.

```
//for brevities sake 'theMainSpritesList' has been shortened to 'sprites'
//and performOperationThroughTextBox has been shortenededed to 'performOp'
void performOp(string baseOp, int width, double xPos, double yPos){
TextBox temp = theView.addTextBox(width, xPos, yPos,

     (object o, KeyEventArgs s) =>
         {
             if (s.Key == Key.Return)
             {
               string finalContent = ((TextBox)o).Text;
               //The following two lines can be reversed, however this order
               //ensures that reflectModelState happens after the
               //new textbox has been disposed of.
               parentWindow.canvas_Main.Children.Remove(((TextBox)o));
               Model.performOperation(baseOp + finalContent, reflectModelState,
                                      freePlay);
             }
         }); //end of addTextBox method call

//minor aesthetic details:
temp.Background = new SolidColorBrush(Color.FromArgb(10, 10, 10, 10));
temp.FontSize = 20;
temp.Focus();
}
//an example of performOperationThroughTextBox in use:
sprites.Add("intpad", new Sprite("notepadpencil.png", 150, 10, 389,

     //parent referencing key press event handler:
     (object senderx, EventArgs ex) =>
         {
             performOp("directhandwrite int ", 120,
             sprites["intpad"].xCoord + 5, sprites["intpad"].yCoord + 30);
         }));
```

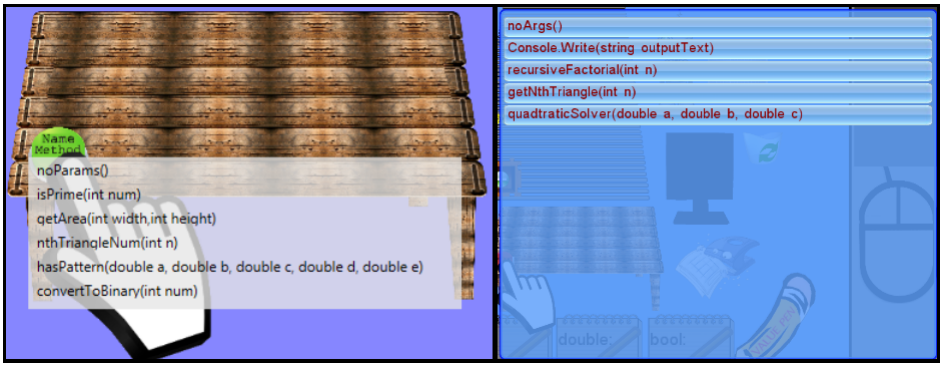


Figure 8.3: Left: non-intrusive built-in WPF menu. Right: intrusive custom XNA menu.

comparison should not be taken as a denouncement of XNA or game-loop frameworks in general, but instead as an illustration that under certain circumstances there are easier alternatives. Conversely the same is true of XNA over WPF in other situations - for example, rendering a 3D model in XNA is easier than in WPF.

A similar comparison of input mechanisms can be made for menu selection, however as the details of the WPF version do not differ significantly to the textbox input mechanism, it is sufficient to say that the XNA version of a menu required even more work than the version of textual input and wound up being far more intrusive than a dynamic WPF menu. Figure 8.3 shows one menu beside the other, and demonstrates just how different they are.

The above examples of ease of extensibility are fairly specific to the WPF implementation, however there are other extensibility examples that are not implementation specific. Details for these examples can be found in Section 8.2.

There are other less significant advantages of WPF over XNA (such as being able to build hints into the sprite's tooltips), however the author was only able to identify one serious advantage of using XNA over WPF for 2D purposes: dynamic loading and changing of sprite source images is easier in XNA than in WPF. The primary reason for this, however, is not a framework design issue, but is instead an implementation problem - when loading files for use as sprite source images, the built in Uri class caused several issues. The Uri class is supposed to be able to handle both relative and absolute file addresses simultaneously and correctly, however non-trivial relative addresses cause errors regarding files being incorrectly labelled as non-existent. After a great deal of effort was spent identifying the exact cause of the error, a work around was devised that utilised the System Directory class as a converter for relative addresses, as shown here:

```
string finalAddr = Directory.GetParent(address).ToString() +
    address.Substring(address.LastIndexOf('\\'));
bi.UriSource = new Uri(finalAddr, UriKind.Absolute);
```

If the Uri class worked correctly the following piece of code would have been the easiest way to load image sources:

```
bi.UriSource = new Uri(address, UriKind.RelativeOrAbsolute);
```

A final illustration of just how much easier and more succinct the WPF framework made things: when combining all the non-shared code for the two implementations (which is essentially the View and Control classes and subclasses) the XNA version came to about 4800 lines of code, while the WPF version was closer to 1000. Also, the largest collection of code overall - at approximately 4900 lines - was the combination of the model classes (the API deliverable).

A secondary take away from the details discussed in this section is how the switch from XNA to WPF opened the door to taking a higher-level abstract view of the control mechanism.

8.1.4 Not using XAML in WPF, a Design Choice

XAML (or Extensible Application Markup Language) is a declarative approach to defining the GUI, and is a central part of the WPF philosophy. But the WPF implementation did not make use of XAML in any significant way, this was done primarily to promote portability: by implementing everything in the WPF version procedurally and using just C# code to create GUI components, it ensures that the API could also be used by other GUI toolkits, new or old. For example, the code created for the WPF version could be altered to target a Windows Forms GUI, or perhaps GTK+ with relative ease (one would only need to make a few framework specific alterations as all the logic is framework independent) - an alternative example would be creating a web-based implementation of the proposed system.

In short, not using XAML is a design trade-off: by doing additional work in code, migration to other platforms such as the web, becomes easier. If the creation of the WPF

version had been done using XAML, some of the future portability would have been sacrificed. Also, one particular scenario might have caused problems: when there can be an arbitrary number of a particular sprite (such as the stack or variable sprites), implementing the creation of these run-time-dependant sprite instances becomes awkward in XAML.

8.2 Specific Technical Evaluation

This section gives specific examples from the API and game which demonstrate some desirable quality or other. As explained in Section 6.2.3 above, one such way to do this is to add a brand new feature to the existing system and explain and validate the process undertaken to implement such an extension - this process is used to demonstrate extensibility through the inclusion of the BoolEater and Undo-button features.

An alternative way of demonstrating program design quality is to take a specific (already implemented) piece of code or system component, and demonstrate how its original design proved its quality later in the development process. The code-interface form is a good example of this, more specifically regarding the modularity with which it was designed.

8.2.1 Completely New Functionality

This section gives two examples of functionality that was not originally present, but which was included later to improve the user's experience. Both examples also illustrate aspects of system extensibility.

8.2.1.1 Adding the BoolEater

During the hallway testing phase described in Section 5.4, the need for a conditionals mechanism became clear.

The implementation of this mechanism was a good test of the system's extensibility, because it introduces changes on almost every level, including: new visual components, new user interactions, control level alterations, and even changes to the API. Details of the process to include this feature follow.

The first step was to add the new sprite to the interactiveSpriteCollection object. It did not yet have any interactive functionality, nevertheless the game then had an in-game visual component in place that could be used for testing:

```
interactiveSpriteCollection.Add("booleater", new dynamicSprite...);
```

At the start of every level the sprites that are enabled are determined by the level in question. Unless this new sprite is included in one of the modes, it will not be enabled and thus will never draw or update. This feature also prevents invalid interactions when in a 'no-conditionals' level.

```
if (mode.Contains("conditional")
    enabledKeys.Add("conditionals");
```

Now the sprite will draw to the screen, however it has no behaviour associated with it yet. There are two behaviours that need to be exhibited by this sprite: it must only draw itself when the next operation in the current level is a conditional, and when clicked it needs to send a message to the world tracker (the control class) saying that it has received an interaction request. Before altering the control class, a dummy delegate was set up to check that the behaviour was correct - the enhancement to be made hides and shows the sprite, and takes place where the new sprite is originally added to the sprite collection:

```
Predicate<string> operationIsBooleanConditional =
    delegate(string key)
    {
        // Dummy temporary scaffolding:
        return DateTime.Now.Second % 2 == 0;
        //return theWorldTracker.isNextOperation("consumeBool");
    };
interactiveSpriteCollection.Add("booleater", new dynamicSprite...
    , operationIsBooleanConditional);
```

The above code results in the sprite hiding itself every other second, the commented-out line is what will be used later once the control class actually knows about the "consumeBool" operation. The next place holder delegate will eventually send messages to the world tracker, like the previous delegate the actual message line is disabled as it would send an unrecognised message:

```
Action consumeBool =
    delegate()
    {
        Console.Write("clicks are working");
        //theWorldTracker.performOperation("consumeBool", freePlay)};
    };
interactiveSpriteCollection.Add("booleater"...
    new interactiveRegion(... consumeBoolean...)...
    , operationIsBooleanConditional);
```

If this sprite did not require an alteration to the more fundamental behaviour of the system, the above enhancements would be enough to do a great deal already. However, as the inclusion of this feature requires alterations to the way the API behaves, changes need to be made in some other areas. The first step is to make the controller (the world tracker) recognise "consumeBool" as a legal operation:

```
case("consumeBool"):
    Console.WriteLine("No behaviour for this op yet");
    //succeeded = consumeBoolFromHand(finishingAction);
    break;
```

The above code fits in the switch block of the controllers performOperation method, which attempts to perform a given operation and return a boolean based on whether the operation was successful or not. The code does not have a definition for consumeBoolFromHand yet, and there are two ways one can do it: either using delegates or a normal method declaration. The important part of the controller's code for consumeBoolFromHand looks like this:

```
if (thePlayer.examineHeld().readType() != "bool")

    return false; //attempted an illegal action

thePlayer.removeHeld();
finishingAction();
```

The above code is surrounded by some standard exception handling. By this stage the new feature is almost complete, all that has to be done is to remove the appropriate place-holder lines and uncomment the actual checks. Now when one clicks on the sprite with a boolean in-hand, it will consume the boolean value and advance the current level to the next operation. The sprite will only display when the next operation is one which needs to consume a Boolean.

This enhancement took under 20 lines of actual code, spread over three areas. The only steps not included here are those required to integrate the new functionality with the existing levels and level editor (a minor task). This extension also had to be included in the WPF version. Because of the API extensions explained above, there was very little extra that needed to be included in the WPF version in order to include the BoolEater: no extra tracking needed to be included, and the only 'special' provision that had to be made for the BoolEater sprite was the inclusion of a conditional statement which either hides or shows the BoolEater depending on whether the user is holding a boolean value or not. In short, the WPF version required slightly less implementation-specific code to include the BoolEater than the XNA version did.

The takeaway from this feature addition: the system appears easily extensible down to an API level, with this new feature requiring just 20 new lines.

8.2.1.2 Implementing an Undo Button

Following the advice of one of the hallway testers, an undo feature was provided to allow users to take back an arbitrary number of actions.

Operations are not reversible in the API. The two obvious ways to implement UNDO are to keep all intermediate states of the virtual machine after every step, or the more simple method that was implemented: to record all user actions, and then replay all but one of them to a newly initialized machine. Therefore the first extension step was to include a list of completed operations in the WorldTracker Model class. The reason it is done

here and not in the level class (which already contains a list of completed and yet-to-do operations) is that this is a general piece of functionality that does not rely on the current level, especially if it should also work in sandbox mode:

```
public List<string> completedOperations = new List<string>();
```

Whenever one undoes an operation the system will essentially have to re-initialise the WorldTracker to its starting state and run through all the stored operations (except the last one). Therefore an 'undo' method needs to be included in the WorldTracker Model class, which does all of this:

```
public void undoLastOperation(bool freeplay, Action completionAction)
{
    if (completedOperations.Count > 0)

        completedOperations.RemoveAt(completedOperations.Count - 1);
    else

        //optimization in case no changes have been made yet:
        return;
    initialiseFields();
   //reload current level
    currentLevel.restartLevel();


    //most important part of the whole thing:
    foreach (string op in completedOperations)
        performOperation(op, freeplay);
    if(completionAction != null)
        //Perform the GUI update:
        completionAction();
}
```

Notice that the call to `performOperation` does not include a `finishingAction` parameter (which defaults to null) - this prevents the GUI from being incrementally updated, the call to `completionAction` at the very end updates everything at once. This feature prevents the user from seeing a fast replay of their actions, and also prevents unnecessary update tasks from impeding performance.

A set of delegates already exist that call set methods upon correct or incorrect operations, therefore the only change needed in order to start including operation tracking, is to add to the functionality of one of those. This could be done via the controller, however the better option is to set the add-new-completed-operation method call inside of WorldTracker, this way the undo functionality is built into the API rather than any single implementation. This extension simply requires that `addCompletedOperation(opAsString)` be included in two locations (one for sandbox play and one for regular levels)

At this stage the undo functionality is fully integrated into the model API, the only remaining step is to include a way for users to call the undo undoLastOperation method through a sprite or a button. This step is no different from any of the other sprites: it is simply added to the sprite collection and told that upon clicking it must call the undoLastOperation method.

Overall this extension required just one new method, and three extra lines spread across two Model methods, to incorporate it into the API, and finally two implementation specific lines to present the functionality to the user (closer to four lines for the XNA version). This situation serves to further illustrate two points that have previously been demonstrated: the proposed system is highly modular and easy to extend. Because of this extensibility and modularity, the potential downsides of iterative refinement and quick prototyping (namely that one might have to make quite deep changes) are not time consuming or costly.

8.2.2 The Code Interface Form, and the Modularity it Demonstrates.

The XNA framework provides somewhat of a “least common denominator” approach to its supported platforms: the Xbox , the PC, and some smaller mobile devices. But ActionWorld targeted only the PC. Even with the XNA version of ActionWorld, there was a hybrid approach that created a Windows Forms class called LevelCodeInterface. Its sole responsibility was displaying level-code details to the user. This class was created

outside of the XNA framework (and was thus not portable to an Xbox), because of how difficult implementing simple form functionality is in XNA. This hybrid design decision turned out to have unforeseen advantages later.

As explained in Section 8.1.3, once all the basic user-interaction behaviour was in place for the WPF version, level functionality needed to be included. But rather than creating a WPF specific version of something that was already available, it was decided to once again mix the different technologies, and use the Windows Forms component together with the WPF version. Creating the hybrid of WPF and the code interface (LevelCodeInterface), a Windows Form class, took two lines to include and draw in the WPF version (with no real functionality), two lines to add level loading functionality, and one line of code to make the form reflect the state of the current level and move along with the user's interactions. By including only five additional lines of code, the just-functional shell - without purpose, or guidance of any kind - was changed into a game with meaning, direction and goals.

While creating the WPF version the author had not pre-planned for the integration of the LevelCodeInterface class, simply because the features could be added to the WPF form itself, which means that no special provisions were made for its inclusion (this is highlighted this to make it absolutely clear just how modular this particular class is).

Another point of interest relating to the LevelCodeInterface and level loading is that of cross-cutting concerns (an issue explained best through Aspect Oriented programming theory [74]): The LevelCodeInterface has no idea how to load a level, but does know when a user clicks on the level-load menu button; the converse is true of the control class, which knows exactly how to load a level from a given directory, but does not know *when* to load a level. Rather than trying to give each class extra responsibilities (which shouldn't belong to those classes anyway) the author was able to minimise the effect of the level-loading cross cutting concerns by creating an event handler inside of Control but associating it with an event that belongs to the LevelCodeInterfaces class. The following piece of expanded code (present in the Control class as just two lines) is all it took to include level loading functionality into the WPF implementation *and* avoid common complications associated with cross cutting concerns:

```
textualInterface.openLevelToolStripMenuItem.Click += new EventHandler(
   (object sender, EventArgs e) =>

    {

        clearCanvas();
```

```
            LoadNewLevel(textualInterface.levelToOpen, ref Model);
            reflectModelState();
        });
```

This modularity is something of a design trade-off: the system needs two modules (LevelCodeInterfaces and the main WPF window) instead of just one, as well as functionality to facilitate communication - however what is bought for the price of this extra work is additional separation of concerns. Once again, this seems to illustrate the recurring theme that the extended C# power of events and delegates can substantially enhance and simplify the coupling between components.

This last observation applies to both to the XNA and WPF versions of this system, and simply serves to further illustrate the modularity and MVC compliance of the assorted classes. The author believes that this design choice is ultimately worthwhile, as it means that any future implementations can make immediate use of the existing class (even if such usage is only temporary, while developers create their own version).

8.3 Summary

This chapter went into a fair amount of technical detail in order to demonstrate aspects of the API and game implementations, which seem to be indicative of quality. The techniques ranged from demonstration of MVC adherence to reduction of cross-cutting concerns via continuation passing (a technique that could potentially be used in a variety of other applications to simplify GUI development, and improve functionality).

The evidence gained from this system suggests that coupling the MVC architecture with newer and more powerful features of delegates and events in C# has substantial advantages. Furthermore, it is not necessary to choose any single candidate between the competing technologies like Windows Forms, XNA and WPF. The technologies can be successfully intermingled, and each can be exploited for its strengths and should perhaps be avoided for its areas of weakness when appropriate.

The next chapter elaborates on how one might extend ActionWorld even further (both its theoretical and technical components).

Chapter 9

Future Work

There are a large number of ways in which this work could be altered, enhanced, or extended in the future. This chapter describes some of the ways one might extend the proposed system, or parts of it.Section 4.4 elaborated on enhancements to the ActionWorld system that had been attempted or investigated to a far greater degree than the ideas proposed in this chapter. If an alteration or enhancement would require changes to multiple components (for example, both the API and game implementation), its description will be included in the more fundamental of the sections.

There are multiple possible future extensions that can be added to the system with varying degrees of difficulty. For example, one could use this system with more than one language (which would require no alterations other than the creation of new levels), one could alter the display of code to be visual rather than textual (such as through flowcharts), or one could alter the way players interact with the metaphors (either simply by changing the appearance of the sprites, adding drag and drop interactions, or including more sprites to allow for greater detail).

9.1 Enhancing the Metaphors

Alterations to the metaphors have been possible from the start, as demonstrated through the use of an iterative methodology and hallway tests. These alterations could be purely aesthetic, or they could attempt to incorporate more advanced concepts. A simple example of a non-aesthetic enhancement would be to include polymorphic concepts, such as inheritance, generics, and aggregation - this particular example would be useful up to

a point, however, as the system targets novice programmers (and these concepts are not at a novice level), their inclusion could be moot.

With the above example in mind, any enhancements to the metaphors need to be weighed up against their usefulness given the target demographic.

9.2 Enhancing the API

There are multiple ways one could extend the system's API, a simple example of one such extension could be a feature that allows users to deem a line of code invalid, and pass over it. This sort of extension would not be difficult to include, however it does run the risk of causing students to believe that the language is interpreted or that the computer can detect and skip invalid code at run-time. For these reasons this extension was not included. According to Garner et al. [75], syntax errors cause the most problems for novices. Provided students are warned at the outset that the code they are looking at might not run on a real system, one might be able to provide a mode with invalid code to strengthen syntax skills as well as semantic skills.

Another possible addition would be to allow starting a level in a partially complete state. The existing *undo* mechanism is ideal for this: by pre-canning several opCodes that are performed automatically when starting a level, the user could be asked to compare the already executed code with the current state and draw conclusions from that. An example where this might be useful, is to start a level with several iterations of a loop complete, and then the user would have to gauge for themselves when the loop needs to quit based on the state of whatever is being used as a conditional.

Certain simple operations could get tedious if done too often; one might overcome this sort of issue by including macro-like operations: for example, once a user has proved that they can do single line variable declarations, expression evaluation and value-variable assignment (such as int x = y * 3;), they could click on a variable declaring shortcut that would ask for the type, name, and value all in one step. Another example would be that of method calling: once a user has proved that they can call methods with arguments, a macro-sprite could be used to auto-assign values and perform the call all at once, after the user has selected the method they want to call. Macros should only be included after users have proved they are able to do a certain sequence of operations, otherwise the meaning behind the metaphors could become unclear.

9.3 Enhancing the Finished Games

During the course of this work a number of potential metaphors and manipulatives were developed (and each one had various possible implementation methods), thus an enhancement that allows different sets of metaphors to be selected would allow for easy testing of metaphor validity and fidelity as well as allowing a change of perspective. This sort of thing could be done in a purely aesthetic manner, where the only alteration is the appearance of the sprites. The MVC design of the manipulatives API is highly conducive to this kind of enhancement.

Several systems use non-textual methods of representing code, for example flowcharts and images. With this in mind, and considering the modularity of the code interface class, one could extend the system to allow for alternative methods of code representation. The biggest benefit to this kind of extension (if it could be done in such a way that users can switch between representation), would be that if users get stuck on a particular piece of code, they can simply swap representations in order to see things differently. Extra representations could be generated procedurally (deriving representation from code), or they could be done manually by altering the Level and CodeInterface classes to include the extra representations. Procedural generation would be more technically challenging, but would save time when it comes to creating levels.

9.4 The Test Framework and More Rigorous Evaluation

A potential test that was not included in the framework would be to give the API (and the accompanying metaphor set) to some intermediate game development students and ask them to implement a simple version of the game: this would serve to further test the robustness and ease of use of the API.

A valuable piece of future research (which relates directly to this work), would be the performance of a rigorous marks-driven evaluation such as that described in Section 6.1. Such a test would lend objective support to the applicability of this work in real teaching environments, and on real students.

9.5 Enhancing the Level Editor

While the implementation of a level editor has made the creation of custom levels much easier than it was originally (where one had to create them one line at a time in an XML file), the process could still be improved upon: implementation of an automatic code interpreter, as described in Section 5.1.2, would be the most obvious improvement. An extension of this nature would allow not only experienced users to create levels, but would also allow novices to step through code which they would like to understand via the game. This extension *could* be relatively easy if one were to use the .NET Compiler Platform (also known as "Roslyn"), whose design specification includes Compiler as a Service - this feature gives users direct access to semantic and lexical code analysis, and dynamic compilation to CIL [76, 77]. Being able to create IL from arbitrary code, and then interpret that IL as level opCodes, would effectively automate the creation of levels.

With the current generic nature of the metaphors and environment there is no real call for any sort of procedural map or level generation, but perhaps generation of hints and tips based on the next action could be included. If one were to implement a level-code interpreter, *and* create an alternative representation of the game world (such as the earlier code-as-maze representation), then procedural level generation would become a must have.

9.6 Other Enhancements

Another future enhancement (or even separate project), could be to include the mazes and maps that were proposed during the early developmental stages; the focus of such a project would be on teaching flow of control over flow of data. This sort of alteration would greatly benefit from procedural map generation, as explained in the previous section.

The survey of different programming environments and languages performed by Kelleher and Pausch [4] says a lot about the development of programming-oriented education systems, however the most 'modern' game they surveyed was released in 2002, and it might be interesting to perform a similar survey of games released since then, in order to better understand the developmental history of these systems. A survey of this nature might also serve to highlight potential areas of improvement for systems such as this one.

9.6.1 Reverse-Mode Code Generator for Sandbox Play

In short, this potential feature would try to generate code that corresponds with the user's metaphor interactions while in sandbox mode. The idea behind this potential enhancement is that it would encourage exploration of the various metaphors, and exploration is an excellent way to learn. So far as the author could tell, no systems exist that allow the user to both create code through graphical interaction, *and* interpret code using those same actions (Gilligan's [5] system does the former, but not the latter), making this a potentially very novel feature.

Unfortunately there are several major drawbacks to implementing something like this with ActionWorld's proposed metaphors. The first and most obvious one is that not all actions have a single code equivalent, for example `int x = 3 + 3` would take three steps in this system (when using the notepads rather than the calculator), of those three steps the middle one (write the integer 6 on a notepad) would not generate code immediately, and when it did the generated code would look like this: `int x; x = 6;` - the 3+3 part is lost. Other more serious examples like this one would be "what is in the uncalled part of an if-else?" or "does the Boolean the user just gave to the BoolEater belong to the conditional of an if, a while, or a for?". Gilligan encountered these issues, and tried to avoid them by not including loops in the inferred code.

If one were to overcome these limitations, the resulting system could create even more serious conceptual-risks for the user: because a student would be creating code for one specific run of a program, without considering alternative conditions, all but the most trivial of programs run the risk of encouraging students to program specific, rather than general, solutions.

If investigated as a separate piece of work, rather than an extension to this system, the idea might be worth investigating further (provided a powerful enough inference engine could be created). After all, there are plenty of systems that let the user arrange objects and use nice GUIs to simplify the process of creating unambiguous programs, but few (if any) of them show the code that would match what the user has done, which is a learning method through which some people might flourish.

9.7 Summary

This chapter explained a number of ways in which this system as a whole (and its constituent components), could potentially be enhanced or used in future related work. Potential future work ranges from simply adding to (or enhancing) the proposed metaphor set, to performing a more in-depth evaluation of the system as a whole in order to determine it effectiveness at teaching novices. The next chapter summarised this dissertation, and enumerates the various conclusions that where made based on the work.

Chapter 10

Conclusion

10.1 Introduction

There is ample evidence that weak mental models of how code executes leads to difficulties in learning to program. Visualisation of concepts and data can be highly beneficial (as evidenced in Sections 2.2 and 2.5, among others). Metaphors are also a powerful and widely used tool, but investigation showed that they are often designed in isolation and only address a narrow version of the problem. For example, the simplest 'variable as a box' metaphor might represent variables well enough until the variable needs to be passed by reference, at which point one has to somehow incorporate memory and pointers (see Section 2.3 for more examples). The inherent risk of using 'weak' metaphors is that they can break down or mislead the learner when applied more generally, potentially leading to flawed mental models.

By combining the visualization and use of interactive metaphors, the author set about constructing a system that was believed could aid in the teaching of novice programmers. It attempts to give learners a high-fidelity mental model for various fundamental concepts.

The difficulties associated with learning to program (and how to teach programming) are extensively documented, however no stand out solution to either problem has been established yet. In was decided that the system should be built so that it emphasizes the users' active manipulation of the concrete mechanisms of code execution.

10.2 Contributions of the Dissertation

Through the developmental process of theoretical metaphor design, all the way to proof of concept game implementations, several important artefacts, theories, and frameworks were created:

- First and foremost this work proposed an unusual programming teaching technique: visual demonstration of code from the perspective of the computer, in order to improve mental models and understanding. This teaching technique encourages "active participation" on the part of the students, which should hopefully improve their engagement and thus their understanding. In theory this teaching technique might also be applied to other subjects (however investigating that line of thought would go well beyond the scope of this dissertation).

- A set of unified metaphors was created. Of the literature examined, it appears as if none of the authors in question have created or proposed a set of unified real-world metaphors for representing the state of an executing program. Gilligan [5] comes closest to creating something like this, however potential weaknesses are present even before one attempts to extend his metaphors, and even more appear after trying to include extensions. See Sections 2.6 and 2.3.3 for more on Gilligan's set and metaphor limits respectively. While this systems proposed metaphors are specific to teaching programming, the theory behind them might also be applied to other abstract subjects.

- A modular API (or virtual machine) was developed to simplify the task of implementing alternative future versions of this system (whatever the purpose of those implementations might be). Chapter 5 goes into the details behind the API's design and implementation.

- Two complete, usable, games were created, which could be used by future researchers to test the impact of this sort of teaching mechanism and which also served to demonstrate and improve both the metaphors and the API. Details regarding improvement based on the early implementation can be found in Section 5.4.3, and Chapter 5 elaborates on game design and implementation.

- A level editor which simplifies things for future users, by making level creation almost trivial. This allows for curriculum (and even language specific) customisation of the game levels, and therefore also makes more rigorous testing easier.

- The use of a 'constituent evaluation' test framework, which Chapter 6 explains.
- A suite of sample tests that conform to the aforementioned test framework, and which (simply through their successful application) serve to prove the validity of such an evaluation technique. Chapter 7 elaborates on specific test results.
- The WPF game, and the underlying API, both demonstrated the potential power of combining the newer, more advanced, features of C# with the older MVC architecture. More specifically the reduction of cross cutting concerns brought about through the use of continuation passing, as well as the ability to analyse and develop at a higher level of abstraction. Chapter 8 contains several examples of how continuation passing mixed well with MVC in order to simplify development.
- It was shown that one can take advantage of the strengths of certain technologies (while bypassing potential weaknesses), through careful hybridization of those technologies. Section 8.1.3 gives examples of how one might take advantage of WPF in order to bypass XNA's occasionally restrictive game loop (in order to improve the user's experience), while Section 8.2.2 shows how Windows Forms can be merged almost seamlessly with both XNA and WPF in order to add functionality that may have been difficult to include otherwise.

10.3 More Detailed Conclusions

10.3.1 The Metaphors

After following a well-defined iterative methodology, and discarding no less than 50 different potential metaphors, a set of nine distinct foundation-analogies were decided upon. The process undertaken to finalise these metaphors involved in-house experimentation, extensions to pre-existing metaphors, and refinements through user testing (Section 3.2.1 explains the development process in full). The final set of metaphors are platform independent and have a certain level of customisability to cater for more specialist situations (Section 4.1 describes the finalised set in full).

When examining the metaphors, one is able to see their potential versatility, especially in the area of incorporating more complex concepts which they were not originally designed to accommodate (such as Enumerators). This capacity for complex extensions led to the

conclusion that the fundamental metaphors offer a high-fidelity way of visualising program execution (more examples can be found in Section 4.4.4).

While at times the proposed analogies sacrificed accuracy for understandability (for example, there is no use of registers), these compromises arguably did not detract from the user's experience, and instead often enhanced it - Sections 4.3 and 4.4 go into more detail regarding analogy alternatives and improvements. This leads to the conclusion that the metaphors have likely achieved a good balance regarding detail levels, considering that more detail (while usually more accurate) increases the risk of overwhelming users (but at the same time nothing is *completely* hidden from the user, especially when it might give them a simplified understanding).

10.3.2 The API

The metaphors were demonstrated through two separate game implementations, both of which were built on the same API: essentially a virtual machine that took on the role of background state tracker. A thorough technical evaluation of the API - using techniques such as 'from-scratch' extra implementation, new feature inclusion, and paradigm adherence analysis - led the author to conclude that the API met all of the following criteria:

- Easy to understand (it is mostly self-managing, with a few key methods that need to be called externally).
- Easy to use (a fully functional 2D game can be built on it in less than two weeks).
- Extensible (easy to add new features).
- Modular (components can be easily replaced).

For more details on the technical evaluation, see Chapter 8.

10.3.3 The Test Framework

Conclusions regarding the proposed student-as-interpreter teaching technique are hard to come by without performing further tests. In order to address this, not only was a

'constituent evaluation' technique used, but several sample tests were included which one might undertake in order to further investigate the effects that the proposed teaching method might have on students. Chapter 6 elaborates on the test framework itself, as well as several samples.

Despite not applying comprehensive tests to every element of the system, several conclusions could still be drawn out via the application of key tests from the test framework (for more details on the application of these tests see Chapter 7). In short, the theoretical test framework has been satisfactorily validated both in principle and in practice (insofar as evaluating systems such as ours is concerned).

10.3.4 The Games

Much like drawing conclusion for the proposed metaphors, there is only so much one can say about what the two game implementations might offer students (at least without performing more tests). What one *can* do regarding the games, is apply in-house tests and perform comparative evaluations on them (as shown in Chapter 7). Based on tests such as these, it was concluded that this system has more features than the majority of those evaluated by Kelleher and Pausch [4] (Section 7.1). Of those system evaluated by Kelleher and Pausch, the proposed system outperformed the one most similar to itself (Gilligan's[5] system) on almost every applicable metric (see Section 7.3.1).

When looking at the two prototype games created for this dissertation as conceptual demonstrators (rather than end-user games), one is able to conclude that they definitely achieve that goal: they are able to demonstrate both the quality of the API, and the fact that the metaphors are not only usable in games, but are also platform independent (based on the static nature of the demo games). Therefore an unexpected conclusion one might reach is that for the duration of this research, the games served more as in-depth testing mechanisms than as actual games, especially considering how few people played them during the course of this work. (See Section 5.4 on hallway testing for detail regarding this).

10.4 In Closing

Both teaching and learning programming is difficult, and as such there is an abundance of material that aims to assist both teachers and students. Despite this, no single stand-

out solution to these issues has emerged. To the field of programming education, this work emphasizes an alternative teaching technique that neatly combines multiple teaching theories. It also provides the tools required to implement, test, and further explore this approach.

Bibliography

[1] Mike Bostock. Visualizing algorithms. Website - http://bost.ocks.org/mike/algorithms/, June 2014. Last Accessed 2014-07-02.

[2] Nikolas S. Boyd. Software metaphors. Website - http://www.educery.com/papers/rhetoric/metaphors/, 2003. Last Accessed 2014-07-07.

[3] M.R. de Villiers and P.A Harpur. Design-based research - the educational technology variant of design research: Illustrated by the design of an m-learning environment. In *Proceedings of SAICSIT 2013, Annual Research Confrence of the South Africa Institute of Computer Scientists and Information Technoligists*, pages 252–261, 2013.

[4] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137, June 2005.

[5] David Gilligan. An exploration of programming by demonstration in the domain of novice programming. Master's thesis, Department of Computer Science, Victoria University, Wllington, Victoria, August 1998.

[6] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.*, 39(1):499–503, March 2007.

[7] Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the effectiveness of a new instructional approach. *SIGCSE Bull.*, 36(1):75–79, March 2004.

[8] Wanda Dann, Stephen Cooper, and Randy Pausch. Making the connection: Programming with animated small world. In *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*, pages 41–44. Press, 2000.

[9] TurtleAcademy. Turtleacademy lessons. Website - http://turtleacademy.com/learn.php. Last Accessed 2014-07-05.

[10] Lawrence Snyder. Cs principles pilot at university of washington. *ACM Inroads*, 3(2):66–68, June 2012.

[11] Lindsey Ann Gouws. The role of computational thinking in introductory computer science. Master's thesis, Rhodes University, 2013.

[12] D. G. Jones and A. K. Dewdney. Core war guidelines. Website - http://corewar.co.uk/cwg.txt, March 1984. Last Accessed 2014-07-10.

[13] John Maloney, Kylie Peppler, Yasmin B. Kafai, Mitchel Resnick, and Natalie Rusk. Programming by Choice: Urban Youth Learning Programming with Scratch. *SIGCSE*, pages 367–371, 2008.

[14] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, November 2009.

[15] Michael Tempel. Blocks Programming. *CSTA Voice*, 9(1):3–4, March 2013.

[16] Steve Klabnik. Hacketyhack lessons. Website - http://hackety.com/lessons, April 2013. Last Accessed 2014-07-03. Alternative source - https://github.com/hacketyhack/hackety_hack-lessons.

[17] J. F. Pane and B. A. Myers. Usability Issues in the Design of Novice Programming Systems. Technical report, Carnegie Mellon University, August 1996.

[18] Bret Victor. Learnable programming - designing a programming system for understanding programs. Website - http://worrydream.com/LearnableProgramming/, September 2012. Last Accessed 2014-06-19.

[19] Marc Eisenstadt, Blaine A. Price, and John Domingue. Software visualization as a pedagogical tool: Redressing some its fallacies. Technical report, ITS Fallacies, Instructional Science, 1992.

[20] Edward R. Tufte. *The Visual Display of Quantitative Information.* CT: Graphics Press, 1983.

[21] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, pages 161–165, New York, NY, USA, 2009. ACM.

[22] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, 26(9):677–679, September 1983.

[23] Tina Götschi, Ian Sanders, and Vashti Galpin. Mental models of recursion. *SIGCSE Bull.*, 35(1):346–350, January 2003.

[24] Richardo Jiménez-Peris, Cristóbal Pareja-Flores, Marta Patiño Martínez, and J. Ángel Velázquez-Iturbide. The locker metaphor to teach dynamic memory. *SIGCSE Bull.*, 29(1):169–173, March 1997.

[25] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM.

[26] Mark Guzdial. Programming Environments for Novices. In Sally Fincher and Marian Petre, editors, *Computer Science Education Research*, pages 127–154. Taylor & Francis Group, plc, London, UK, 2004.

[27] Janet Rountree and Nathan Rountree. Issues regarding threshold concepts in computer science. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 139–146, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[28] Lynn Schofield Clark. Critical theory and constructivism: Theory and methods for the teens and the new media @ home project. Webpage - http://www.ihrcs.ch/?p=92, 2002.

[29] Patricia Moyer, Johnna Bolyard, and Mark Spikell. What Are Virtual Manipulatives? *Teaching Children Mathematics*, 8(6):372, February 2002.

[30] Allan Collins. Toward a design science of education. Technical Report 1, Center for Technology in Education, New York, NY., http://www.eric.ed.gov/PDFS/ED326179.pdf, January 1990.

[31] Paul A. Kirschner, John Sweller, and Richard E. Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, 41(2):75–86, June 2006.

[32] Charlotte Hua Liu and Robert Matthews. Vygotsky's philosophy: Constructivism and its criticisms examined. *International Education Journal*, 6(3):386–399, 2005.

[33] Miguel Baptista Nunes and Maggie McPherson. Constructivism vs. objectivism: Where is difference for designers of e-learning environments? In *Proceedings of the The 3rd IEEE International Conference on Advanced Learning Technologies*, July 2003.

[34] David A. Kolb, Richard E. Boyatzis, and Charalampos Mainemelis. Experiential learning theory: Previous research and new directions. In *Perspectives on cognitive, learning, and thinking styles*, August 1999.

[35] Daniel Coyle. *The Talent Code: Greatness Isn't Born. It's Grown. Here's How.* Random House Publishing Group, April 2009.

[36] Shane Parrish. How people learn. Website - http://www.farnamstreetblog.com/2013/01/how-people-learn, January 2013. Last Accessed 2014-06-19.

[37] Bradley Miller and David Ranum. *Python Programming in Context - Second edition.* Jones & Bartlett Learning, 2014.

[38] Peter Wentworth. Personal Correspondence, February 2013. Email correnspondence between author and Peter Wentworth.

[39] Jean-Baptiste Huynh and Patrick Marchal. Dragonbox algebra. Webpage - http://dragonboxapp.com/story.html, 2012.

[40] Jordan Shapiro. It only takes about 42 minutes to learn algebra with video games. Website - http://www.forbes.com/sites/jordanshapiro/2013/07/01/it-only-takes-about-42-minutes-to-learn-algebra-with-video-games/, July 2013.

[41] Mick Flanagan. Threshold concepts: Undergraduate teaching, postgraduate training and professional development. a short introduction and bibliography. Website - http://www.ee.ucl.ac.uk/ mflanaga/thresholds.html, August 2014. Last Accessed 2014-08-19.

[42] Jan Meyer and Ray Land. Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within the Disciplines. Website - http://www.etl.tla.ed.ac.uk/docs/ETLreport4.pdf, May 2003. Last Accessed 2014-07-16.

[43] Michael Edmonds. Threshold concepts and troublesome knowledge - an introduction. Website - http://sciblogs.co.nz/molecular-matters/2010/11/12/threshold-concepts-and-troublesome-knowledge-November 2012. Last Accessed 2014-07-16.

[44] Carol Zander, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Kate Sanders. Threshold concepts in computer science: A multi-national empirical investigation. *Threshold Concepts within the Disciplines*, 16:105–118, 2008.

[45] Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. Threshold concepts in computer science: Do they exist and are they useful? *SIGCSE Bull.*, 39(1):504–508, March 2007.

[46] R. McCartney and K. Sanders. What are the "threshold concepts" in computer science? In T. Salakoski and T. Mäntylä, editors, *Proceedings of the Koli Calling 2005 Conference on Computer Science Education*, Baltic Sea '05, page 185, November 2005.

[47] Lynda Thomas, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, and Carol Zander. A broader threshold: Including skills as well as concepts in computing education. In Mary O'Rourke Catherine O'Mahony, Avril Buchanan and Bettie Higgs, editors, *Fourth Bienniel Conference on Threshold Concepts: From personal practice to communities of practice*, pages 154–158. NAIRTL, January 2014.

[48] Will Thalheimer. People remember 10%, 20%...oh really? Website - http://www.willatworklearning.com/2006/10/people_remember.html, October 2006. Last Accessed 2014-06-28.

[49] D. G. Treichler. *Are you missing the boat in training aids?*, chapter 1, pages 14–16, 28–30, 48. Film and Audio-Visual Communication, 1967.

[50] Jorge F. Trindade, Carlos Fiolhais, Victor Gil, and José C. Teixeira. Virtual environment of water molecules for learning and teaching science. In *Graphics And Visualization Education*, Coimbra, Portugal, July 1999.

[51] Nico Rutten, Wouter R. van Joolingen, and Jan T. van der Veen. The learning effects of computer simulations in science education. *Computers & Education*, 58(1):136 – 153, 2012.

[52] Marco Cantu. Of Strings, Immutability, COW, and AnsiStrings. Website - http://blog.marcocantu.com/blog/strings_immutability_cow_ansistrings.html, May 2013. Last Accessed 2014-07-07.

[53] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, September 1986.

[54] Joel Spolsky. The Joel Test: 12 Steps to Better Code. Webpage - http://www.joelonsoftware.com/articles/fog0000000043.html, August 2000.

[55] Tel Amiel and Thomas C. Reeves. Design-based research and educational technology: Rethinking technology and the research agenda. *Educational Technology & Society*, 11(4):29–40, 2008.

[56] T. C. Reeves. Design research from the technology perspective. *S. McKenney, & N. Nieveen (Eds.), Educational design research*, pages 86–109, 2006.

[57] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and Logic, Fifth Edition.* Cambridge University Press, 2007.

[58] J.S. Armstrong and T.S. Overton. Estimating nonresponse bias in mail surveys. *Journal of Marketing Research*, 14:396–402, August 1977.

[59] Bruce L. Berg and Howard Lune. *Qualitative Research Methods for the Social Sciences.* Pearson Education Limited, 2011.

[60] Kasper Hornbæk. *Maturing Usability - Usability Evaluation as Idea Generation.* Springer London, 2008.

[61] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[62] David Squires and Jenny Preece. Predicting quality in educational software: Evaluating for learning, usability and the synergy between them. *Interacting with Computers*, 11(5):467–483, May 1999.

[63] Rachelle S. Heller. Evaluating software: A review of the options. *Computers and Education*, 17(4):285–291, December 1991.

[64] Office of Technology Assessment U.S. Congress. Power On! New Tools for Teaching and Learning. OTA-SET-379 (Washington, DC: U.S. Government Printing Office), September 1988.

[65] David Squires and Anne McDougall. *Choosing and Using Educational Software: A Teachers' Guide*. Falmer Press, London, 1994.

[66] J. Winship. Software review or evaluation: Are they both roses or is one a lemon? In *Proceedings of the Australian Education Conference, Perth*, 1988.

[67] Guidelines for the evaluation of instructional technology resources for california schools, 1997.

[68] D. Allen and K. Tanner. Rubrics: tools for making learning goals and evaluation criteria explicit for both teachers and learners. *CBE Life Sciences Education*, 5:197–203, 2006.

[69] Jakob Nielsen. *Usability Inspection Methods*. John Wiley & Sons, New York, NY, 1994.

[70] Thomas C. Reeves, Lisa Benson, Dean Elliott, Michael Grant, Doug Holschuh, Beaumie Kim, Hyeonjin Kim, Erick Lauber, and Christian S. Loh. Usability and instructional design heuristics for e-learning evaluation. In P. Barker and S. Rebelsky, editors, *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2002 (pp. 1615-1621).*, Chesapeake, VA, 2002. AACE.

[71] South Africa Council on Higher Education. Framework for institutional quality enhancement in the second period of quality assurance. *Institutional Audits Directorate*, pages 7–8, February 2014.

[72] Dave Moursund. *Computational Thinking and Math Maturity: Improving Math Education in K-8 Schools*. University of Oregon, June 2006.

[73] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3-4):233–247, 1993.

[74] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer-Verlag, June 1997.

[75] Sandy Garner, Patricia Haden, and Anthony Robins. My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ACE '05, pages 173–180, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[76] Neil McAllister. Microsoft's Roslyn: Reinventing the compiler as we know it. http://www.infoworld.com/article/2621132/microsoft-net/microsoft-s-roslyn–reinventing-the-compiler-as-we-know-it.html, October 2011. Last Accessed 2014-09-20.

[77] Alex Turn. Welcome to the .NET Compiler Platform ("Roslyn"). https://roslyn.codeplex.com/, August 2014. Last Accessed 2014-09-20.

Appendix A

In-Game Screenshots

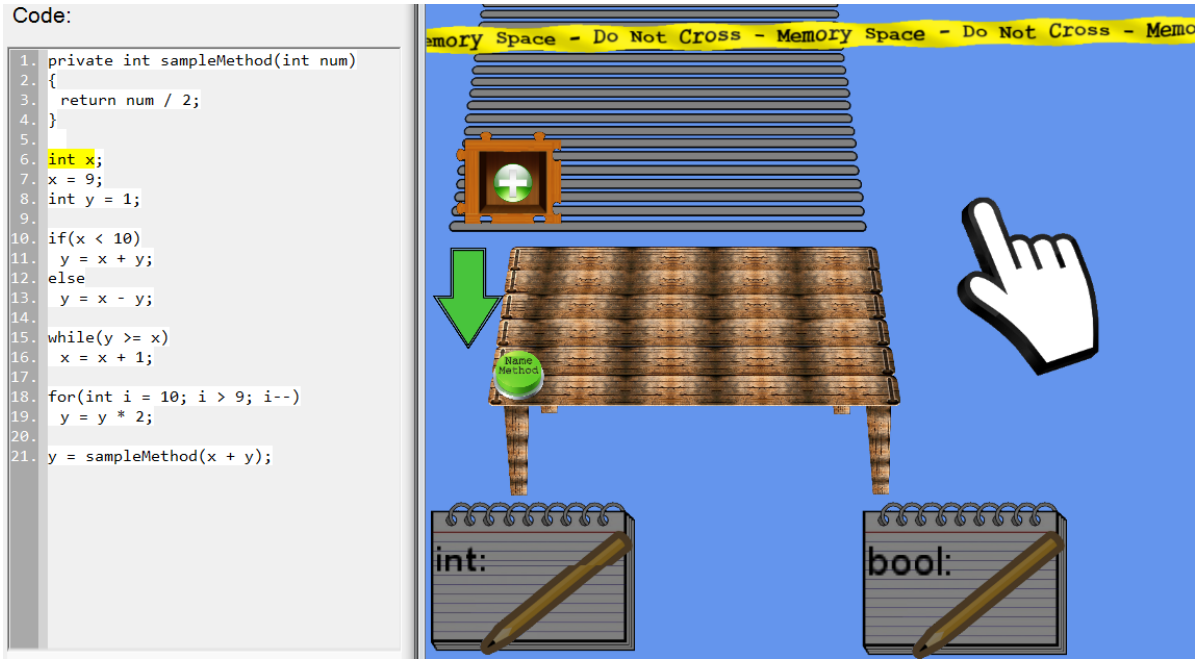


Figure A.1: The start of a sample level. The user has not done anything yet, and is being prompted to create a new integer variable 'x' (highlighted in yellow on the left).

Figure A.2: The user has succesfully declared the integer 'x' which currently has no value. The user is currently creating an integer value via the int notepad.

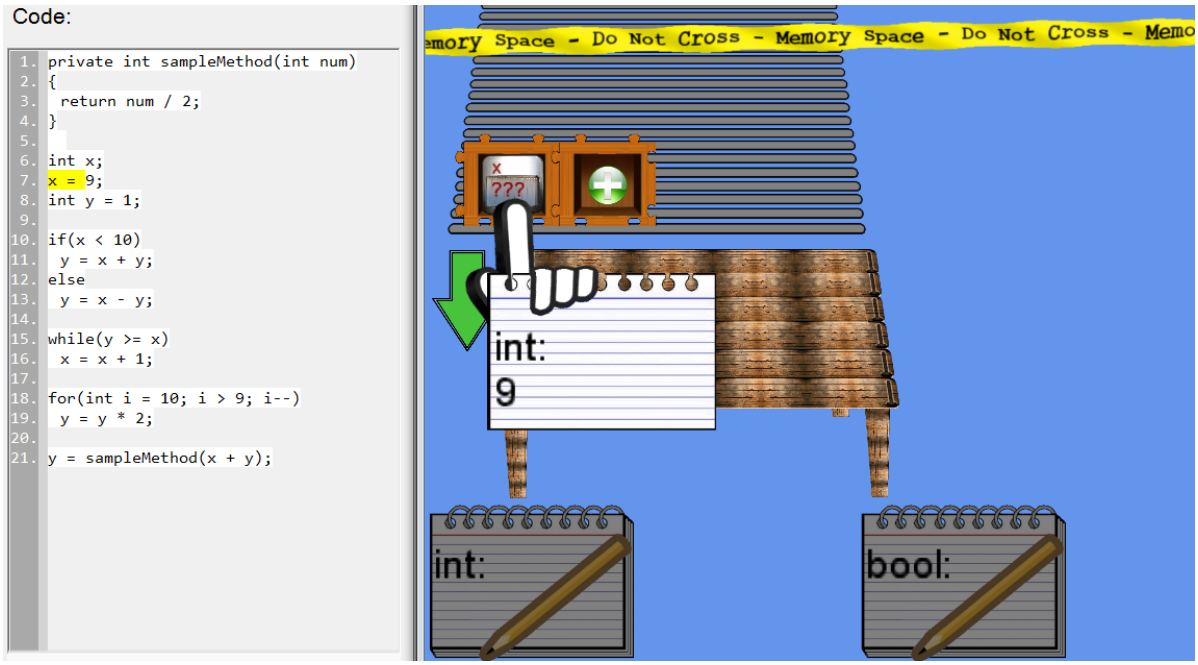


Figure A.3: The user is now holding the newly created value, and is about to assign it to the integer variable 'x'.

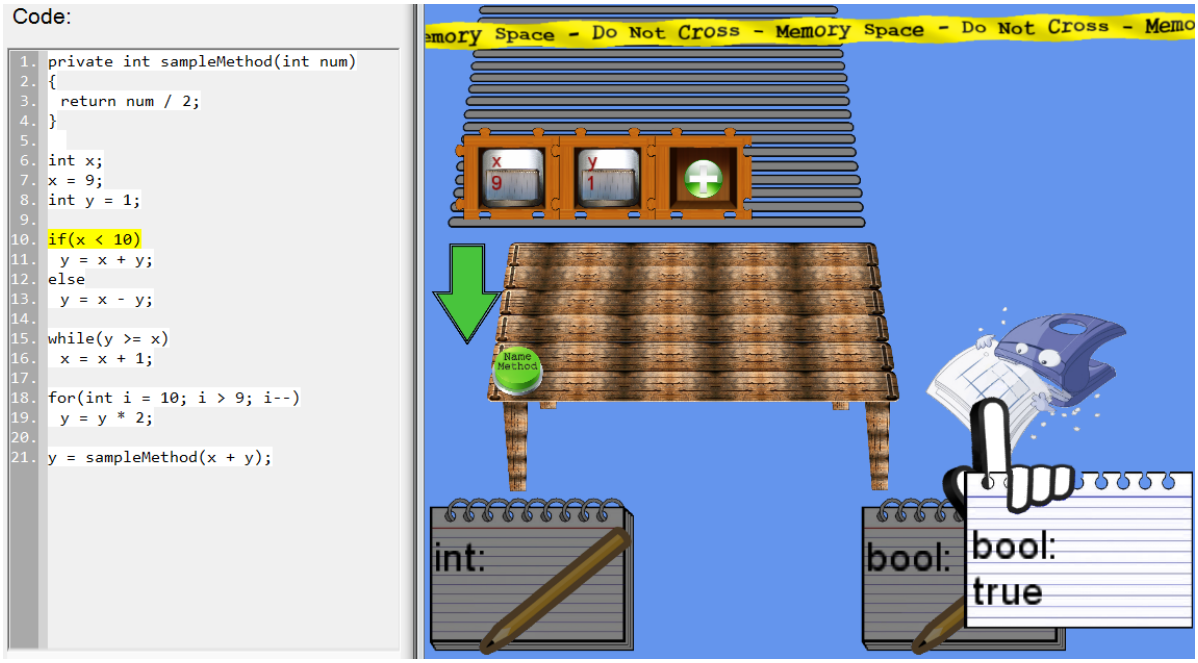


Figure A.4: By this stage the user has declared and assigned values for the variable 'y', and is about to feed the newly created Boolean value to the BoolEater in order to move into the body of the if().

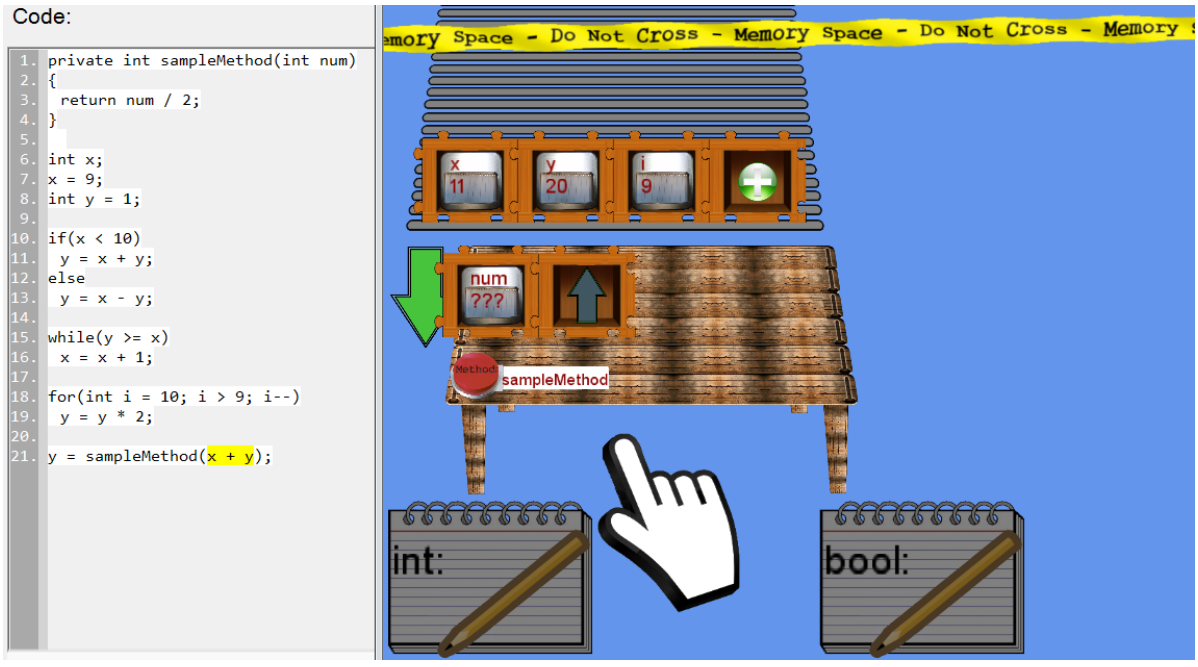


Figure A.5: The user has complete all the steps required by the various conditional structures and loops (all very similar), and has prepared an empty stack frame for calling 'sampleMethod'.

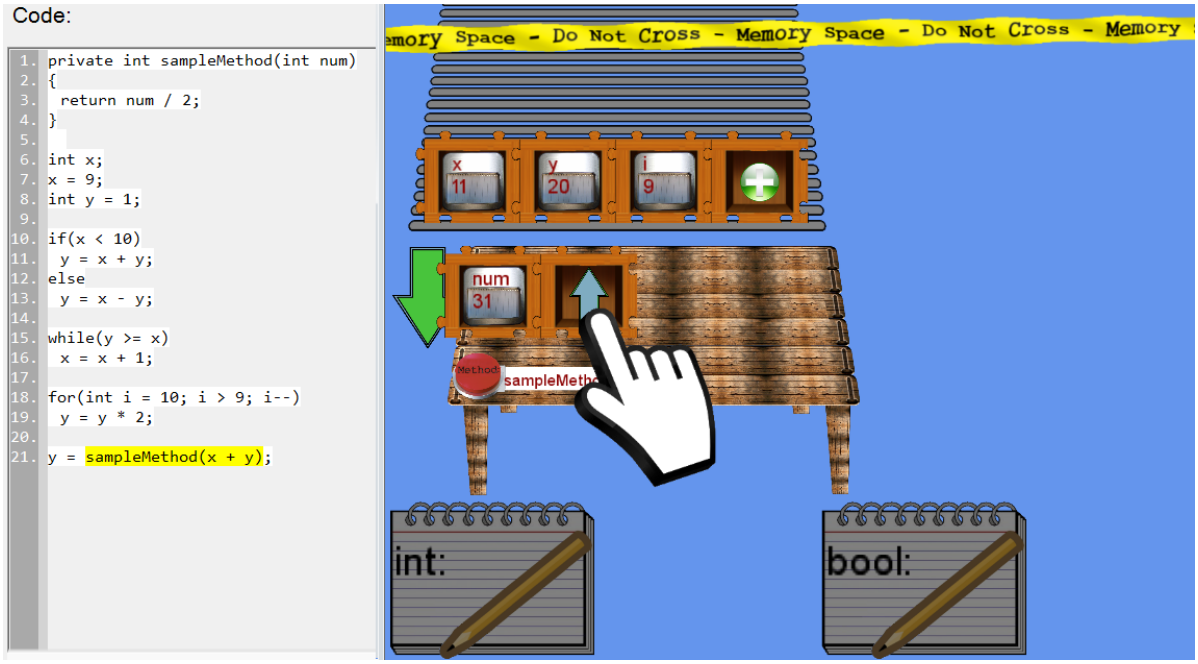


Figure A.6: The user has created and assigned a value to the 'num' parameter, and is preparing to perform the method call by clicking the call arrow.

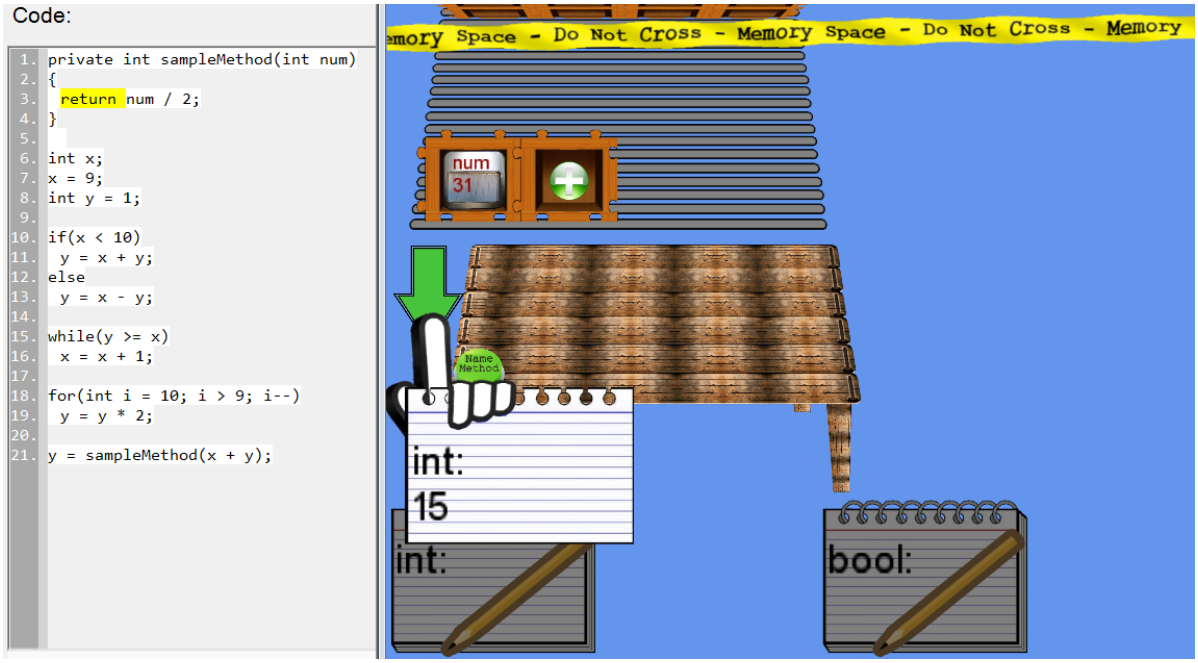


Figure A.7: The user has entered 'sampleMethod', and has created a value based on the expression 'num/2'. They are preparing to return to the previous stack frame by clicking the return arrow. Notice the cut-off stack frame near the top of the image - this is the frame of the method they will be returning to.

Appendix B

Hallway Test Levels

This appendix contains the exact code presented to users during hallway testing. Depending on the students prior experience, they would either be presented with these levels in order, or they would start with level 6 (which assumes the user has a bit more background).

B.1 Level 1 - Variables, Assignment and Operators

```
int x;
x = 23;
int y = 2;
int z = x + 4;
y = x * 2;
y = y + 4;
z = x % 4;
z = z + x - y;
z++;
```

B.2 Level 2 - Conditional Branches

```
int x = 1;
if(3 > 2)
    x = 5;
```

```
if(x == 1)

    x = x + 2;

if(x < 5)

    x = 6;

else

    x = 10;

int y = x - 2;
if(x == y || x < 6)
{

    y = -10;
    x = 10;

}
if(true && false || true)

    x = x + y;
```

B.3 Level 3 - While Loops

```
int counter = 1;
while(counter <= 6)
{

    counter = counter + 2;
    counter = counter + 3;

}
```

B.4 Level 4 - For Loops

```
int sumToN = 0;
int N = 3;
//Ordinarily these 3 are on 1 line:
for(int i = 1;
```

```
    i <= N;
        i++);
        {
            sumToN = sumToN + i;
        }
```

B.5 Level 5 - Introduction to Methods

```
private int someMethod()
{
    return 64;
}
private void anotherMethod(int x)
{
    x = x + 2;
}
private int argumentMethod(int x, int y)
{
    return x * y;
}
public void main()
{
    int x;
    x = someMethod();
    anotherMethod(x);
    x = argumentMethod(3, 4);
}
```

B.6 Level 6 - All in One

```
private int sampleMethod(int num)
{
```

```
    return num / 2;
}
int x;
x = 9;
int y = 1;
if(x < 10)
    y = x + y;
else
    y = x - y;
while(y >= x)
    x = x + 1;
for(int i = 10; i > 9; i--)
    y = y * 2;
y = sampleMethod(x + y);
```

Appendix C

Unabridged Hallway Test Results

This appendix contains all of the feedback obtained over the five separate hallway test sessions.

C.1 Test Group One

Test group one was comprised of three students who had prior programming experience. The reason these people were selected, was to ensure that misunderstanding of the code did not interfere with their experience of the interface.Their feedback follows:

- When presented with "int x = 1", all three tried to perform operations in the same order as when presented with "int x; x = 1" (which is reasonable). At the time, 'empty' variables were being avoided by putting the value straight into the variable from the hand, and not letting users see the intermediate state. This meant that the 'correct' order was: evaluate the expression, then declare and assign variable in one step.
- One user noted that they found it confusing that values take their initial default value from the hand.
- Participants did not find it obvious that the hand was acting as the go between for transferring values.
- The primary complaint was about what to do with a statement like "if(x $<$ 10)".

- Two of the three users complained that the mouse buttons should be reversed for interactions with variables (originally the left mouse button was for read, and the right mouse button was for write).
- One of the test subjects asked why there is no variable-value substitution when dealing with for complex expressions (such as "x = x + y"). This was when the game used a single pencil to declare all values, rather than a calculator.
- A single tester reported a minor inconvenience: the hint prompt, when entering a value for the hand ("<type> <value>"), was inconvenient after it had been seen once.
- No testers said so explicitly, however it seemed that none of them realized there was a division between user and machine space.
- The testers appeared to not know what the purpose of the game was at all, or their function in it.
- Once users got to the method mechanism, they were told to name the method for calling. To do this they tried entering things such as: "functionName", "functionName()", "functionName(x)", "functionName(int num);". Once they were eventually told to input the methods signature (without a semi colon), they had trouble figuring out how to add a new parameter, or what they had to do at all (some went straight for the method call button).

C.2 Test Group Two

The second group of testers were presented with an introductory screen that explained their role in the game. This group of testers all had experience with C#, this was entirely coincidental.

Subject one of group two gave this feedback:

- Wanted to know where to get values for expressions, such as "x = 1". Which meant he did not realize the function of the pencil.
- After having the pencil explained, had no trouble with "int x = 1". This was after making the operation order more consistent (per the feedback from group one).

- Had trouble figuring out what to do on the `if` statement, more specifically he wanted to know where to put the "true" once he had it in hand.
- No real trouble with the `for` loop, but did try to do the conditional first. That is an issue more to do with his understanding than the UI.
- Had no trouble with the method mechanism, either for calling or returning.

Subject two had a number of comments regarding the program, as well as some unusual behaviour:

- "I'd like the option to turn off the magnification area."
- "The conditional branches make no sense."
- "I feel very restricted, I'm given the illusion of freedom when there is in fact none."
- "Most games only show you what you need or what you've asked for." - by which he meant "hide the things I cannot use".
- He was a very experimental subject, in that he tried clicking all sorts of things erratically in order to see what did what. Because the Guide interfaces look very similar, he would get them mixed up.
- Tried to enter "int x;" once (notice the inclusion of the semicolon). This is more to do with programming habits than the UI.
- When presented with no format prompt for data entry (such as "<type> <value>"), the subject tried to put just a value (for example, 1). This meant that the alterations based on feedback from the previous group needed to be optional.
- Until it was pointed out, this subject did not notice the code interface on the left. This meant a means of drawing attention to it was needed.

Subject three had these experiences:

- The purpose of the pencil was not obvious.
- Had trouble with the `if`, just like everyone seemed to.
- As with subject 2, when presented with no format prompt for data entry (such as "<type> <value>"), the subject tried to put just a value.

- Had no trouble at all with the `for` loop.
- Seemed fine with calling methods.

C.3 Test Group Three

The third group of testers had five participants, four of which had programming experience of some kind. Interestingly, the student with no prior experience only had trouble with the first `if`, and after understanding it, he gliding through the `while` loop. The only thing that These are some the recurring concerns taken from this group (there was very little variability between individual participant's feedback):

- The relationship between the metaphors and the actual code is still not clear (the only change, regarding this point, between this group and the last was an arrow included in the introductory screen). It was decided that further improvements could be made by enlarging the code, and providing a more verbose hint system capable of highlighting sprites, and explaining how they relate to code.
- Most of these students had trouble deciding which part of a line needed to be done next (with the biggest offender being `for` loops). This was addressed by including more sensitive code highlighting.
- The context hints present during this test stage were occasionally misleading - for example, "4+5" is listed as an expression, but it is not clear that the result of the expression would be "int 9". The help was changed to read "4 + 5 -> int 9". This may highlight a problem with students understanding of type inferencing.
- Between some operations (especially assignment statements), the state of the metaphor world does not appear to change. This was partially addressed by making variables always display their contents (provided they had any). Another fix to the apparent lack of state change was to remove the values from the hand once they are used.
- "Once you get into how things work it becomes fairly easy" - this serves to promote the idea of using tutorial levels, and an optional highly-verbose introductory help system.

- Despite attempts to improve the intuitiveness of conditional behaviours, students still had trouble. It was at this stage that the decision was made to remove the Boolean value from the hand once it is used in a conditional.

C.4 Test Group Four

This test 'group' was comprised of a single student, and was done slightly differently: rather than going straight into the "all in one level", he was first asked to complete the first two lines of level one (`int x; x =1;`). This first level used the new minimalist layout, that only showed level-relevant sprites. Without **any** prompting at all he completed the first two lines in under 20 seconds. The only thing that caused this student any problems was the `if` statement, and after turning on the contextual help and highlighting the example value declaration ("bool true"), he understood fairly quickly.

C.5 Test Group Five

By this stage of testing there were not many student left who had not already tried the game, and who were willing to spend time away from their other work in order to participate. The only 'test' subject for this round was in fact a lecturer, nevertheless two valuable points where brought up:

- The first qualm was regarding how the barbed wire barrier was unnecessarily intimidating, and gave the impression that what happens in memory space is something to fear and avoid at all costs. Several alternatives were suggested, until in was decided to use the 'caution-tape' barrier with a custom message.
- The second suggestion was even more valuable than the first: having multiple pencils was fine, however the arrangement and behaviour at the time of testing was unintuitive. For example, there were no significant transitions between clicking on a pencil, writing down a value, and then actually holding the value. Secondly this subject believed that it was somewhat disconcerting that users somehow write on the piece of paper it their hand, with the pencil that isn't being picked up in any way. This might seem like an almost petty complaint, however, if one considers that metaphors being presented in an unnatural manner could easily throw off a novice,

this sort of alteration becomes fairly significant. The three stage hand sprite was included based on this.

Appendix D

Kelleher and Pausch's Survey Data

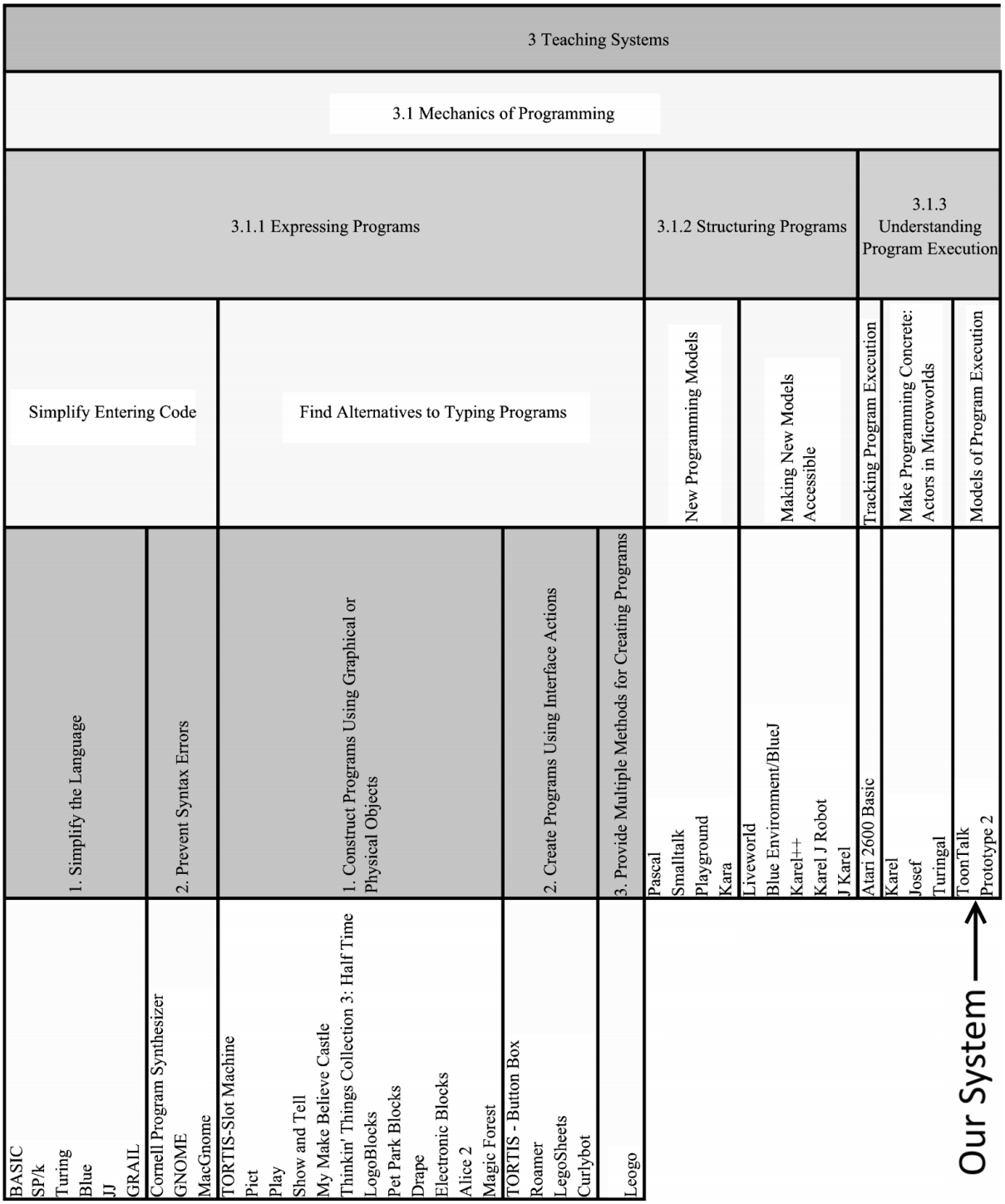


Figure D.1: The first half of Kelleher and Pausch's [4] educational software categorisation taxonomy. The most suitable category for the proposed system is shown - next to Prototype 2 and ToonTalk.

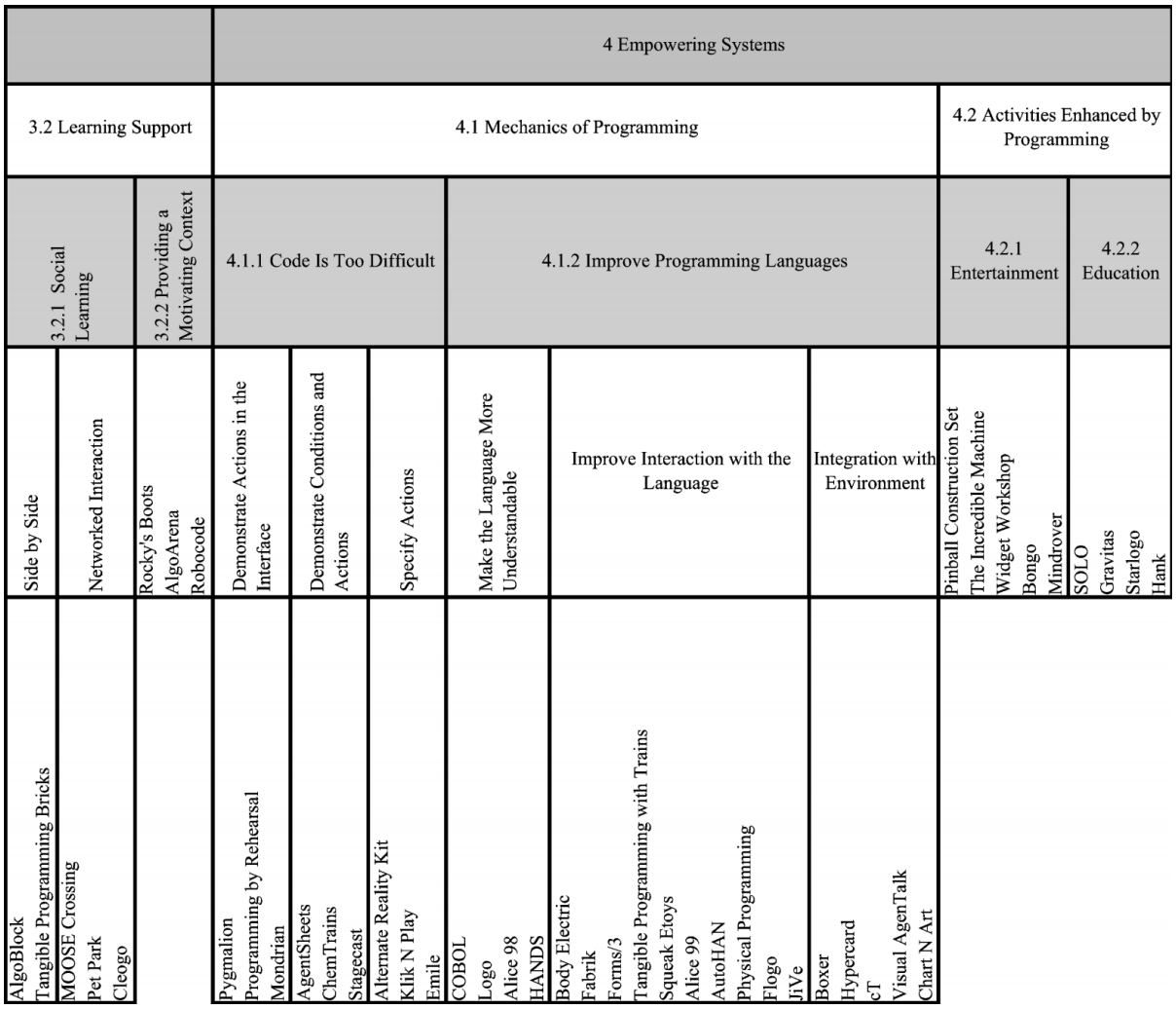
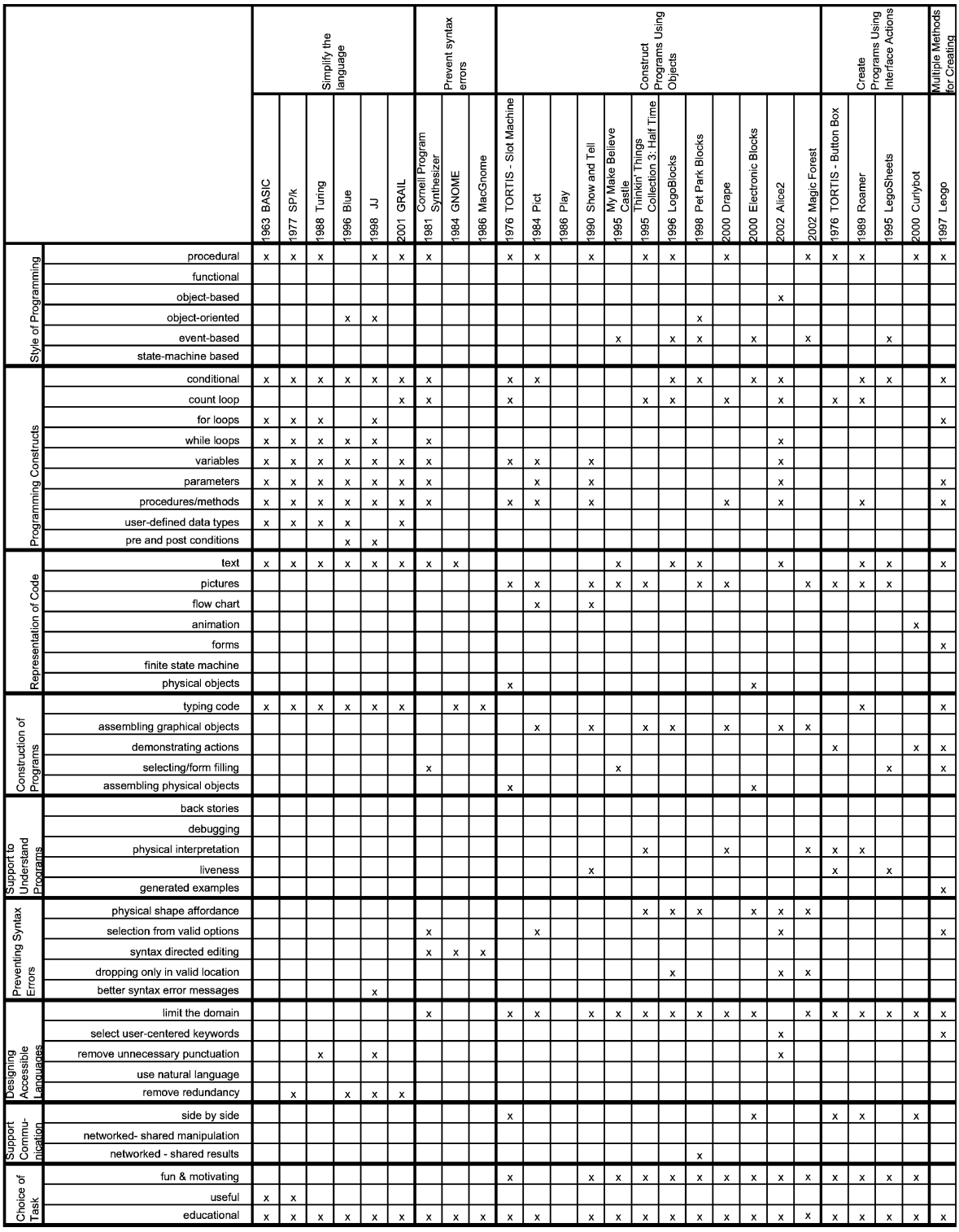


Figure D.2: The second half of Kelleher and Pausch's [4] educational software categorisation taxonomy.



		Simplify the language						Prevent syntax errors			Construct Programs Using Objects												Create Programs Using Interface Actions				Multiple Methods for Creating
		1963 BASIC	1977 SP/k	1988 Turing	1996 Blue	1998 JJ	2001 GRAIL	1981 Cornell Program Synthesizer	1984 GNOME	1986 MacGnome	1976 TORTIS - Slot Machine	1984 Pict	1986 Play	1990 Show and Tell	1995 My Make Believe Castle	1995 Thinkin' Things Collection 3: Half Time	1996 LogoBlocks	1998 Pet Park Blocks	2000 Drape	2000 Electronic Blocks	2002 Alice2	2002 Magic Forest	1976 TORTIS - Button Box	1989 Roamer	1995 LegoSheets	2000 Curlybot	1997 Leogo
Style of Programming	procedural	x	x	x		x	x	x			x	x		x		x	x		x			x	x	x		x	x
	functional																											
	object-based																				x							
	object-oriented				x	x												x										
	event-based														x		x	x		x		x			x			
	state-machine based																											
Programming Constructs	conditional	x	x	x	x	x	x	x			x	x					x	x		x	x			x	x		x
	count loop						x	x			x					x	x		x		x		x	x				
	for loops	x	x	x		x																						x
	while loops	x	x	x	x	x		x													x							
	variables	x	x	x	x	x	x	x			x	x		x							x							
	parameters	x	x	x	x	x	x	x				x		x							x							x
	procedures/methods	x	x	x	x	x	x	x			x	x		x					x		x			x				x
	user-defined data types	x	x	x	x		x																					
	pre and post conditions				x	x																						
Representation of Code	text	x	x	x	x	x	x	x	x						x		x	x			x			x	x			x
	pictures										x	x		x	x	x		x	x			x	x	x	x			
	flow chart											x		x														
	animation																										x	
	forms																											x
	finite state machine																											
	physical objects										x									x								
Construction of Programs	typing code	x	x	x	x	x	x		x	x														x				x
	assembling graphical objects											x		x		x	x		x		x	x						
	demonstrating actions																						x				x	x
	selecting/form filling							x							x										x			x
	assembling physical objects										x									x								
Support to Understand Programs	back stories																											
	debugging																											
	physical interpretation															x			x			x	x	x				
	liveness													x									x		x			
	generated examples																											x
Preventing Syntax Errors	physical shape affordance															x	x	x		x	x	x						
	selection from valid options							x				x									x							x
	syntax directed editing							x	x	x																		
	dropping only in valid location																x				x	x						
	better syntax error messages					x																						
Designing Accessible Languages	limit the domain							x			x	x		x	x	x	x	x	x	x		x	x	x	x	x	x	x
	select user-centered keywords																				x							x
	remove unnecessary punctuation			x		x															x							
	use natural language																											
	remove redundancy		x		x	x	x																					
Support Commu-nication	side by side										x									x			x	x		x	
	networked- shared manipulation																											
	networked - shared results																	x										
Choice of Task	fun & motivating										x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	
	useful	x	x																									
	educational	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure D.3: The first page of Kelleher and Pausch's [4] attribute frequency table.

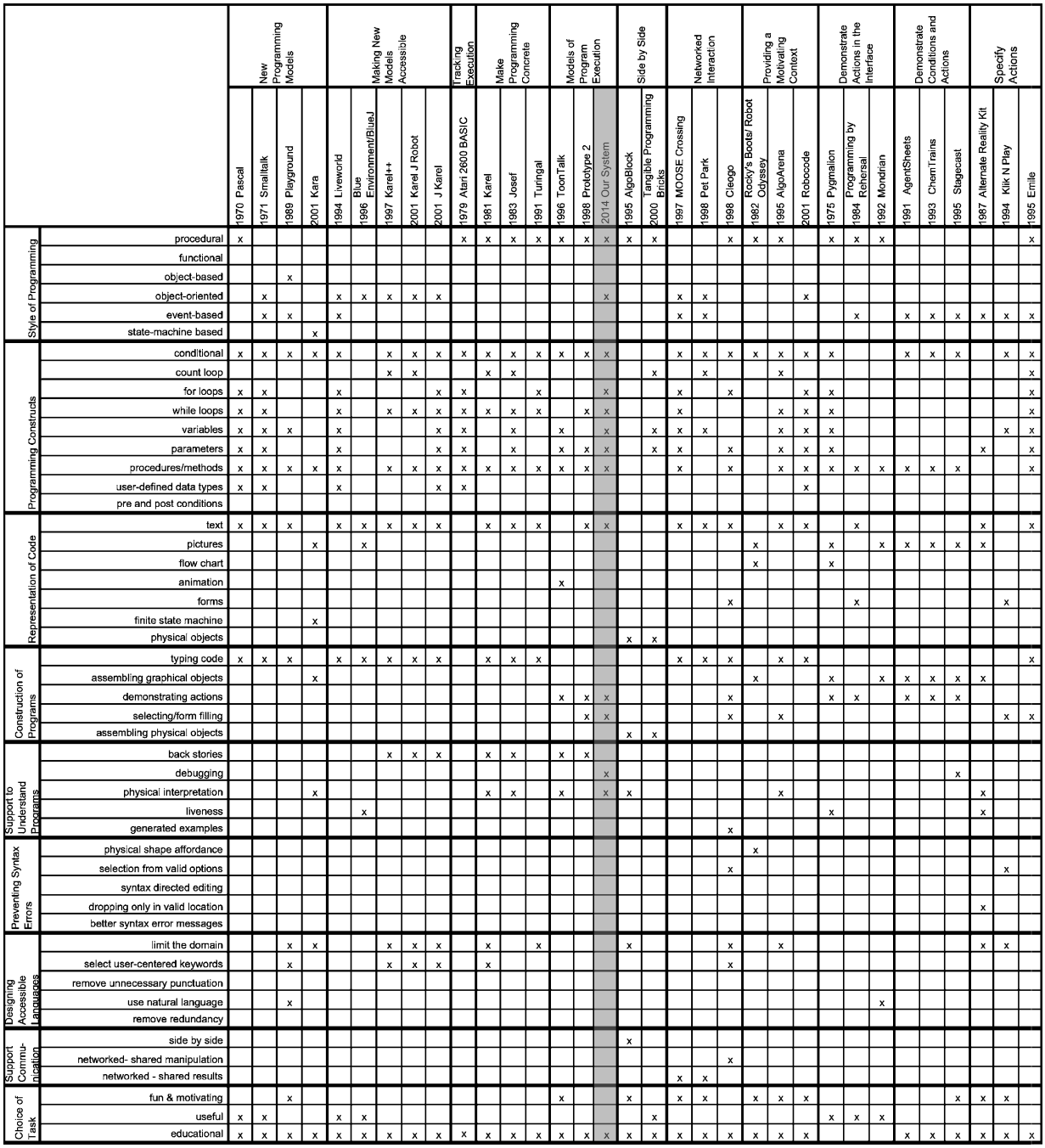


Figure D.4: The second page of Kelleher and Pausch's [4] attribute frequency table. The proposed system has been inserted here, and is highlighted.

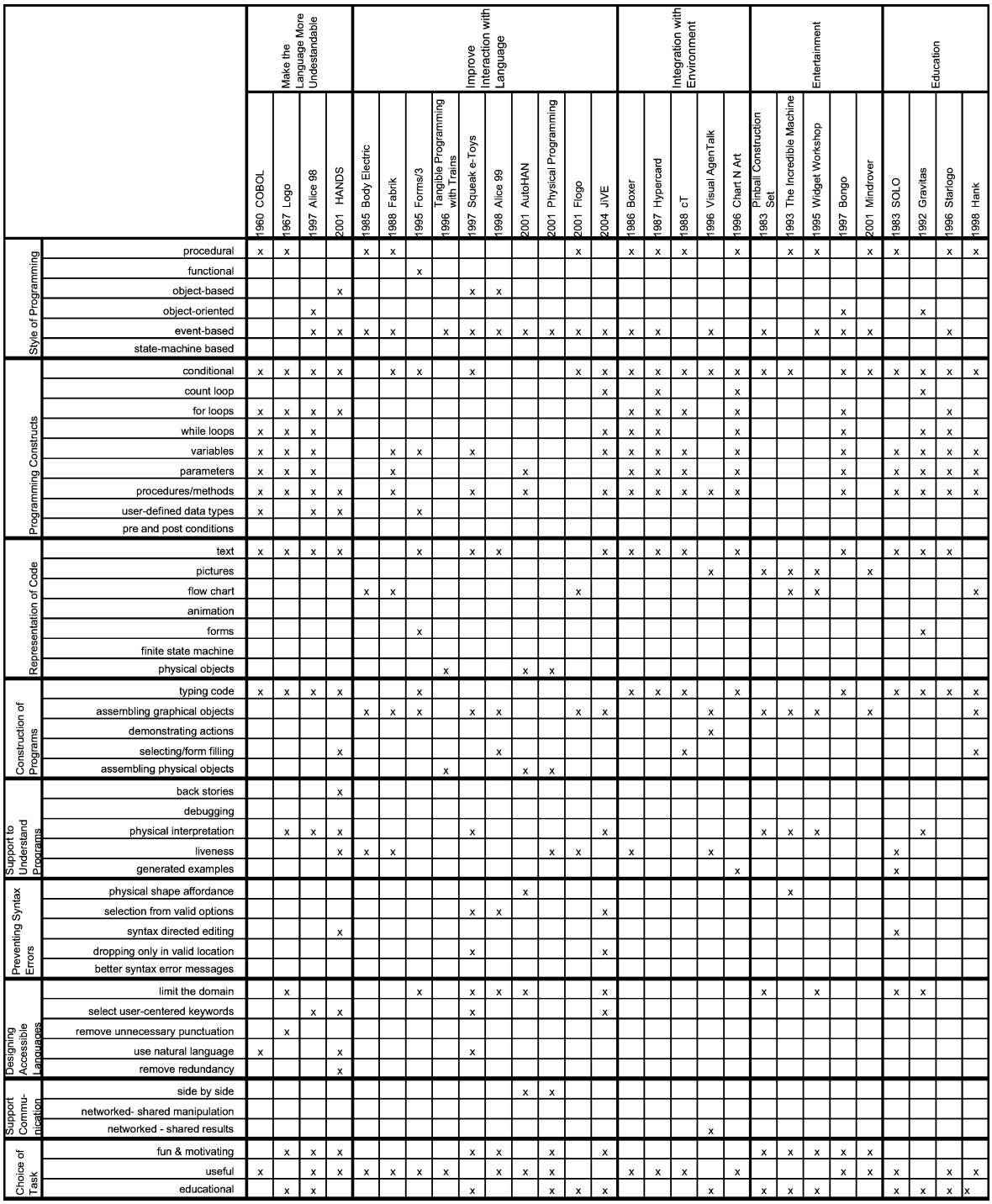


Figure D.5: The final page of Kelleher and Pausch's [4] attribute frequency table.

Appendix E

Sample Survey

This appendix contains the complete sample survey that was devised in order to potentially evaluate the quality of the metaphor set. Less formal language was used in this survey to ensure user understanding.

E.1 Metaphor-Validity Survey

A quick summary of what this survey is all about: we are developing a set of metaphors to aid novice programmers in establishing high-fidelity mental models of fundamental programming concepts - the metaphors created are intended to be usable across several different media including educational games, textbooks, and classroom environments. The metaphors presented here have been through several rounds of internal testing, followed by several rounds of user testing with students in order to identify and rectify potential problem areas.

At this stage we are of the opinion that further testing through students will no longer yield meaningful results as they cannot generally compare the metaphors being presented to them with their own mental models of programming concepts (either because they don't any or because they are inaccurate or incomplete), which is where this survey comes in: We hope to draw on the knowledge and existing mental models held by experienced programmers in order to further analyse, improve on, and validate the metaphor set we have come up with.

So as to avoid 'contaminating' test subject opinions, the questions for each concept-metaphor pair are broken down into two sections: pre-questions about how the subject

sees and imagines a concept, and post-questions which focus more on the metaphors we are proposing.

The final questions for each section are perceived scores:

- Understandability - How easy is it to grasp what the metaphor is trying to explain, and how it works?
  - 1 - I couldn't understand it at all.
  - 5 - I had to think about it but I got it in the end.
  - 10 - Perfectly intuitive and easy to grasp
- Versatility - How widely used could the metaphor be, or how many concepts do you think it could explain?
  - 1 - I don't think it relates to the concept in question ever.
  - 5 - I could use it to explain about half the scenarios I can think of.
  - 10 - It works everywhere so far as I can see.
- Durability - How likely is this metaphor to fail or break down?
  - 1 - This metaphor is so inaccurate it breaks down right out the gate.
  - 5 - It won't break down for simple example.
  - 10 - This metaphor seems as if it wouldn't fail in even the most complex situations.
- Relatability - How well does the given metaphor relate to your own mental model or understanding of a concept?
  - 1 - It doesn't relate at all.
  - 5 - It sort of relates but not across the board.
  - 10 - It's almost like you read my mind.
- Accuracy - How well does the given metaphor compare to the way a particular concept, structure, or function actually works?
  - 1 - The two aren't remotely similar.
  - 5 - The metaphor is a reasonable representation some of the time.
  - 10 - The metaphor is a perfect representation of what happens behind the scenes.

Two Quick Pre-Pre-Questions:

Pre-Pre Question 1:

Approximately how many years of programming experience do you have?

Pre-Pre Question 2.

Please list your favourite programming languages from most to least favourite.

You should note that the concepts in question have been arranged from most fundamental (and thus most important) to most complex (and thus least important). This means that if you choose to submit a half finished survey (which is fine) the questions you already answered were the more important ones anyway. HOWEVER If you want to stop halfway you need to skip to the end of the survey and hit submit...Google Forms isn't smart enough to save partial data.

Consent:

All information gathered is anonymous. I consent to the anonymous use of the information I provide in this survey. Y/N

Values in a Programming Language

Pre-Questions

Values Pre-Question 1:

If you were to explain values to a student as a real-world metaphor, how would you do it?

Values Pre-Question 2:

Do you have a mental model for values, if so could you try to explain it?

Values Pre-Question 3:

Can you think of any scenario where your mental model or proposed metaphor for values becomes inaccurate?

Our Metaphor

The most fundamental things that code uses are individual values, either as actual values or the value of a reference address (this would include integers, doubles, Booleans, chars, and all other primitives). As this is such a key concept, its associated metaphor needs to be one of the most reliable and easy to understand: it was proposed that a simple piece of paper with the value written on it would be suitable. Everyone can relate to pen and paper, you cannot erase pen from paper (meaning values are never 're-used'), and if one were to imagine solving an expression in their head the most sensible thing to do with the answer would be to write it down. Figure 1 shows an example of a value notepad, the user inputting a value, and finally the user holding the value.

Post-Questions

Values Post-Question 1:

How does the metaphor we give for values compare with how you imagine it working?

Values Post-Question 2:

Can you find fault with our metaphor for values, if so what would it be?

Values Post-Question 3:

If you could alter one thing about this metaphor what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Variables in a Programming Language

Pre-Questions

Variables Pre-Question 1:

If you were to explain variables to a student as a real-world metaphor, how would you do it?

Variables Pre-Question 2:

Do you have a mental model for variables, if so could you try to explain it?

Variables Pre-Question 3:

Can you think of any scenario where your mental model or proposed metaphor for variables becomes inaccurate?

Our Metaphor

Variables are represented by boxes, with transparent lids, that contain a single piece of paper (in the same way that a simple variable can contain a single value). Reading of a variable would be done courtesy of the transparent lid; you would simply look into the variable box and copy the value off the paper without changing the content of the box. Assigning to a variable involves opening the box, disposing of the old value-paper, and then placing the new value (which you would be holding) into the box. Figure 2 shows one potential version of the variable box metaphor. The representation of variables on the stack comes later - up to this point it is enough for users to know that variables exist and how to imagine them.

Post-Questions

Variables Post-Question 1:

How does the metaphor we give for variables compare with how you imagine it working?

Variables Post-Question 2:

Can you find fault with our metaphor for variables, if so what would it be?

Variables Post-Question 3:

Can you find fault with our metaphor for variables, if so what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Expressions, Arithmetic and Calculation

Pre-Questions

Expressions Pre-Question 1:

If you were to explain expressions and calculation to a student as a real-world metaphor, how would you do it?

Expressions Pre-Question 2:

Do you have a mental model for expressions and calculation, if so could you try to explain it?

Expressions Pre-Question 3:

Can you think of any scenario where your mental model or proposed metaphor for expressions and calculation becomes inaccurate?

Our Metaphor

Expressions of all kinds have two viable metaphors in our unified set: they can either go through an in-game calculator, or the user can work them out in their head. Good arguments can be made for either case, and thus both metaphors have been included so that anyone who wants to expand on this work can make up their own mind. The original technique for expressions and evaluation was the use of an in-game calculator, which worked much like a real-world scientific calculator, where users could enter an expression and then go back and alter terms once they had the required values. For example, a user might enter "x + 5" into their calculator, they would then look at the local variables and copy the value from x onto a piece of value paper, that value would be fed into the calculator (almost like a fax machine, except the paper is destroyed).

When the user asks to substitute into the place holder 'x' in the expression, the calculator would replace the variable in the expression and then wait for further instructions from the user (either for more instructions or for the user to ask it to evaluate the answer). When the expression no longer has variables that need values, the user would hit 'evaluate' and the answer would be printed out from the calculator onto a piece of paper. Figure 3 illustrates the sequence of events when using the calculator metaphor.

An alternative technique for expression evaluation actually takes away extra metaphors entirely by just asking the user to evaluate the expression in their heads (or on paper, or with their real-world calculator), in much the same way as when users debug a piece of code. Steps A, B and C in Figure 1 demonstrate how users would enter the answer to an expression.

Post-Questions

Expressions Post-Question 1:

How does the metaphor we give for expressions and calculation compare with how you imagine it working?

Expressions Post-Question 2:

Can you find fault with our metaphor for expressions and calculation, if so what would it be?

Expressions Post-Question 3: Can you find fault with our metaphor for expressions and calculation, if so what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Local Variables and the Current Stack Frame

Pre-Questions

Local Scope Pre-Question 1:

If you were to explain the current stack frame (perhaps think local scope) to a student as a real-world metaphor, how would you do it?

Local Scope Pre-Question 2:

Do you have a mental model for the current stack frame, if so could you try to explain it?

Local Scope Pre-Question 3:

Can you think of any scenario where your mental model or proposed metaphor for the current stack frame becomes inaccurate?

Our Metaphor

The next step up the abstraction ladder is to delimit scope and accessibility using some kind of uncrossable barrier (for example, a line of barbed wire, police tape, or even a simple fence - Figure 4 shows two possible barriers). This barrier serves to separate the current frame (i.e. the local scope) from the other stack frames (as well as global variables from heap variables). The non-local area, and everything in it, will be discussed

later; scope-delimitation was mentioned here for clarification of the next concept to be described: how to represent the current frame and all the variables that are stored within it, while still being relatable, and without complicating the pushing of stack frames onto the stack.

We represent the current frame as something akin to a jigsaw or Lego bookshelf: users would start out without a bookshelf at all when there are no local variables, a mechanism which dispenses bookshelf 'pieces' is all that would be present. Whenever the user needs to declare a new variable they would first have to extend the bookshelf so there would be enough space for it. Each space on the bookshelf would be able to contain one variable box. The expandable bookshelf metaphor allows you to teach students about how local variables can go out of scope, for example, when they were declared inside a loop, the out-of-scope variables (and their associated bookshelf sections) can be removed entirely. The metaphorical bookshelf sits on top of a conveyor belt, this only becomes important when the user is able to call methods, thus it is discussed in more detail in the next section. Figure 5 shows what the user might be presented with, depending on the current stack frame state.

Post-Questions

Local Scope Post-Question 1:

How does the metaphor we give for the current stack frame and local scope compare with how you imagine it working?

Local Scope Post-Question 2:

Can you find fault with our metaphor for the current stack frame and local scope, if so what would it be?

Local Scope Post-Question 3:

Can you find fault with our metaphor for the current stack frame and local scope, if so what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

The Stack, Without Method Functionality

Pre-Questions

Stack Pre-Question 1:

If you were to explain the whole stack (rather than just one frame) to a student as a real-world metaphor, how would you do it?

Stack Pre-Question 2:

Do you have a mental model for the stack, if so could you try to explain it?

Stack Pre-Question 3:

Can you think of any scenario where your mental model or proposed metaphor for the stack becomes inaccurate?

Our Metaphor

Once one understands how to interpret the representation of the current frame, one is then also able to interpret the stack as a whole: just like a library usually has more than one bookshelf, a stack usually has more than one frame - therefore we can cross the two ideas and represent the stack as rows of bookshelves. This is also where the accessibility divide becomes important, as the stack is primarily located of the directly accessible space. When a new frame is created from a method call, all non-local stack frame bookshelves are moved across the memory divisor so that they exist in memory space. Figure 6 shows how the stack would expand into memory space after each method call, and shrink after each return statement. For clarity the memory space includes the heap, the non-local stack frames, and nothing else.

Post-Questions

Stack Post-Question 1:

How does the metaphor we give for the stack compare with how you imagine it working?

Stack Post-Question 2:

Can you find fault with our metaphor for the stack, if so what would it be?

Stack Post-Question 3:

Can you find fault with our metaphor for the stack, if so what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Methods and Method Functionality

Pre-Questions

Methods Pre-Question 1:

If you were to explain methods and their associated functionality to a student as one or more real-world metaphors, how would you do it?

Methods Pre-Question 2:

Do you have a mental model (or more than one) for all things method-related, if so could you try to explain it (or them)?

Methods Pre-Question 3:

Can you think of any scenario where your mental model/s or proposed metaphor/s for methods becomes inaccurate?

Our Metaphor

Now that we know what the stack looks like we can examine the metaphors governing everything to do with methods.

As mentioned previously the current frame's associated bookshelf is positioned on top of a conveyor belt, this allows bookshelves to be moved across the barbed wire and into or out of the fix all. This mechanism is important for method calls and returns: the conveyor belt has two control buttons, the call button, and the return button. These two buttons move everything on top of the conveyor belt either toward or away from the directly accessible space. So far it is not hard to imagine the moving of the conveyor belt as matching up with the actual process of pushing and popping frames to and from the stack; the tricky part of designing this section of the metaphor set is considering what happens to frames that have been popped off the stack, and where new frames come from. Both situations can be explained in a similar way to overwritten variable values: For popped frames and overwritten values the object in question no longer belongs anywhere and so must be destroyed, the paper value would be torn up or burnt, and a similar thing would be done to the popped off frame. In the same way that we prepare paper values using notepads we decided to use a workbench to represent the creation of a soon to be used frame.

After getting the student to name the method they are preparing to call, you have several options for method preparation that you can present them with. The two proposed techniques are as follows: the first way is to ask users to name and assign parameters and arguments in much the same way as local variables (one at a time, with types, name, and values all the explicit responsibility of the user); the second possible presentation method is to give users a selection of available method signatures which they need to pick from, and then automatically provide all the parameters ready to receive values, which the user simply needs to fill. Figure 7 compares the two main method preparation techniques side-by-side.

Much like the two possible ways of asking students to evaluate expressions, these two method mechanisms are both valid and the technique used will depend on the instructor or implementation. For a void method no further explanation is really required for how the user leaves the method (they simply press the conveyor belt return button). However, it might need saying that a value returning method works by simply making the user hold the return value in their hand before pressing the return button, that way the return value is in hand when they get back to the previous frame, and thus they can use the value straight away.

Post-Questions

Methods Post-Question 1:

How do the metaphors we give for methods compare with how you imagine them working?

Methods Post-Question 2:

Can you find fault with our metaphors for methods, if so what would it be?

Methods Post-Question 3:

If you could alter one thing about this metaphor what would it be?

Methods Post-Question 4:

Of the two possible representations, which do you prefer and why?

Methods Post-Question 5: Initial usability tests have shown that students pick up the signature-picking metaphor far faster than all the other metaphors explained so far, do you have any opinions on why that might be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Global Variables

Pre-Questions

Globals Pre-Question 1:

If you were to explain global variables to a student as one or more real-world metaphors, how would you do it?

Globals Pre-Question 2:

Do you have a mental model (or more than one) for global variables, if so could you try to explain it (or them)?

Globals Pre-Question 3:

Can you think of any scenario where your mental model/s or proposed metaphor/s global variables becomes inaccurate?

Our Metaphor

More modern programming languages don't strictly speaking have truly global variables anymore, instead we have class fields. So in this context global variables refer to class fields of whatever context we are currently in. This metaphor is more to do with their representation than with access to then (which is covered later)

Firstly, because global variables are always accessible, they need a representation inside the user accessible space. Secondly, because they exist almost from the very start of the program the user needn't be able to create more of them (unlike local variables). These two facts can be brought together with another bookshelf metaphor: a fixed size bookshelf that has all the necessary variables already declared. Aesthetically one can represent this sort of bookshelf in more than one way, our proposed representations is shown in Figure 8. If the number of globals variables does not fit nicely on a regular (rectangular) bookshelf, one could instead use the jigsaw representation used for the local stack frame. A potential concern for some might be that globals should be created explicitly much like locals, this point is debatable.

Post-Questions

Globals Post-Question 1:

How do the metaphors we give for global variables compare with how you imagine them working?

Globals Post-Question 2:

Can you find fault with our metaphors for global variables, if so what would it be?

Globals Post-Question 3:

If you could alter one thing about this metaphor what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Representing the Heap, Objects and Instances (but not communicating with either just yet)

Pre-Questions

Heap Pre-Question 1:

If you were to explain the heap and objects (or reference types) to a student as one or more real-world metaphors, how would you do it?

Heap Pre-Question 2:

Do you have a mental model (or more than one) for the heap and objects, if so could you try to explain it (or them)?

Heap Pre-Question 3:

Can you think of any scenario where your mental model/s or proposed metaphor/s for the heap and objects becomes inaccurate?

Our Metaphor

Reference types in the local scope can easily be represented by a simple memory address written down as a value. The more challenging thing to represent about reference types is what gets stored in memory (outside of immediately addressable space). Without any additional metaphors there is no way for players to affect what exists outside of the

directly accessible space, to solve this issue we introduce a robot to represent the memory manager, which is elaborated on later. For now it is sufficient to say that the memory manager robot receives messages from the user, and carries them out in memory space. With our memory manager robot ready to interact with the heap and non-local stack on our behalf, we now need a way to represent the heap and access to the non-current frames on the stack. Access to the stack is fairly simple as we already have a concrete metaphor for the stack itself (the rows of bookshelves): the memory manager robot simply takes the address he has been given and goes between the bookshelves to interact with the appropriate variable box.

The heap requires more thought. After considering that the size of the heap is in fact finite (determined by the available memory) in an actual computer, we decided that the heap metaphor could also be represented by a finite, fixed-size structure. The easiest way to represent it without deviating from our existing metaphors is to have one super-sized bookshelf, possibly with movable dividers, where unused space is represented by the lack of a variable box. Figure 9 shows an example of this extra large bookshelf, and also illustrates that in certain media the bookshelf representation falls slightly short due to the size limitations. For this reason, in-game, the heap can instead be shown as a compact series of squares (as shown in Figure 10), this alternative representation can just be thought of as a compact version of the bookshelf.

Representing objects, that are more complex than arrays of values, is where reference types become particularly difficult to represent: A simple non-array object such as a Random generator could be represented on the heap as two numbers and something to point to the objects associated method code. For this example the numbers would be the start seed and the current seed; however the methods belonging to the object would have to be seen only in the code that the students are following. Figure 11 shows a potential representation of the heap with instances of Random and Account classes, unlike Figure 9 the objects on this heap have their properties labelled so as to make it clear to users that they are not just arrays of numbers. The 'name' property of the Account class would be a string, its contents on the heap would depend on how one chooses to represent strings, as explained in the final section of the survey.

Post-Questions

Heap Post-Question 1:

How do the metaphors we give for the heap and objects compare with how you imagine them working?

Heap Post-Question 2:

Can you find fault with our metaphors for the heap and objects, if so what would it be?

Heap Post-Question 3:

If you could alter one thing about this metaphor what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Interacting with Objects and other Reference Types

Pre-Questions

Reference Communication Pre-Question 1:

If you were to explain object referencing and communication with the heap to a student as one or more real-world metaphors, how would you do it?

Reference Communication Pre-Question 2:

Do you have a mental model (or more than one) for object referencing and communication with the heap, if so could you try and explain it (or them)?

Reference Communication Pre-Question 3:

Can you think of any scenario where your mental model/s or proposed metaphor/s for object referencing and communication with the heap becomes inaccurate?

Our Metaphor

Once you understand how the heap and various reference types are represented, communication with them becomes fairly simple to understand as well. The memory manager robot is responsible for all interactions in the memory space, he receives his instruction via the terminal (which is located in directly accessible space). When writing to the heap, values go from user to terminal to robot to memory address (and the other way for reading from the heap). In order to access data on the heap the user needs to provide the memory manager robot with two things: the objects base address, and the offset of the specific object-element of interest. For non-array objects the 'offset' would instead be the name of the field or property in question. Reference access to variables on the stack is done in much the same way, except there is no need to provide an offset, an address alone is sufficient. In order to facilitate easier state recognition the memory manager robot can have several ways of showing whether it currently has an address, offset or value: either lights on its torso can show up saying what it has (as shown in Figure 12), or it could hold visible values. These two metaphors do not require much explaining as they are just middlemen for communication with the metaphors of importance.

Post-Questions

Reference Communication Post-Question 1:

How does the metaphor we give for object referencing and communication with the heap compare with how you imagine it working?

Reference Communication Post-Question 2:

Can you find fault with our metaphor for object referencing and communication with the heap, if so what would it be?

Reference Communication Post-Question 3:

If you could alter one thing about this metaphor what would it be?

Understandability Score - Please score our proposed metaphor's understandability from 1 to 10

Versatility Score - Please score our proposed metaphor's versatility from 1 to 10

Durability Score - Please score our proposed metaphor's durability from 1 to 10

Relatability Score - Please score our proposed metaphor's relatability from 1 to 10

Accuracy Score - Please score our proposed metaphor's accuracy from 1 to 10

Strings

Pre-Question

Strings Pre-Question:

If you could represent strings as either objects or value types (bearing in mind the trade-offs between usability and accuracy), which would you choose and why.

Our Metaphor

Strings deserve a special mention regarding their representation; this is because while they are strictly speaking objects, their immutability means that treating them as value types is unlikely to cause problems. For this reason it was proposed that strings could have two representations in our metaphor set, depending on the preference of the user or teacher: one can either use the more accurate (but more bulky) option of treating them as objects on the heap, or they can be treated as value types that reside on the stack.

If one chooses to represent strings as objects on the heap then they would be treated as arrays of characters, with the special property of being read only after their initial declaration (due to their immutability). When they are treated as value types the metaphors do not break down: when you assign one to another you can treat them as though they are copies rather than aliases. As briefly mentioned when explaining the heap and objects, objects on the heap that have string properties would have different representations based on the choice of string representation: if we use the more accurate model with strings as objects then string properties would contain memory addresses, and the strings themselves would exist elsewhere on the heap as seemingly separate objects. If one uses the strings-as-value representation then string properties become simple, with the strings clearly being inside the parent object.

Post-Question

Strings Post-Question

Given our explanation of strings as either objects or value types, would you change your previous answer, why/why not?

Appendix F

Overlarge Sample Code

This appendix contains the code for a single method: `buildUpValueInputString`. This method is approximately 27 lines long, and still does not cater for all possible user input types (such as special characters or alt+ Unicode input). It serves to illustrate how much easier it is to use pre-existing frameworks for textual input, rather than recreating them in XNA.

```
private void buildUpValueInputString()
{
    //Backspace functionality
    if (builtUpUserInput.Length > 0 && isKeyPressed(Keys.Back))
    {
        //remove the last character:
        builtUpUserInput = builtUpUserInput.Remove(builtUpUserInput.Length - 1);
        return;
    }
    //This whole if can be compressed into a single line with in-line
    //conditionals and lambdas...it just becomes hard to read
    //Character case checking via Shift-key dependant delegates:
    Func<string, string> caseConverter;
    if (keyboardNewState.IsKeyDown(Keys.LeftShift) ||
        keyboardNewState.IsKeyDown(Keys.RightShift))
    {
        caseConverter = delegate(string keyAsString)
            { return keyAsString.ToUpper(); };
        //Can also be done with labdas, eg:
        //(string x) => { return x.ToUpper(); };
```

```
        }
        else
        {
            caseConverter = delegate(string keyAsString)
                { return keyAsString.ToLower(); };
        }
        //Loop handles multiple simultaneous key presses:
        var pressedKeys = keyboardNewState.GetPressedKeys();
        foreach (Keys key in pressedKeys){

            if (!isKeyPressed(key)) //if the key isn't pressed then do not use it
                continue;
            string asString = key.ToString();
            //is it a normal letter?
            if (asString.Length == 1){
                builtUpUserInput += caseConverter(asString);
                continue;
            }
            //is it a number off the normal number keys?
            if (asString.Length == 2 && asString[0] == 'D'){
                //no need to change the case if its a number
                builtUpUserInput += asString[1].ToString();
                continue;
            }
            //is it a number off the numpad?
            if (asString.Contains("NumPad")){
                 //no need to change the case if its a number
                builtUpUserInput += asString[6];
                continue;
            }

            //Any extra characters desired can be included in this dictionary.
            Dictionary<string, string> remainingConversions =
                new Dictionary<string, string>()
                { { "Space", " " }, { "OemPeriod", "." }, { "Decimal", "." } };
            if(remainingConversions.ContainsKey(asString))

                builtUpUserInput += remainingConversions[asString];
        }

    }
```