Collaborative and Multiple-Notation Programming Environments for Children

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy
in the
University of Canterbury
by
Timothy Nicol Wright

Examining Committee

Associate Professor Andy Cockburn	Supervisor
Professor John Grundy	University of Auckland
Professor Claire O'Malley	University of Nottingham

University of Canterbury
2004

For all my whānau, especially my parents.

Table of Contents

List of Figures

List of Tables

Abstract

Users who can program computers can use their computer more effectively than those who can not. While much research has examined how to help users easily program computers, two methods that show promise have not yet been throughly investigated in the context of children's programming environments. The first method provides users with multiple representations of a computer program. Providing multiple representations lets users choose a representation close to their own mental model of computer programming and helps them transfer their knowledge from one domain to another. The second method provides support for collaboration. When users collaborate they build a shared understanding of a computer program and they can learn from teachers and more competent peers. This thesis presents an investigation into these two methods: collaborative and multiple-notation programming environments for children. Our target users are children aged eight to twelve years.

The contributions described in this thesis are: providing an analysis of how programming environments use different representations of computer programs; describing two evaluations (one of collaboration and one of multiple notations), and introducing our programming environment, Mulspren. The analysis describes three cognitive gulfs that can create problems for users when programming and identifies eight factors related to notation use that risk creating these gulfs. The first evaluation finds that children can read and understand a conventional-style notation faster than an English-like notation, but that they prefer the English-like notation. The second evaluation finds that the type of collaboration support present in a collaborative environment does not reliably affect how well children learn to solve a puzzle. The analysis and two evaluations influenced the design of our programming environment: Mulspren. Mulspren users can interact with two different notations at the same time, and can move between the notations seamlessly. We call this programming style '*dual notation programming*.'

Acknowledgments

First and foremost, my supervisor, Andy Cockburn. His relentless pursuit of good research and perfect writing has kept me on track (although, at times, I wasn't sure which track I was on). His pursuit of waves and rocks reminded me that there is a life outside university.

I thank those at *RDU*[1], *Solo Bravo*[2], and the *Bealey Massive*[3] for giving me that life outside university. Michael JasonSmith[4], who was doing a PhD at the same time and place as me, is a great friend and provided a great sounding board for thesis frustrations[5].

We are the grateful recipients of a Marsden grant, and thank Ilam, Merrin, Westburn, Brooklyn, Northland, and Wadestown Primary schools for access to their students. Jane Mackenzie and Fiona Wright[6] provided teacher-guidance about primary school children. James Noble encouraged me to be lazy, and Peter Andreae gave me the idea of dedicating the first half of every day to thesis work. Without this idea I would still be writing my thesis now.

[1] RDU: A radio station.
[2] My band.
[3] You know who you are.
[4] JasonSmith: capital S, one word, no hyphen.
[5] I hope I've done the same for him.
[6] Fiona Wright was Fiona Smith for most of this thesis.

Chapter 1

Introduction

Programming computers is a remarkably useful skill. Computer programmers can instruct their computer to perform tasks including: adding new functions to existing applications, designing and building custom applications, and contributing to community-based open-source projects. While these tasks are far beyond the everyday tasks a non-programming user performs, he or she may benefit from programming experience while performing activities ranging from automating repetitive tasks to modifying programs that the non-programmer receives from a variety of sources [109]. These sources could include: web pages (JavaScript is often embedded in webpages); word processors (generating macros using programming by demonstration mechanisms); and friends (through email).

Despite the usefulness of computer programming, only a small proportion of end-users have programming skills. To help end-users gain these programming skills, many researchers have created many varied programming environments which use techniques ranging from programming by demonstration environments, where the computer attempts to infer a computer program based on user actions (eg. [36, 54]), to programming environments where users must specify exactly what the program is going to do at all times (eg. [109, 134]); from programming environments where users manipulate textual symbols (eg. [31, 137]) to programming environments where users manipulate physical objects (eg. [177, 201]). Although this plethora of research in-

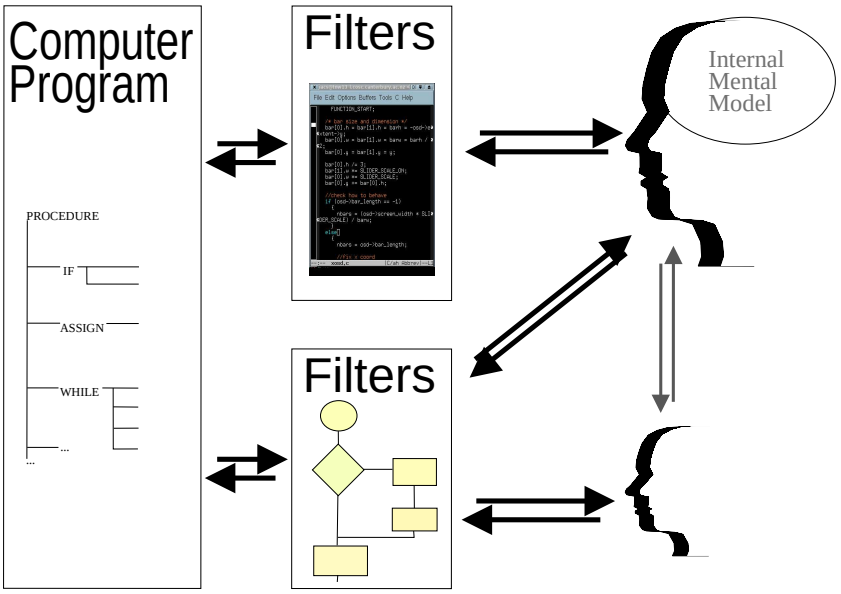


Figure 1.1: Collaborative and Multi-Paradigm Programming: Two users are interacting with each other as well as multiple representations of a computer program. While they do this they create a shared understanding of the computer program.

cludes many user-friendly programming environments, the primary programming environments that many users have access to are those that come with their office suite: typically a spreadsheet and a word processor (with a macro language). In these environments users program using textual symbols and a syntax akin to a professional programmers' programming language.

We have identified two areas that have potential to increase understanding of conventional syntax. The first area is multiple-notation programming. Multiple-notation programming environments show multiple representations of a computer program to a user. These representations range from conventional ideas of programming notations through physical representations of programming construct to transient spoken notations. While little research has examined the effects of multiple notations on users, they show considerable promise helping users lever knowledge of one domain to learn about a different domain. Additionally, users can also select a notation that best suits their mental model of how their program works. For our target users (children, 8–12 years old) we hypothesise that by showing computer programs both with an English representation and a conventional-style representation, we can help the users lever their knowledge of English and learn the conventional-style representation. Part I of this thesis investigates this hypothesis and describes an empirical evaluation of knowledge transfer in a multiple representation

context.

The second area is collaborative programming. Collaborative programming environments are those that let multiple users interact with the same computer program. For a collaborative environment, the physical location of the users doesn't matter: they can be in the same room or even in different countries. As well as the location of the users, the type of collaboration can occur in a variety of ways. The types of collaboration support range from simply providing a virtual space for users to talk about computer programs to letting users in different locations modify and run the same computer program while being aware of what the other user is doing to the program and seeing their changes immediately. Many researchers believe that collaboration aids learning because it creates shared understanding [100, 172] and allows interactions with teachers or more-competent peers [186]. This importance of collaboration for programming is reinforced by examining professional programmers: they often work in teams, and some software development methodologies provide explicit support to gain from collaboration. Examples of these methodologies include extreme programming [10] and open source software development [147]. Unfortunately, few collaborative programming environments have been built for children. Part II of this thesis investigates collaboration.

We have identified both learning and technical reasons to combine the two approaches (see Figure 1.1). The learning reason is that when multiple users are presented with multiple representations they can use the representations to aid communication between the users and to increase shared understanding. For example, the representations could act as a translator between one child who is proficient at understanding conventional code and another who is proficient in only English. The technical reason is that both approaches require similar program design: a shared model and multiple representations of the model. Collaborative programming environments need a shared model of a program and need to display this model to multiple users. To avoid breakdown of social protocols, changes by one user need to be shown to all other users immediately. Multiple-notation programming environments need an abstract representation of the program and should provide different representations of the program to a user. To avoid inconsistency, changes in one representation need to be immediately reflected in the other representations. Both these designs have issues of locking, overlapping notations, and consistency management.

The remainder of this chapter outlines our contributions, defines the scope of this thesis, and outlines the thesis structure.

1.1 Contributions

The contributions of this thesis are:

1. We introduce a method to concisely describe how different notations are used in programming environments. We call the description a *Language Signature*, and use this method to classify, review, and assess end-user programming environments. This method can be used to predict some usability problems with particular programming environments. We published a paper relating to this contribution [197].

2. We describe how several cognitive gulfs exist in programming environments, and these gulfs can hinder user's programming experience. This contribution has implications for programming environment developers: the developers should make all efforts to avoid gulfs that can hinder users' programming experience. We also describe a set of heuristics related to the three programming gulfs. These heuristics can be used to analyse the usability of notations in a programming environment. We published two papers related to this contribution [193, 197].

3. Through an empirical study we show that children can understand computer programs represented with multiple redundant representations and that children understand code written in a conventional-style faster than code written in English, with no reliable difference in accuracy. We published a paper relating to this contribution [200].

4. Through another empirical study we find that children do not create reliably different measurable learning outcomes whether they are collaborating with one computer and a single user application or two computers and a groupware application. We published three papers related to this contribution [194, 196, 198].

5. We describe a programming environment called Mulspren which is designed to overcome the cognitive gulfs we identified and to provide multiple editable representations of a computer program. This system shows that we can develop multiple notation programming environments that overcome the programming usability problems predicted by our Language Signatures. We published two papers related to this contribution [195, 199].

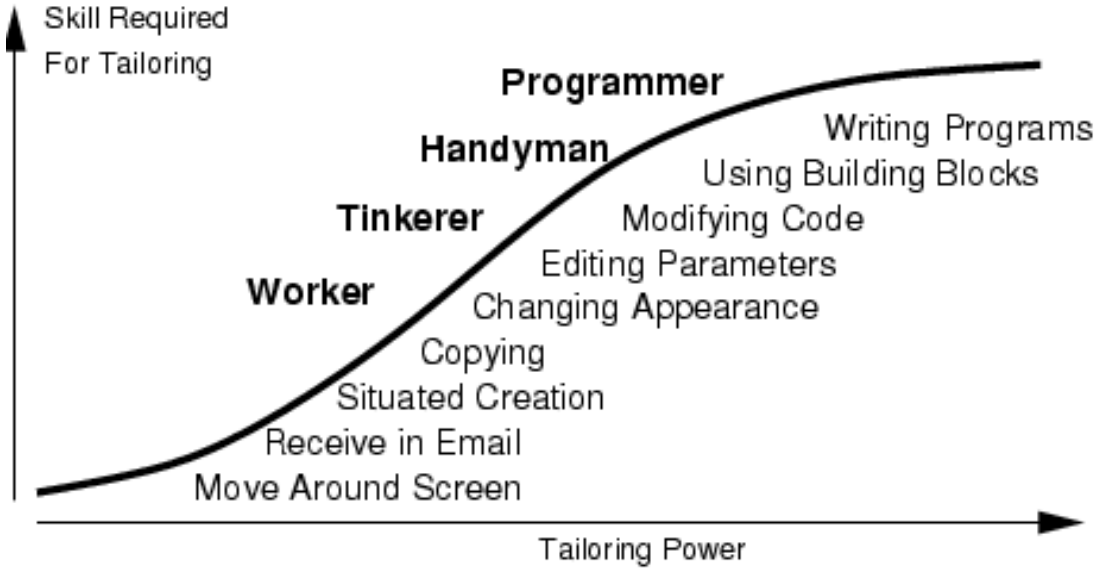


Figure 1.2: Tailoring Techniques in Buttons. After MacLean *et al* [109]. Although McLean's taxonomy focused on Lisp programming we have removed the Lisp references from this figure. This thesis focuses on the activities that require notation manipulation: activities ranging from editing parameters to writing programs.

1.2 Scope

This area, collaborative and multiple-paradigm programming, draws from several areas of research and can be approached from several research perspectives, including computer science, education, and psychology. In computer science, much work has been done building and evaluating programming environments, examining the power of different notations, and mathematically analysing the features of programming in general. In education, much work has examined how children learn, and some work has examined how to use multiple representations of problem domains to increase learning outcomes. In psychology, much research has examined how the brain process and stores information.

The research perspective we use is an Human-Computer Interaction (HCI) perspective. HCI grew from computer science and psychology and has a strong focus on the users of computer software, the types of errors they make, and how to build usable software. HCI also has a strong focus on empirical evaluations: tightly focused evaluations of a restricted aspect of a computer application. These evaluations are typically analysed statistically to try to confidently claim that one computer interface is better than another.

Computer science and HCI researchers have designed many different programming environments for many different users. While much of this work has been designing environments for professional programmers, much work has designed, built, and evaluated programming environments for non-programmers. This thesis examines these programming environments for end-users, and focuses particularly on children's programming environments. Researchers have different opinions about what end-user programming involves, ranging from users writing macros to automate repetitive tasks [36, 139], through to users building visual simulations of things they are interested in [47, 51, 146, 166]. Even simple tasks like programming a mail filter, specifying repeating appointments in a calendar program, and customising a tool-bar can be considered programming. Like MacLean, Carter, Lövstrand, and Moran [109], we believe that end-user programming encompasses a wide range of activities from customising the colour of a button to writing programs using a programming language (MacLean *et al*'s taxonomy is shown in Figure 1.2). In this thesis we are concerned with the subset of end-user programming activities where the end-user must manipulate a notation. The notation can be textual, iconic, visual, or even consist of physical objects. In McLean *et al*'s taxonomy, these tasks range from editing parameters to creating programs.

1.3 Overview

This thesis is written in three parts. Part I examines end-user programming environments and multiple notations, Part II examines collaboration, and Part III describes an implementation of a programming environment that uses results from the first and second parts.

Part I is spread over two chapters: chapters 2 and 3. Chapter 2 introduces our framework for understanding multiple notation programming environments and reviews end-user programming environments using this framework. That chapter finds that multiple notations can be useful, although there are many usability issues with multiple notation programming environments, and provides advice to overcome these issues. To further examine the utility of multiple notations in programming environments, chapter 3 describes an evaluation of multiple notations. The evaluation found that users preferred reading programs with multiple notations. A major limitation of the evaluation was that it only examined people reading and understanding static representations of computer programs.

Part II examines which modes of collaboration are suited for programming environments. This part is also spread over two chapters: chapters 4 and 5. Chapter 4 provides a background

of related work on collaboration, learning, and programming. In that chapter we describe how little work has empirically examined computer-supported collaboration and learning. Chapter 5 describes and analyses an experiment to help fill that gap and determine what supports for collaboration aid problem-solving.

Part III describes our programming environment: Mulspren (MUltiple Language Simulation PRogramming ENvironment). Mulspren's design is heavily influenced by the evaluations and background research described in the first two parts. Chapter 6 describes Mulspren.

Part I

Multi-Paradigm Programming

Chapter 2

An Analysis and Review of the Notations used in Programming Environments

While not directly examining programming environments, some relevant research from education has examined how multiple notations can be used in computer-based learning environments. It examines design issues in multiple representational learning environments (MRLEs), in particular examining the additional cognitive tasks that users must perform when using an MRLE and the pedagogical function of multiple representations in MRLEs [3]. Typically, the domains that these MRLEs are being used to teach are mathematical or mathematically-based, including: the effects of rounding numbers to perform computational estimation; the physics of elastic collisions; learning model logic; understanding the quadratic function; and understanding that there are multiple ways to give a particular amount of change (if, for example, a user is working at a shop) [2].

Unfortunately, this research does not examine computer programming. We believe that computer programming has different properties to the types of domains investigated by the MRLE research for several reasons. First, computer programmers are concerned with solving problems rather than learning about a particular domain: most of the research into MRLEs has examined how to teach people about a particular problem rather than how to let people solve their own problems. Second, the act of editing a computer program is notation manipulation. Teaching computer programming contains elements of both teaching a particular notation and teaching how to manipulate the notation to define (or specify) program behaviour. Third, in the majority of computer programming environments, the notation used for editing a computer program is different to the notation used for watching the program run. For example, users might type tex-

tual code and see (and be able to interact with) a graphical user interface when their program is run. This means that computer programming environments are inherently multiple representation environments. We need an analysis of how notations are being used currently in programming environments as well as an examination of the usability problems with multiple notations that are unique to computer programming.

This chapter examines how notations are used in programming environments. It has several interesting findings. First, many programming environments use multiple programming notations. Second, several usability problems are caused when notations are used in particular ways. Third, there are both advantages and disadvantages in using multiple programming notations: on one hand users can interact with a representation of a computer program that best matches their mental model and, on the other hand, users may get confused or mentally overloaded when moving between representations.

To classify programming environments we use an abstract description of how notations are used called Language Signatures. Using this description we group together environments that use different notations in similar ways and inspect the environments for indications of similar types of usability problems. After classifying the environments, we summarise how the different methods of using notations can cause three types of cognitive programming gulfs: the

gulfs of expression, representation, and visualisation. These gulfs are related to Norman's gulfs of expression and evaluation [125].

This chapter is organised as follows. First, we describe how we classify programming environments. The classification scheme identifies three fundamental programming activities, describes three cognitive programming gulfs, and introduces Language Signatures. Language Signatures are a concise and precise way of specifying how different notations are used for different activities in programming environments. Second, we review many programming environments, and find evidence that the programming gulfs can hinder programming. Third, we analyse and discuss the gulfs. Fourth, compare Language Signatures and Gulfs with other theories of programming. Finally, we examine other (not end-user programming) environments that use multiple notations.

2.1 Classification Scheme

Central to our understanding of how notations are used in programming environments are two concepts: the activities people perform while programming, and a method of specifying how the activities are supported by notations in a programming environment. We determined the fundamental activities by performing an activity-based decomposition of programming, and identified three fundamental activities: *reading* programs, *writing* programs, and *watching* programs run. We call our method of specifying how the activities are supported in a programming environment a Language Signature. This section introduces our three programming activities and Language Signatures. To aid the explanation of the activities and gulfs, we begin by introducing three programming environments that are used as examples throughout this chapter: Logo, StageCast, and OpenOffice, and we examine how notations are used in these environments.

2.1.1 Sample Programming Environments

This section describes how notations are used in three sample programming environments. From these environments we learn two things. First, programming environments use multiple notations and using multiple notations can cause problems for users of the environments. Second, these notations are used for three separate programming activities: reading, writing, and watching. These three activities are the foundation our framework that describes how notations are used programming environments and are discussed further in the following section.

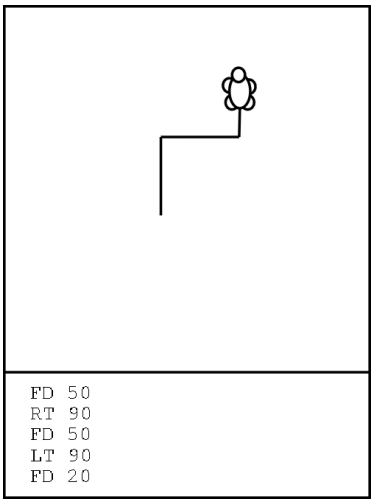


Figure 2.1: A mockup of the Logo Programming Environment. Users read and write programs using text but see a turtle move around drawing lines when their program is executed.

Logo

Logo is an environment where users edit text to control how a turtle moves around a screen and draws lines [137]. For example, a user might program a square by typing:

```
REPEAT 4 [   Do the following commands 4 times
  FD 50      Move the turtle ForwarD 50 units
  RT 90      Turn the turtle RighT 90 degrees
]
```

A mock-up Logo environment is shown in Figure 2.1.

The Logo environment contains two notations. The first is the textual description of what the turtle will do when the program is executed and is used when a user reads or writes a program. The second notation is the collection of lines that are drawn as the turtle that moves around the screen as well as the turtle itself drawing the lines. This notation is used for watching a program run

We postulate that the difference in notations between the reading/writing and watching activities could cause problems for a user as they must reason about one notation using a different notation. For example, a user might ask themselves "*why did the turtle turn right instead of left*"

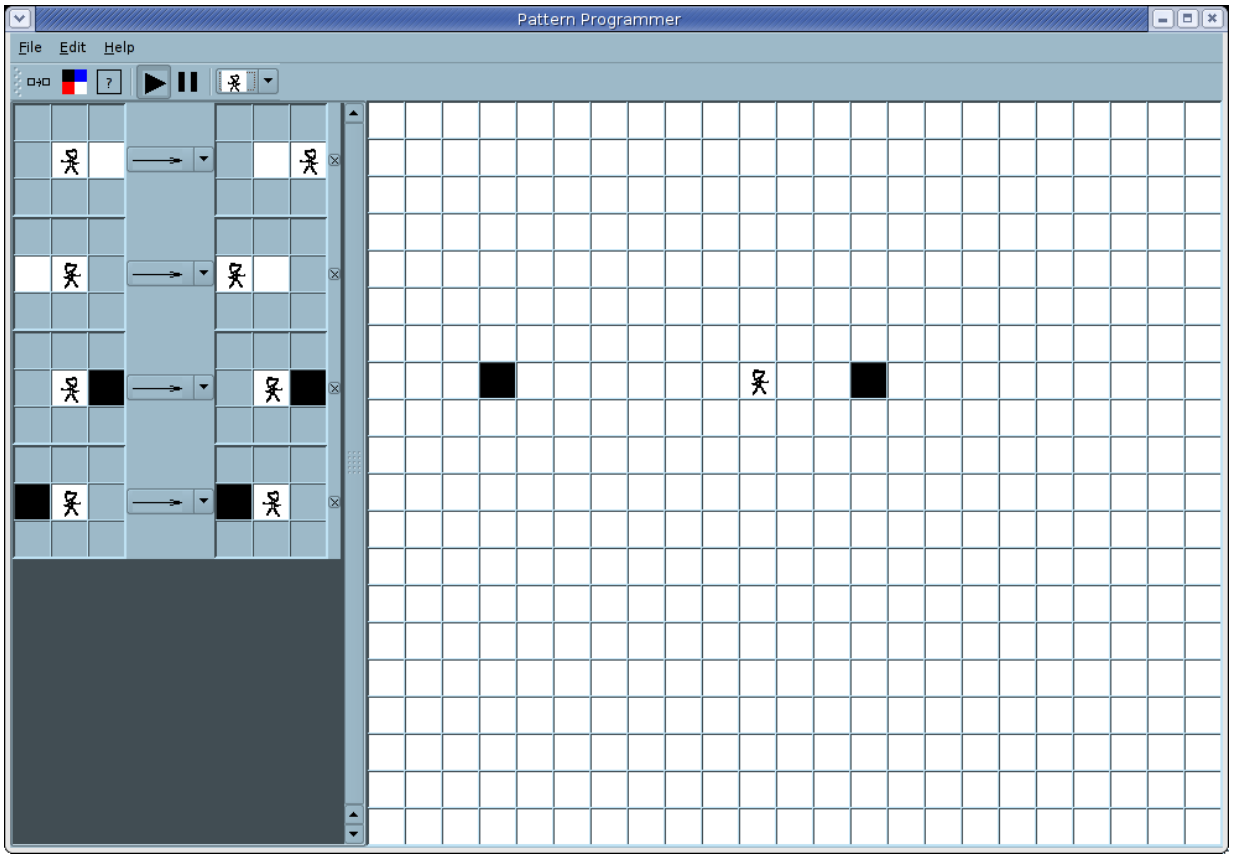


Figure 2.2: A 2D simulation. Many programming environments, including StageCast [167], AgentSheets [151], and Playground [47], use this domain for programming. The environment pictured (PatternProgrammer) was developed by Wright [192].

and have to reason about their program by asking "*Is my mistake typing* ***LT*** *instead of* ***RT*** *or* ***FD*** *instead of* ***BW****?*".

StageCast

StageCast (formally called KidSim or Cocoa) is a programming environment for 2D visual simulations [167]. 2D simulations are programs that run in a two dimensional area of a screen where agents can move around and interact with other. An example simulation is shown in Figure 2.2. StageCast users program using *graphical rewrite* rules to define behaviour in the simulation. A graphical rewrite rule has two parts: an arrangement of agents to search for in the simulation, and an arrangement of agents with which to replace the found agents. An example rule is shown

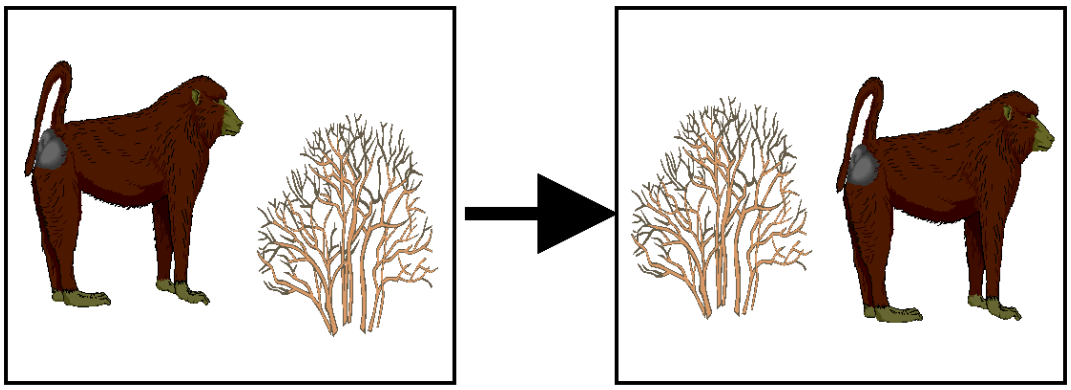


Figure 2.3: A graphical rewrite rule similar to those used in StageCast [167]. A user is writing a program to move a baboon past a bush. When StageCast executes, it will search the simulation for arrangements of agents matching the left-hand side of the rule and replace the agents with the arrangement of agents on the right-hand side of the rule.

in Figure 2.3.

StageCast uses the same notation for reading, writing, and watching programs: users write graphical rewrite rules by manipulating visual representations of agents. When users execute their program they see the same set of agents move around and interact with each other.

Unfortunately StageCast is not perfect. Usability studies of StageCast found that children have trouble predicting what StageCast will do when their program is executing [145]. The problem was caused by a difference in StageCast's rule scheduling algorithm and users' expectations of rule scheduling.

OpenOffice.org

OpenOffice.org is a free word processing program with capabilities similar to Microsoft Word. In particular, OpenOffice.org provides functionality so users can program macros to manipulate their documents. Users program using the symbols provided by the environment: for example, a user wishing to program a macro to transpose two adjacent letters might turn on macro recording, select and cut the letter to the right of the mouse cursor, then move the cursor left and paste. Users can execute the macro and watch letters in their document change places. Unfortunately, if users discover a problem with their program they must either re-demonstrate the program from scratch or edit their program using a complex textual language: OpenOffice Basic (an example

of OpenOffice Basic can be found in Figure 2.8 on page 35).

We postulate that this difference, between the notation used for writing a program and the notation used for reading the program, can cause usability problems for a user of OpenOffice.

2.1.2 Three Fundamental Activities

The previous section describes three programming environments: Logo, StageCast, and OpenOffice, and describes several ways programming environments can use notations. The three environments described use notations in very different ways: Logo uses one notation for editing programs and another notation for watching programs; StageCast uses the same notation for editing and watching programs; and OpenOffice uses one notation for writing and watching programs and another notation for editing programs. By decomposing the editing task into two sub-tasks, reading and writing, we uncover three fundamental programming activities: reading, writing, and watching. These three activities are both fundamental to programming and fundamental to our framework for understanding how notations are used in programming environments. We now describe each activity in depth.

Reading is the act of viewing a notation describing program behaviour, writing is the process of using a notation to describe program behaviour, and watching is the act of viewing program behaviour, either viewing an animation of the notation or viewing the behaviour specified by the notation. The activities are shown in Figure 2.6. Also, we use the term "program expression" to refer to any notation used for writing, "program representation" to refer to any notation used for reading and "program visualisation" to refer to any notation used for watching. Program visualisations can range from animations of the code to agents interacting in a 2D simulation.

To use an example (described in the previous section and also shown in Figure 2.6), consider a user of a word processing program who wants to write a program to transpose two characters. First, using their word processor's programming by demonstration mechanism, they record themselves transposing two characters. This recording of their behaviour is the writing activity, and in this example they are writing using the icons and behaviour provided by their word processor. Next, they execute their macro several times to transpose various characters that were out of order. This execution of their program is the watching activity and they are watching using the icons and symbols provided by the word processor. After they have executed their macro several times, they discover that there is a bug in their macro: the macro transposes the two characters to the left of the cursor instead of the intended behaviour of the two characters surrounding the cursor. To fix their program, the user opens their macro in the macro editor to first read and un-

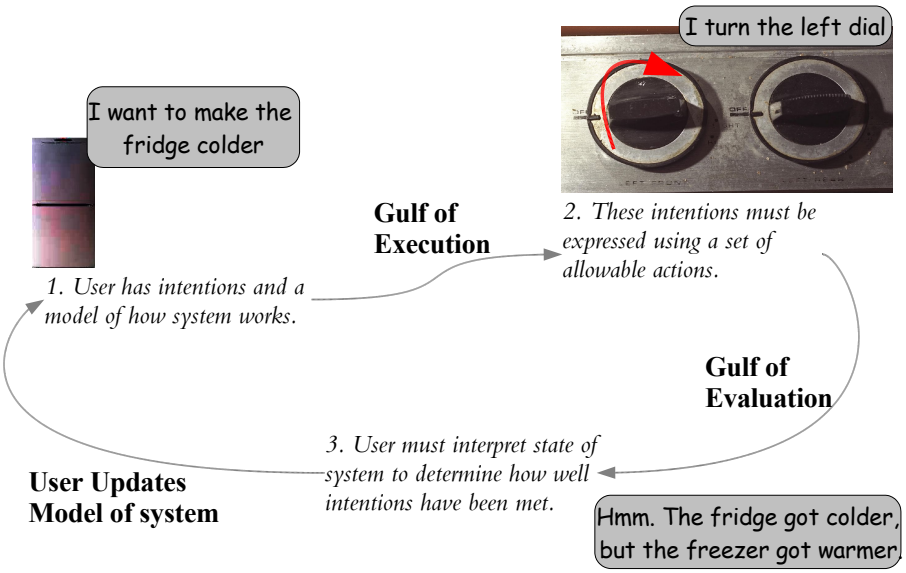


Figure 2.4: Donald Norman's gulfs of execution and evaluation. The gulf of execution is "*the difference between user intentions and allowable actions*" and the gulf of evaluation is "*the amount of effort the person must exert to interpret the physical state of the system and to determine how well the expectations and intentions have been met*." [125]

derstand it. This is the reading activity. In many current word processors, they will see their code in a textual form close to a conventional programming language. In this example, the program representation is the textual form they view to read and write their macro while the program visualisation refers to the dynamic effects caused by the user executing their program.

2.1.3 Three Cognitive Gulfs

In 1988, Norman examined usability problems in the real world [125]. He identified two cognitive gulfs: the gulfs of execution and evaluation. He describes the gulf of execution as "*the difference between user intentions and allowable actions*" and the gulf of evaluation as reflecting "*the amount of effort the person must exert to interpret the physical state of the system and to determine how well the expectations and intentions have been met.*" For example, consider a person attempting to decrease the temperature of a fridge (Figure 2.4). First, they examine the

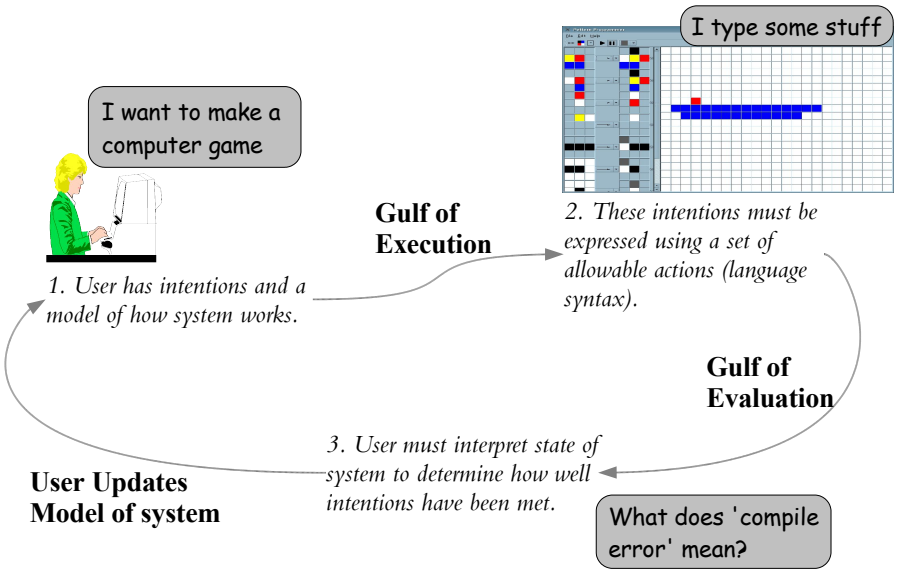


Figure 2.5: Donald Norman's gulfs of execution and evaluation in a programming context [125]. The gulf of execution is the difference between a user's model of desired program behaviour and how they must express their program to the computer. The gulf of evaluation is how hard it is for the user to figure out if they have expressed their program correctly.

dials on the fridge and note two dials. Assuming that one dial is for the fridge and the other dial for the freezer, they turn the left hand dial to the right. This difference between the user's intentions (make the fridge colder) and the allowable actions (two dials that can be turned left or right) is the gulf of execution. After letting the fridge's temperature stabilise, the user notices that while the fridge has become colder, the freezer has also become warmer. The effort the user must expend interpreting the system and understanding why their actions had a different effect than they expected is Norman's gulf of evaluation. Norman's gulfs are shown in Figure 2.4.

Figure 2.5 contextualizes Norman's gulfs in a programming context. While the gulf of execution is still applicable (users must translate their mental model into the symbols of a programming language), the gulf of evaluation has become more complex: a program has both a static representation and a dynamic visualisation that users must interpret to determine how well their intentions have been met. To distinguish the different gulfs of interpreting the representation or the visualisation, we decompose Norman's gulf of evaluation into two gulfs: the gulf of representation and the gulf of visualisation.

So, in a programming context, Norman's two gulfs become three (see Figure 2.6). A gulf of expression is created when the user's mental model of desired program behaviour differs to how the user must express the program. A gulf of representation is created when the user's mental model of program behaviour differs from the program representation. A gulf of visualisation is created when the user's mental model of program behaviour differs from the program visualisation.

As an example, consider a user writing a program to transpose two letters. If the user is using a word processor with a macro recorder, they can write the program using the icons and behaviour of their word processor. As they are likely familiar with the icons and behaviour of their word processor, they have a low gulf of expression. However,consider if their word processor had no macro recorder and the user had to write their program using a conventional textual language. Unless the user is experienced with this conventional textual notation, they will be unfamiliar with the notation and we argue the notation creates a high gulf of expression. Section 2.3 (on page 49) identifies several factors that can influence a user's mental model and change the gulf of expression for a user.

We define the gulf of visualisation as the cognitive difference between a user's mental model of program behaviour and the program visualisation (what the program actually does as it executes). To continue the example of a user writing a program to transpose two letters, consider the possibility that when the users executes their program, the program does not behave as expected.

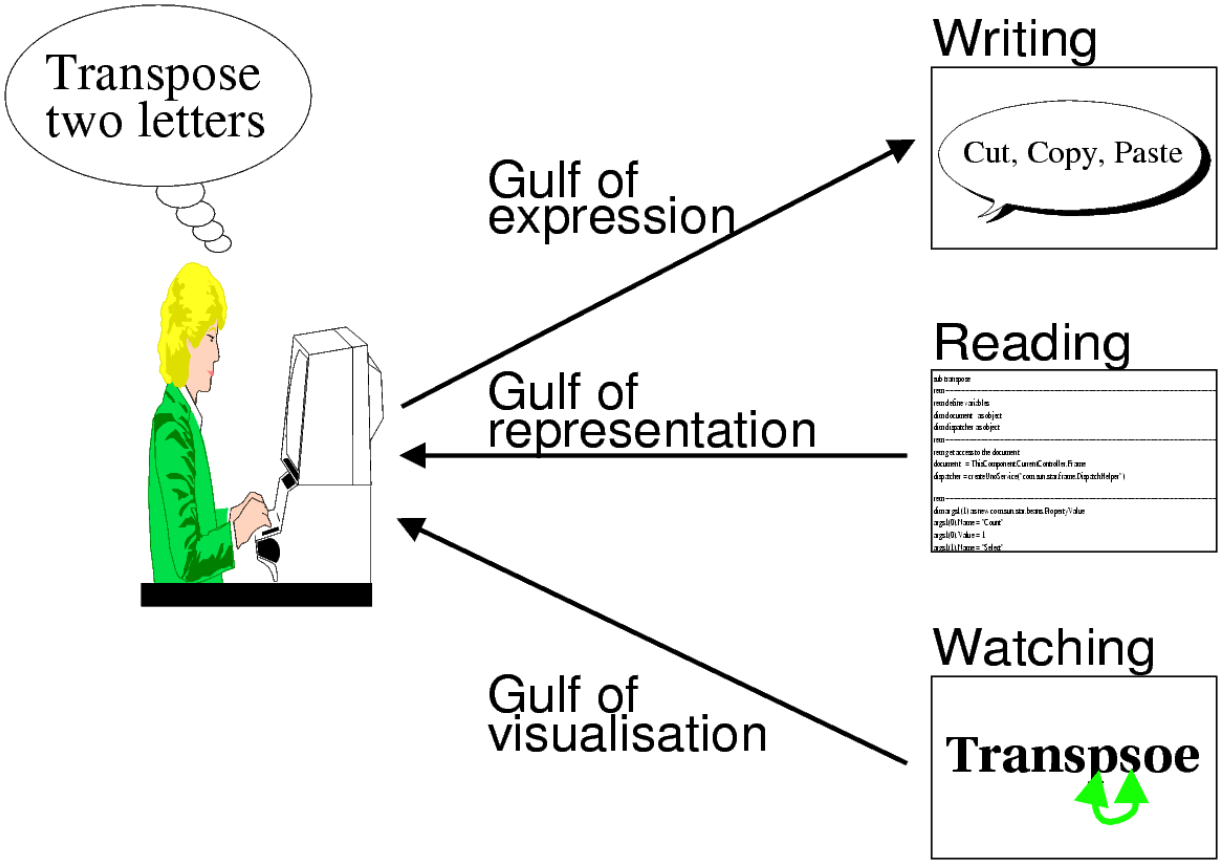


Figure 2.6: The gulfs of expression, representation, and visualisation and their associated activities. In this example a user is writing a program using programming by demonstration mechanisms. The user is reading an computer generated representation of their program. When they run their program they see letters in their document being transposed.

Although the program does not behave as expected, it uses the symbols and behaviour that the user used when they wrote the program, so there is a low gulf of visualisation (in this example we are using the notation the user wrote their program in as a cognate for their mental model).

We define the gulf of representation as the cognitive difference between the users mental model of their program and the program representation. As an example, consider our user who is writing a program to transpose two letters. After they have written their program, they run their program and discover that it doesn't work as expected. When they view their program, they see conventional textual program code. Like the gulf of visualisation, we can use the notation the user wrote the program in as a cognate for their mental model. As the notation consisting of the icons and behaviour of a user's word processor is very different from a conventional textual programming language, there is a high risk of a high gulf of representation.

Smith, Cypher, and Tesler have also done work contextualising Norman's gulfs in a programming context [168]. They argue that the appropriate way to make programming easier is to reduce Norman's gulfs by moving the system closer to the user. Their assumption is that a programming environment provides a consistent representation of the programming system: people write, read, and watch programs using the same notation. Our structure of the three activities (reading, writing, and watching) and the three gulfs (execution, representation, and visualisation) gives us flexibility to consider systems that use different notations for the three activities or different notations for the same activity. We use an abstract syntax for how notations are used in programming environments called Language Signatures, described in the following section.

2.1.4 Language Signatures

By describing how programming environments use notations for the three activities, we can compare different programming environments and discover ways of using notations that help users when programming and ways of using notations that hinder users when programming. Unfortunately, the description of how different notations are used for different activities is long, increasing the chance that someone will misread a notation description. They also make comparing how notations are used in different programming environments harder. For an example description, consider Logo. Logo is a programming environment where programmers describe how a turtle moves around a 2D surface and draws lines. An example Logo program with output is shown in Figure 2.1. The description of how notations are used in Logo is: *users read and write programs using textual commands and watch a turtle draw lines.*

To avoid parsing problems, and to help people compare how different programming environ-

ments use different notations, we use Language Signatures. A Language Signature succinctly expresses how a programming environment uses different notations for the three fundamental programming activities. It is written inside square brackets with plus (+) symbols separating different notations. Each notation is described by stating the activities it supports, abbreviated to RE (Reading), WR (Writing), and WA (Watching) and separated by slash (/) symbols. For clarity, a textual description of the notation's symbols can be subscripted to the description of the activities supported by that notation. Logo's Language Signature is [RE/WR$_{text}$ + WA$_{turtle,\ lines}$].

An alternative to the Language Signature syntax we decided on is to have the Language Signature describe first activities and second the notations used for that activity. The Logo signature might then look like this: [Re$_{text}$ + Wr$_{text}$ + Wa$_{turtle,lines}$]. This way of expressing Language Signatures suffers several limitations: it is hard to immediately determine the number of notations present in an environment; it is hard to determine if Logo uses the same notation for reading and writing, and it is hard to see which activities a notation does **not** support.

While we do not believe there is an ideal Language Signature, they provide considerable leverage for classification and assessment of programming environments. The next section classifies and reviews programming environments by Language Signature, and finds evidence that programming environments with certain Language Signatures risk creating the cognitive gulfs identified in subsection 2.1.3. Also, in section 2.5, we use Language Signatures to examine computer applications that use multiple notations that are not end-user programming environments.

In the following review, we use several descriptions of programming notations. Common descriptions are summarised in Table 2.1.

2.2 Review

This section classifies, reviews, and assesses end-user programming environments. We group programming environments into these categories by examining how many notations are present in a programming environment's Language Signature. For each class of programming environment, we look for empirical data describing usability problems in the programming environment. When we find evidence of usability problems, we perform an analysis relating the usability problems to our three cognitive gulfs. These analyses are collated and examined in Section 2.3.

The review is structured in four parts. The first part examines programming environments that use only one notation for the programming activities; the second part, two notations; the third part, three; and the final part, an arbitrary number of notations for the three programming

Type	Description
Text	The notation is based on users typing or manipulating textual statements. The statements might be conventional code, natural language, or somewhere in between. Example environments that use a textual notation are Alice [31], Hands [136], and C [94].
Iconic	Iconic notations use either icons for programming statements or have animated icons for visualisations. Flowcharts programming notations are an example of iconic programming notations. Environments with an iconic visualisations include StageCast and PatternProgrammer (Figure 2.2).
User Interface	Some programming environments use the set of symbols present in a standard user interface for writing and watching programs. Typically, these environments are Programming By Demonstration (PBD) environments where users can demonstrate behaviour using a standard user interface and the programming environment attempts to infer the user's intended program.
Tangible	Tangible notations use (for program representation) physical items in place of program statements, or (for program visualisation) physical items that move and interact. An example environment that has a tangible representation is AlgoBlock: users join together physical cubes where each cube represents a single Logo statement [177]. An example environment that uses a tangible visualisation is Electronic Blocks: when a program is run, users see physical blocks make sounds, lights, move around, and interact with other blocks [201].

Table 2.1: Description of some common notation types used in Language Signatures.

activities.

2.2.1 Single Notation Environments

During the review, we discovered three different Language Signatures for single notation programming environments. They are: programming environments that do not support reading programs [WR/WA], programming environments that do not support watching programs [RE/WR], and programming environments that use one notation for all activities [RE/WR/WA]. This section finds evidence of usability problems caused by the gulfs of representation and visualisation. The problems occur when programming environments support only two of the three fundamental programming activities, and when a programming environment's model of program execution is different to a user's model.

This section does not cover program animation tools. These tools are not programming environments as they do not provide support for writing programs. However, Language Signatures can be used to describe program animation tools. A discussion of this use of Language Signatures is in Section 2.5.4.

No Reading Support: [WR/WA]

All programming environments with no reading support were programming by demonstration (PBD) or programming by example environments. They are listed (with descriptions) in Table 2.2, and share a usability problem: as there is no program representation, users cannot easily deduce program behaviour. Not being able to deduce program behaviour creates a risk that users will be unsure what their program will do when it executes. This lack of confidence creates the risk of a cognitive barrier for program execution and is an instance of the gulf of representation, shared by all [WR/WA] environments.

To illustrate this factor leading to a gulf of representation, we will use examples taken from papers describing two environments in this category: Gamut [111] and Eager [36]. Both are PBD environments.

Gamut is a PDB environment where users build user interfaces by demonstration. A Gamut user can perform two actions to help Gamut learn the correct behaviour. First, they can demonstrate behaviour to Gamut, and provide a positive hint that the behaviour is wanted and which objects on the screen are important. Second, a user can execute a program and if the behaviour is not what the user wanted (i.e.: if Gamut has inferred an incorrect program),

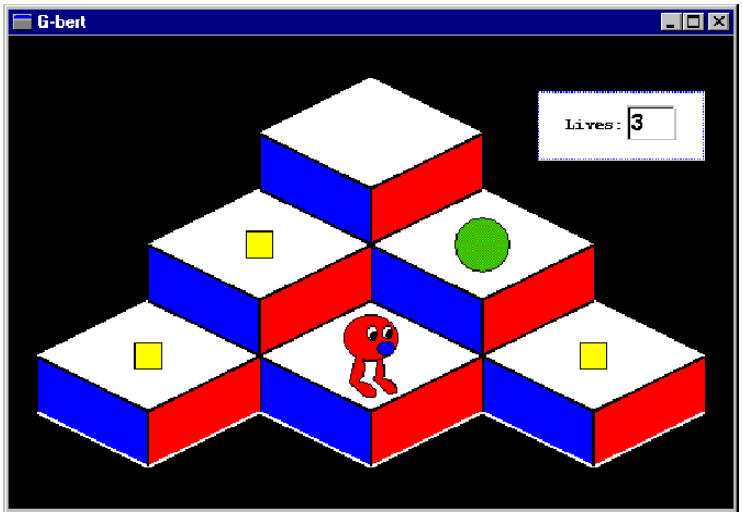


Figure 2.7: A sample Gamut program. Used from [111].

the user can tell Gamut that the behaviour is wrong, and also indicate which objects on the screen behaved incorrectly. A screen snapshot of a Gamut program is shown in Figure 2.7. McDaniel and Myers performed a usability study of Gamut. They videotaped four users performing a series of three tasks, where each task was to create a game using Gamut. This study found that many users wanted to read their program: "*More work is needed to inform the developer about what the system knows and what it can infer*" and "*one potential method for feedback would be to display Gamut's internal language in a readable form*." [112]

Eager is a programming environment for text reformatting. It is implemented as a background process that tracks user actions and searches for cycles of actions. When Eager identifies a cycle, it generalises the next example and asks the user how many times Eager should perform the action. For example, if Eager noticed that a user is italicising telephone numbers, it would tell the user that Eager detected a cycle and ask the user if Eager could italicise other numbers (Eager would also have to infer what types of numbers you were italicising). When describing an evaluation of Eager, Cypher wrote *"all subjects were uncomfortable giving control when Eager took over."* [36] Cypher helped users overcome their discomfort by adding step features so users could step forward and backward through their program. While these step features reduce the gulf of visualisation by adding stronger

watching support (and thus reduce Norman's gulf of evaluation), we argue a more successful change would be to add a program representation so people can understand what Eager has inferred.

No Watching Support: [RE/WR]

Environments in this category have neither a visualisation of the program running nor any animated output. These environments risk creating a gulf of visualisation as users are unable compare expected program behaviour with actual program behaviour. Many conventional programming languages fall into this category. For example, languages such as C or Python provide no environment-generated visualisation of the program running. To supplement these environments, programmers often use debuggers to p tasks including stepping through code, reversing execution, watching data change, and examining how multiple threads (or separate flows of control) interact. This supplementation adds watching support to the environments, without which it would be very hard to debug and understand programs written in conventional programming languages.

Our research found two [RE/WR] programming environments with no support for watching: Vista and Pygmalion.

Vista is a programming environment for data processing in distributed computer networks [160]. Vista uses an object-oriented paradigm to define signal and data flow. Vista programs are written and read using an iconic notation and has a [RE/WR$_{iconic}$] Language Signature.

Pygmalion is a programming environment where users define visual control-flow programs by demonstration [165]. Pygmalion provides support for executing partially completed programs, but provides neither an animated representation nor produce any side effects that can be watched. It has a [RE/WR$_{iconic}$] Language Signature.

These two programming environments share a usability problem: a user can not know what their program is doing as it executes. This lack of knowledge creates the risk of users not being able to cognitively map between what their program is doing and what they intended their program to do. To avoid creating the risk of a gulf of visualisation, environment developers should add watching support so that users of the programming environment can map between actual and expected program behaviour.

System	Description
TELS [WR/WA$_{textual}$]	TELS [189] is a programming by demonstration system for text reformatting. Users demonstrate one example of how they want text reformatted and TELS generalises for other examples. All modifications on users' data must be confirmed by the user, and when TELS makes an incorrect generalisation the user must correct TELS. TELS uses this new information to generalise future examples.
Gamut [WR/WA$_{widgets}$]	Gamut [112] is a programming by demonstration system for building interactive user-interfaces. Gamut users demonstrate examples incrementally and the system generalises behaviour. To help the system generalise correctly, users can provide hints to help the system understand which objects are important. Users can also place guides to help the system generalise correctly. Guides are objects that are not shown when the program executes.
Eager[WR/WA$_{user\ interface}$]	Eager [36] is a background process which is constantly watching the user and examining their actions for repetitions. When Eager finds a repeated list of actions it generalises the list to create the next example and asks the user how many times to run the generalise loop. The user can choose to ignore Eager (and Eager will continue searching for repeated actions) or to perform the actions that Eager suggests.
Simulacrum-2 [WR/WA$_{iconic}$]	Simulacrum-2 [17] is a programming by demonstration environment to program elevator animations. Users create multiple snapshots of elevators and Simulacrum-2 attempts to deduce correct constraints to control elevator behaviour [68].

Table 2.2: One language used for Writing and Watching [WR/WA]

Conventional programming environments risk creating this gulf as they do not produce any default output: a programmer must either use a debugger or specify in their code what text to display and when (also known as *debugging by print statements*). If a programmer does not tell their program to produce output, and they do not use a debugger, the programmer risks creating a gulf of visualisation: their model of expected program behaviour could be vastly different from the actual program behaviour (especially if they have unintentional bugs in their program). Using a debugger avoids the risk of a gulf of visualisation by animating the code. Authors of programming environments should do the same: provide support for code animation, provide a debugger to step through the code, or both.

Environments with Support for All Activities :[RE/WR/WA]

During the review we discovered that single notation environments with support for all activities were either: spreadsheets, programming by demonstration systems, or visual languages with animated representations. They are described in Table 2.3. This class of programming environments have low risks of creating gulfs of representation or visualisation: the notations are the same for each activity. The environments can create a gulf of expression if the notation used for reading is different from a user's cognitive model of program behaviour.

An interesting [RE/WR/WA] environment is StageCast [167] (described in detail in Section 2.1.1). StageCast is a simulation programming environment for children where rules are created in a programming by demonstration style. Figure 2.3 contains an example of a StageCast rule. Researchers have found that children create programs easily using this programming notation, although the children do not necessarily understand all the details of the environment [57, 145].

Despite the simplicity of programming in StageCast, StageCast has a usability problem: users have trouble understanding the scheduling behaviour of the environment when a program is executing. Radar, Brand, and Lewis evaluated an early version of StageCast (then called KidSim), and found that children did not easily understand StageCast's rule execution system [145]. This lack of understanding leads to concrete usability problems. For example, Svanæs wrote:

> *One of the groups developed a "world" where wolves ate rabbits. They first programmed the rabbits to move around on random. Next, they programmed the wolves to eat rabbits whenever a rabbit was in front of a wolf. Both rules worked very well when tested out separately, but when run in combination (free running rabbits together with rabbit-eating*

wolves) no rabbits were eaten. We were not able to help the kids with this problem, and contributed it at first to a bug in Cocoa.

Going through the tapes afterwards, we did a detailed analysis of the problem and found that with Cocoa's computational model the rabbits would always escape the wolves before they got eaten. To understand the problem in detail we had to learn about Cocoa's strategies for rule triggering. This analysis could not have been done by anybody without a background in computer science.

The interesting lesson learned was that with a relatively simple design, the system did not behave as expected due to Cocoa's hidden computational model. [178]

This problem is an example of the gulf of visualisation: it occurred because the designer's view of the scheduling algorithm is different to the user's view, making it hard for a user to understand why their program isn't working as they expect.

[RE/WR/WA] Programming Environments

Spreadsheets	
System	**Description**
Forms/3 [RE/WR/WA$_{spreadsheet}$]	Forms/3 is a spreadsheet programming environment [24]. It is a research environment developed to push the boundaries of the spreadsheet paradigm.
NoPumpG [RE/WR/WA$_{spreadsheet}$]	NoPumpG users can program interactive graphics using the spreadsheet paradigm [103].
Programming by Example/Demonstration Environments	
System	**Description**
StageCast [RE/WR/WA$_{iconic}$]	Previously known as Cocoa and KidSim, StageCast is a programming by example (or by graphical rewrite rules) system designed for children to build visual simulations [166]. An example graphical rewrite rule is shown in Figure 2.3 .

Table 2.3 continued on next page. . .

. . . Table 2.3 continued from previous page

System	Description
BitPict [RE/WR/WA$_{screen\ pixels}$]	BitPict is similar to StageCast, but the graphical rewrite rules operate on screen pixels rather than agents [51]. This subtle difference means that BitPict can operate on arbitrary user-interfaces, instead of in the domain of visual simulations.
Viscuit [RE/WR/WA$_{Fuzzy\ rewrite\ rules}$]	Viscuit is another programming by demonstration system for visual simulations using graphical rewrite rules [75]. Unlike StageCast, Viscuit uses fuzzy rule rewriting logic, where the first half of a rule will match if the objects are not the correct rotation or distance from each other. Viscuit modifies the graphical rewrite rule based on how close the match is and how the match differs from the original rule.
ToonTalk [RE/WR/WA$_{3D\ objects}$]	ToonTalk was designed to make programming similar to playing video games [89]. Programmers can program robots by demonstration to perform tasks, and see the robot perform the tasks. Their program is represented as a series of snapshots of the robot being trained.
Geometer's Sketchpad [RE/WR/WA$_{2D\ objects}$]	This environment was designed to help students gain an understanding of geometry. Users build constraint-based geometric objects and then can resize the shapes and watch the geometric constraints modify the object's shape [88].
Sketchpad [RE/WR/WA$_{iconic}$]	Sketchpad users draw programs with a light pen. Users can define constraints on their drawings and watch their drawings animate [176].
Visual languages with Animated Representations	
System	**Description**

Table 2.3 continued on next page. . .

. . . Table 2.3 continued from previous page

System	Description
Electronic Blocks [RE/WR/WA$_{tangible\ blocks}$]	Electronic blocks is a tangible language designed to teach children about Boolean algebra [201]. Users connect input and output blocks (eg. light, sound, movement) using logic blocks (eg. and, or, not).
Boxer [RE/WR/WA$_{iconic}$]	Users program using boxes [39]. The boxes represent the program, the data, and the program's visible representation.
ObjecTime [RE/WR/WA$_{mixed\ text\ and\ iconic}$]	ObjecTime combines two programming paradigms (finite state machines and text) to program simulations of real-time event-driven systems [127].
The Interface Construction Set [RE/WR/WA$_{iconic}$]	Users program user interfaces using interface widgets and an iconic data-flow language [169].
Pict [RE/WR/WA$_{iconic}$]	Pict programs are represented as an iconic flowchart [58]. The flowcharts are animated at run-time.
Show and Tell [RE/WR/WA$_{iconic}$]	Show and Tell is a visual data-flow programming environment. Users read and write programs using an iconic flowchart-like syntax [96].
Prograph 2 [RE/WR/WA$_{iconic}$]	Prograph 2 uses an iconic combination of a flowchart language, a data-flow language, and a logic language to write programs [35].
Icicle [RE/WR/WA$_{iconic}$]	Icicle is a rule-based programming-by-demonstration environment for children [162]. Icicle's representation of rules is the animation of the demonstration which created the rule—a representation that naturally shows parallelism.

Table 2.3: Table of programming environments with one language used for all three tasks [RE/WR/WA]

2.2.2 Two Notation Environments

During the review, we discovered the majority of end-user programming environments use two notations. To help with the review of these environments, we sub-classify them into three groups. The first group contains two-notation programming by demonstration environments. They have [WR/WA + RE], [WR/WA + RE/WR], or [WR/WA + RE/WR/WA] Language Signatures. The second group consists of two-notation conventional-style programming environments. They have [RE/WR + WA] or [RE/WR/WA + WA] Language Signatures. The final group comprises dual notation environments. They have [WR/RE + WR/RE] Language Signatures. Although we discovered no common Language Signatures between the groups, we don't believe this is an artifact of programming by demonstration or conventional-style environments, and investigate this relationship at the end of this section.

This section finds evidence for usability problems caused by the gulfs of expression, representation, and visualisation. The evidence indicates: a gulf of expression can be created when a programming environment uses a read-only representation, a gulf of representation can be created when an environment uses multiple notations, and a gulf of visualisation can be created when a programming environment does not animate a program representation. To avoid the gulfs of expression and representation program environments should: let users edit representations, keep notations consistent at all times, and provide support for users to understand the relationship between different notations. A more detailed analysis of the gulfs is in section 2.3 on page 49.

Two Notation Programming by Demonstration Environments

Programming by demonstration environments (also called programming by example environments) try to infer what a computer program should do based on a user's manipulations of objects in the program's output domain. They have many well documented problems ranging from inferring what a demonstration means to communicating what a program will do when it is run. As many solutions have been proposed to help make correct inferences of user intentions [91, 167, 191], and we have already examined how to communicate what a program will do (provide a program representation, see subsection 2.2.1), those problems will not be analysed in this section. Rather, we will look for evidence of new types of problems and solutions to those problems.

Our review found that writing a program using one notation and reading a program using

a different read-only notation risks creating a gulf of expression and causes usability problems. Unfortunately, the obvious way to fix this problem, to unify the notations, risks creating a gulf of visualisation (described in Section 2.2.1). Some environments avoided the gulf of expression by making the read-only notation editable. This modification of the programming environment means that users can write programs using multiple notations: evidence that multiple notations are useful to avoid gulfs and their related usability problems.

[WR/WA + RE] environments show a read-only representation of the inferred program. The representation is usually a textual notation, but could be iconic, or even tangible (Table 2.4 describes environments in this category in detail). An interesting environment is Prototype-2: it provides evidence of notation-based usability problems.

Prototype-2 is a research system designed to help first year programmers learn to program in PASCAL. It provides a PBD interface where users see PASCAL program being generated as the users manipulate the user-interface. Gilligan noticed that using a read-only representation created usability problems: "*The lack of unwind, deletion, and editing capabilities hinders programming—if you make one mistake, you have to begin again.*" [54]. This problem, of users not being able to edit their programs, is amplified if the programming environment can not infer the correct behaviour (something that Piernot and Yvon believe is intractable):

> "*Inferring the* user's intentions *[original emphasis] is a crucial problem for programming by demonstration. Nevertheless, sometimes making the right inference is intractable for current learning algorithms, even if more than one example is supplied.*" [143]

[WR/WA + RE/WR] environments show an editable representation of the inferred program. Like [WR/WA + RE] environments, the representation is usually a textual notation, but it could be iconic or tangible (Table 2.5 describes environments in this category in detail). The editors for the representation have been based on: property sheets [72], English-like stimulus-response descriptions [116], and story boards without timing [104]. By avoiding the risk of a gulf of expression, environments in this category are more usable than environments with a read-only representation: "*The use of a second level editor eliminates the need to constantly invoke dialogues to confirm inferences, as was done in early systems*

such as Peridot." [104] (Peridot is a [WR/WA$_{user\ interface}$ + RE$_{text:dialoguebox}$] environment [117].)

Environments in this category risk creating a gulf of representation. When users write their program using one notation and read their program using a different notation they could have cognitive problems mapping between the two notations. OpenOffice.org is an environment that displays this gulf. A program to transpose two characters consists of four user actions (select character, cut, move left, paste), however the macro created by OpenOffice has 36 lines of code (see Figure 2.8) that seemingly bear no relation to the original user actions. To avoid this gulf programming environment authors should provide support to help users map between the actions they use during program demonstration and the code that is generated by the environment.

[WR/WA + RE/WA] environments animate a read-only representation of the inferred program at run-time. Chimera is the only system we found with this Language Signature [102]. Chimera users create multiple snapshots of their desired program, and Chimera deduces program constraints based on these snapshots. Users can view but not modify the constraints. When their program is run the constraints are visible. Chimera risks creating a gulf of expression in the same way as [WR/WA + RE] environments.

[WR/WA + RE/WR/WA] environments show an editable representation of the inferred program that is animated at run-time. Environments in this category are described in Table 2.6. One sample environment is Familer (see Table 2.6). An evaluation of Familiar found the support for modification of the generated language was weak: users would like more control of the generated programs [140]. This is further evidence that any language presented to a user must be completely editable or the environment risks creating a gulf of expression.

[WR/RE/WA + WR/RE] environments are a variation of [WR/WA + RE/WR/WA] environments where the editable representation is not animated, but the symbols used to program by demonstration are animated. Rehearsal World is a programming environment for teachers who are not programmers [49]. Users create interfaces visually and can specify behaviour either by demonstration or by writing Smalltalk code. The visual representation is animated at run-time—actors and cues correspond to objects and messages.

```
sub transpose
rem ----------------------------------------------------------------------
rem define variables
dim document   as object
dim dispatcher as object
rem ----------------------------------------------------------------------
rem get access to the document
document   = ThisComponent.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem ----------------------------------------------------------------------
dim args1(1) as new com.sun.star.beans.PropertyValue
args1(0).Name = "Count"
args1(0).Value = 1
args1(1).Name = "Select"
args1(1).Value = true

dispatcher.executeDispatch(document, ".uno:GoLeft", "", 0, args1())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Cut", "", 0, Array())

rem ----------------------------------------------------------------------
dim args3(1) as new com.sun.star.beans.PropertyValue
args3(0).Name = "Count"
args3(0).Value = 1
args3(1).Name = "Select"
The largest number of end-user programming environments use two notations.
args3(1).Value = false

dispatcher.executeDispatch(document, ".uno:GoRight", "", 0, args3())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Paste", "", 0, Array())


end sub
```

Figure 2.8: Automatically generated OpenOffice.org macro to transpose two adjacent characters.

Environments in this category risk creating a gulf of visualisation: users build a mental model of their program based on the textual code, but the textual code is not animated when the program is executed.

Conventional style environments

Conventional style environments use a notation to describe behaviour in a different domain. That is, they use one notation for reading and writing (and possibly watching), combined with a different notation for watching. Their Language Signatures are [RE/WR + WA] and [RE/WR/WA + WA], and the environments are described in Table 2.7. Two research areas provide arguments that watching support helps users program. First, some research finds that watching support reduces the gulf of visualisation and aids program understanding [73, 185]

The second research area comes from the area of children's literacy. Rose's studies of children learning to read and write found that: "*one of the best predictors of reading and writing success is the amount of 'expert' reading children have seen in the home as their parents read to them or write stories as children dictate*" [158]. For computer programs, computers can act as expert program readers by adding watching support — providing program animations. Some environments even provide programming by demonstration mechanisms and thus act as expert writers.

Additionally, Vygotsky argued that interaction with adults and more competent peers is a pivotal factor in effective learning [186]. When learning how to program computers there are two sources of more competent peers that the student can observe and interact with to test and build their understanding. First, the student can work with human collaborators (teachers or fellow students). This may be through: normal face-to-face interaction, some form of groupware support, or a computerised agent that emulates human behaviour. Second, the computer itself provides an interactive platform for experimentation. When the student writes a program, its behaviour is a dynamic embodiment (the watchable form) of the program (the readable form) that the student has expressed (the written form). Using natural language as an analogy, this is equivalent to the student expressing a verbal phrase (for instance, saying the words "The cow runs"), then seeing *both* the words *The cow runs* and an animation of a running cow on the screen (see Figure 2.6 on page 20). These arguments from children's literacy research provide powerful reasons to increase the level of watching support in programming environments for children: increasing the level of watching support can increase the level of understanding of the users by reducing the gulf of visualisation.

System	Description
Prototype 2 [WR/WA$_{user\ interface}$ + RE$_{text:pascal}$]	Prototype-2 was designed to help 1^{st} year computer science students learn the PASCAL programming language. Students program a user-interface by demonstration and see their program being generated (Prototype 2 generates PASCAL code) [54]. The generated code is read-only. The same symbol set is used for writing and watching: the set of symbols in the user-interface, while PASCAL code is sued for reading.
AIDE [WR/WA$_{user\ interface}$ + RE$_{text}$]	AIDE is a macro recorder with the ability to generalise recorded macros [143]. Users record macros by using the user interface, and AIDE shows the macro in a textual form. Users can modify macros by giving additional examples of interface behaviour. The same symbol set is used for writing and watching: the set of user-interface widgets. Users read their programs using a read-only textual description.
Tinker [WR/WA$_{iconic\ objects}$ + RE$_{text:lisp}$]	Tinker is a PBD environment that produces Lisp code [105] . Users demonstrate multiple examples to specify behaviour. Tinker's programs operate on iconic objects (e.g. blocks).
Peridot [WR/WA$_{user\ interface}$ + RE$_{text:dialoguebox}$]	Peridot is a programming by demonstration environment for describing dynamic user-interfaces [117]. When a user is demonstrating a program, Peridot tells the user about any inferences it is making, and the user has a chance to tell Peridot that the inference is incorrect. There is no way of viewing a list of inferences.

Table 2.4: Programming by demonstration systems with a read-only representation [WR/WA + RE].

System	Description
Pavlov [WR/WA$_{user\ interface}$ + RE/WR$_{text}$]	Pavlov is an programming by example system [190]. Pavlov users write programs by specifying stimuli (for the computer to respond to), and then defining responses (what the computer should do when the stimuli happens). While seeming similar to a graphical rewrite rule, a stimuli-response must be demonstrated to the computer and the computer has to infer correct behaviour. Pavlov also presents inferred rules to the programmer and the programmer can modify these rules. The language used for writing and watching is the set of symbols in the user-interface, whereas the language used for reading and writing is the set of rules, expressed textually.
Triggers [WR/WA$_{screen\ bitmaps}$ + RE/WR$_{iconic}$]	Triggers is a programming by demonstration environment for arbitrary applications [144]. Triggers users specify rules where the trigger is a screen bitmap (like BitPict) and the action is specified in a PBD style (like Mondrian).
Juno [WR/WA$_{drawing\ elements}$ + RE/WR$_{text}$]	Juno is a constraint based drawing program where code to describe a drawing is inferred from a drawing [123]. Users can edit the code.

Table 2.5: Programming by demonstration environments that show an editable representation [WR/WA + RE/WR].

System	Description
SmallStar [WR/WA + RE$_{text}$]	SmallStar uses programming by demonstration techniques to help users build editable scripts, in a similar way to Pavlov and Rehearsal world [72]. The scripts are animated.
Familiar [WR/WA$_{user\ interface}$ + RE/WR$_{text}$]	Familiar is a programming by demonstration system that notices repetitive actions a user is doing at a user interface level [140]. It then presents these cycles to the user and lets the user modify the cycle and run the cycle a set number of times.

Table 2.6: Programming by demonstration environments that show an editable, animated representation [WR/WA + RE/WR/WA].

As further evidence that increasing watching support can reduce the gulf of visualisation, consider an expert programmer debugging a conventional textual program (for this example, written in Java). Invariably, the expert will invoke a debugger to step through the program and perhaps watch how objects and variables are changing over time. In this case, the debugger is acting as a program visualisation tool and is increasing the level of watching support available to the programmer — the debugger is reducing the gulf of visualisation.

Two-Notation Conventional-Style Programming Environments

System	**Description**
Moose Crossing [RE/WR$_{text}$ + WA$_{text}$]	Moose Crossing was designed to let children design and interact with multi-user text-based virtual worlds [20]. Children can populate the worlds they create with any objects they want, and can build virtual rooms and cities. They can then interact with other children in the rooms they created, using the objects they created.
Hands [RE/WR$_{text}$ + WA$_{iconic}$]	Users program simulations using a textual notation designed for children [135]. The notation was designed by asking children to define behaviour of agents in a Pacman game, analysing their answers, and building a programming language based on the children's language.
FAR [RE/WR$_{text/iconic}$ + WA$_{iconic}$]	FAR was designed to help small businesses build web-pages for interactive e-commerce. Users program their web-pages using a combination of a rule based system and a spreadsheet model [25].

Table 2.7 continued on next page...

. . . Table 2.7 continued from previous page

System	Description
HyperCard [RE/WR$_{text}$ + WA$_{user\ interface}$]	HyperCard is used to design user interfaces [7]. It uses a card metaphor, and users program by manipulating a natural-language notation. Unfortunately this notation contains several fundamental flaws [183]. These range from inconsistent use of syntax and semantics to lack of generality and lack of error handling. The flaws are so widespread that Thimbley and Cockburn write: "*it is impossible to say 'HyperTalk fails such-and-such well known principle' — rather, HyperTalk fails principles wholesale* [183]."
Logo [RE/WR$_{text}$ + WA$_{turtle,\ lines}$]	Logo programmers manipulate a textual notation to build geometric shapes [137]. The textual notation contains commands to move a virtual turtle around a screen—the turtle draws the lines. Logo was designed to help children learn geometry, but was used in schools on the rational that learning programming would teach skills that are transferable to other domains.
Star Logo [RE/WR$_{text}$ + WA$_{turtles,\ lines}$]	Star Logo is a parallel version of Logo [154]. Users program many turtles to build complex symmetric shapes [154] .
Playground [RE/WR$_{text}$ + WA$_{iconic}$]	Playground was possibly the first rule-based simulation programming environment [47]. A precursor to AgentSheets or StageCast, users wrote programs in a natural-language style (a textual notation), to define behaviour of agents in a visual simulation. An evaluation of Playground found that children were annoyed with various aspects of the user interface: "*bugs, unnatural syntax conditions, relatively low speed of the interpreted environment, and deficiencies in error handling and reporting.*" [47]

Table 2.7 continued on next page. . .

. . . Table 2.7 continued from previous page

System	Description
Flogo [RE/WR/WA$_{text}$ + WA$_{robots}$]	Flogo is a programming environment designed to help programmers define robot processes [73, 74]. Flogo has support for concurrency designed in at the lowest level and Flogo programmers read and write textual code. The textual notation is animated at runtime, and users see robots performing tasks.
Tcl/Tk [RE/WR$_{text}$ + WA$_{user\ interface}$]	Tcl/Tk is a common textual environment to design graphical user interfaces [132]. Users read and write textual code and when the program is run users see a graphical user interface.
ClockWorks [RE/WR$_{mixed}$ + WA$_{user\ interface}$]	ClockWorks is a research system investigating how user interfaces can be defined using functional languages. The underlying functional language uses a textual notation loosely based on Hascall, and users structure their program using a visual hierarchical editor [62].
Microsoft Visual Basic [RE/WR$_{mixed}$ + WA$_{user\ interface}$]	User interface toolkit from Microsoft.[1] Users read write programs using a combination of user-interface widgets and textual code. When the program is run, users interact with a conventional GUI.
ThingLab [RE/WR$_{graphical}$ + WA$_{iconic}$]	ThingLab is a graphical constraint based simulation programming environment [14]. Users can construct "*dynamic models of experiments in geometry and physics, such as simulations of constrained geometric objects, simple electrical circuits, mechanical linkages and bridges under load.*" [16] Users can define their own types of constraints or can use built-in constraints [15].
Animus [RE/WR$_{graphical}$ + WA$_{iconic}$]	Animus is a graphical constraint based animation programming environment [41]. It is based on ThingLab, and has temporal constraints to provide animation.

Table 2.7 continued on next page. . .

[1] Visual Basic, Microsoft Corporation, http://www.microsoft.com.

. . . Table 2.7 continued from previous page

System	Description
ChipWits [RE/WR$_{iconic}$ + WA$_{iconic}$]	ChipWits' users program virtual robots to solve puzzles in a virtual world. The users program using an iconic flowchart notation [4].
Lapidary [RE/WR$_{iconic}$ + WA$_{user\ interface}$]	Lapidary [204] is a part of part of the Garnet user interface development environment [121]. Users read and using write programs using a combination of user-interface widgets and visual constraints. Users watch just the user-interface. C32 is used to specify complex constraints [119], and the Gilt environment is used to filter expressions [120] .
Play [RE/WR$_{iconic}$ + WA$_{visual}$]	Play is a simulation programming environment for children [181]. Users build programs by first defining simple animations for agent and, second, users write a script that defines how to combine the animations.
Sam [RE/WR/WA$_{3Dicons}$ + WA$_{3D}$]	The structure of a SAM [53] program is similar to an ObjecTime program [127], but agents are specified and animated in a 3D environment. SAM 's output domain is a 3D world.
AlgoBlock [RE/WR$_{tangible\ blocks}$ + WA$_{turtle,lines}$]	AlgoBlock users write Logo programs using physical blocks—each AlgoBlock block represents one Logo command [177]. Suzuki and Kato used AlgoBlock to examine how children collaborated when there is little contention for input devices.
Dick Smith's Funway into Electronics [RE/WR$_{resisters,\ diodes,\ etc}$ + WA$_{tangible}$]	Children create electronic circuits to perform preset tasks.[2] To scaffold the children's understanding of circuits, they are presented with problems of increasing difficulty. This approach of scaffolding learning is similar to that used in ToonTalk [90].
Squeak [RE/WR$_{text}$ + WA$_{arbitary}$]	Squeak [81] is an implementation of Smalltalk [82]. Users structure their programs using a visual IDE and the Morphic toolkit [110], but must write Smalltalk code. Squeak provides many output domains ranging from 2D or 3D graphics to sound and voice.

Table 2.7 continued on next page. . .

[2] Dick Smith Electronics, http://www.dicksmith.co.nz.

. . . Table 2.7 continued from previous page

System	Description

Table 2.7: Two-Notation Conventional-Style Programming Environments. Programming environments with a Language Signature of [RE/WR/WA + WA] or [RE/WR + WA] (they use one possibly animated language for reading and writing, and another for watching).

Dual Notation Programming Environments

We discovered two environments that provide visual interfaces to conventional textual programming notations. They both have a [WR/RE + WR/RE] Language Signature. These environments provide multiple representations of the same underlying program: they use two notations for reading and writing, and provide no notation for watching. Users can move between the notations and changes in one notation are reflected in the other. Although both environments we found in this category have a [RE/WR + RE/WR] Language Signature they could easily add watching support by animating either or both representations.

The two environments are TinkerToy [42] and C^2 [98]. TinkerToy provides an iconic interface to Lisp and C^2 provides an iconic interface to C. The literature describing these papers describes two notation-related usability problems with this class of language. The problems occur when changes in one notation are not immediately reflected in the other notation and when either of the notations are not animated when a program executes. The first problem provides evidence that multiple notation programming environments should keep the notations consistent at all times. This is another case of the gulf of representation: there is a barrier for users to read their program because the system is displaying two different versions of their program (one old and one new). The second problem is an instance of the gulf of visualisation. To avoid the risk of this gulf, environment authors should create environments with a [RE/WR/WA + RE/WR/WA] Language Signature.

To overcome the gulf of representation, Edal recommends removing one notation and executing the iconic structure directly (rather than first converting the iconic structure to Lisp) [42]. He believes this would create an "*integrated program visualisation tool*," letting users watch programs and data structures change as a program executes. This would change TinkerToy's Language Signature to [RE/WR/WA]. Another solution would be to consider each representa-

Programming by Demonstration	**Write**	**Read**	**Watch**
Domain Notation	All systems	No systems	All systems
Other Notation	Some systems	All systems	Few systems

Conventional style	**Write**	**Read**	**Watch**
Domain Notation	No systems	No systems	All systems
Other Notation	All systems	All systems	Some systems

Table 2.8: Differences between two-notation PBD and conventional style environments.

tion as a different view of the same underlying program and data and update all views whenever the underlying abstract model is changed. This technical solution would reduce the gulf of representation by keeping all representations strictly consistent at all times.

One problem with keeping notations strictly consistent occurs when a user makes a change in one notation that can not be reflected in the other notation (syntax errors are a good example of this type of change), or when there is no natural one to one mapping between notations (such is found in Case tools). This problem is further discussed in subsection 2.5.3.

What are The Differences Between Two-Language Programming by Demonstration and Conventional Programming Environments?

We did not find any examples of a two-notation programming by demonstration environment that had the same Language Signature as a two-notation conventional style environment. PBD systems tended to have one notation that was used for writing and watching, but not for reading. Conventional style systems tended to have one notation that was used for watching, but not for reading and writing. Table 2.8 describes the differences in greater detail.

Although the difference in Language Signatures appears to be a property of the different programming styles allowed by each type of programming environment, we believe the two environments classes can be merged to create an environment that uses two notations for all tasks. For example, consider ToonTalk [89]. Both ToonTalk and its relation Pictorial Janus [92] are based on Janus [26], but ToonTalk is a PBD programming environment whereas Pictorial Janus is a more conventional style visual programming environment. Combining the two systems—letting people read and write Pictorial Janus code as well as interact with Robots in a video game style—would create an environment with a Language Signature [RE/WR/WA + RE/WR/WA], or a dual notation environment using a PBD interface to a conventional notation.

2.2.3 Three Notation Environments

Environments in this section use three notations for the three fundamental programming tasks. Although it should be possible to create a programming environment with a different notation for each task, all environments we reviewed used multiple notations for only one task. The task was reading or writing; never watching.

This section describes additional mechanisms to avoid the gulf of expression and representation. Previous sections found evidence that read-only representations risk creating a gulf of expression. This section provides evidence that programming environments using *transient* read-only representations do not risk creating this gulf. Previous sections also found evidence that providing multiple mechanisms to write programs (but only one way to read programs) could cause a gulf of representation: users build a cognitive model of their program while programming and then their program is represented to them in a different notation. This section provides evidence that support for users to map between different notations reduces the risk of creating a gulf of representation. A more detailed analysis of the gulfs is available in Section 2.3.

Environments with multiple notations for reading

This section describes programming environments that have multiple notations for reading and one notation for writing. These programming environments all risk creating a gulf of expression: with multiple notations for reading and one for writing they must have at least one read-only notation. Two environments in this category overcome this gulf by providing an transient read-only notation: the program is read through the computer's speakers. Unfortunately only informal studies have examined the effects of a transient notation.

We found three programming environments in this category: AgentSheets, Pecan, and Mondrian.

AgentSheets is a programming environment for visual simulations [151]. Users describe agent behaviour with an iconic language. The AgentSheets environment can also read a program through the computers speakers, and has a [RE/WR/WA$_{iconic}$ + RE$_{spoken}$ + WA$_{agents}$] Language Signature. Anecdotal evidence reveals that using multiple program representations helps users write and understand programs.[3] We believe that AgentSheets' spoken notation does not create a gulf of expression because the spoken representation is transient: users intuitively know they cannot edit a spoken representation. However, more research is needed

[3] Informal conversation with Alexander Reppenning

to confirm this belief and examine the effect of letting people write using a spoken notation (which would change AgentSheets' signature to [RE/WR/WA$_{iconic}$ + RE/WR$_{spoken}$ + WA$_{agents}$]).

Mondrian is a programming by example system for creating interactive drawings [106]. Users can program new commands into Mondrian by creating graphical rewrite rules. While users are creating these rules Mondrian reads the users' commands through the computer's speakers. Mondrian can also convert the user-defined rules into Lisp code.

Pecan produces a read-only Nassi-Shneiderman diagram from code [148]. Although we could not find any papers describing usability studies of Pecan, we argue that Pecan risks creating a gulf of expression because users might build a mental model of program behaviour based on the Nassi-Shneiderman diagram, and then want to edit the Nassi-Shneiderman diagram directly. This argument is supported from the analysis of [WR/WA + RE] environments: all representations should be editable (see Section 2.2.2).

Environments with multiple ways to write

Environments with multiple notations to write programs, but one way to read programs, risk creating a gulf of representation: users must map from the notation they used for writing to the notation they use for reading. We found one three-language environment with multiple ways to write a program: Leogo [27]. Leogo levers the gulf of representation to teach users to program and is pictured in Figure 2.9.

Leogo is an extension of Logo where users can program using three different notations: textual logo code, an iconic version of the code using buttons for commands and sliders for amounts, and a direct-manipulation notation where users can manipulate the turtle directly. Changes in one notation in Leogo are immediately reflected in the other two notations (see Figure 2.9). The rational behind using multiple notations was to help users lever their knowledge of how a turtle and iconic interfaces work to help children learn Logo. An evaluation of Leogo found that children could use Leogo, and that they tended to pick one notation and stay with that. Leogo's Language Signature is [WR/WA$_{turtle,\ lines}$ + WR/WA$_{iconic}$ + RE/WR/WA$_{text}$], and is shown in Figure 2.9.

One of Leogo's motivations was to help aid knowledge transfer from a notation children would be familiar with (the way the turtle moved) to a notation that provided more power (the

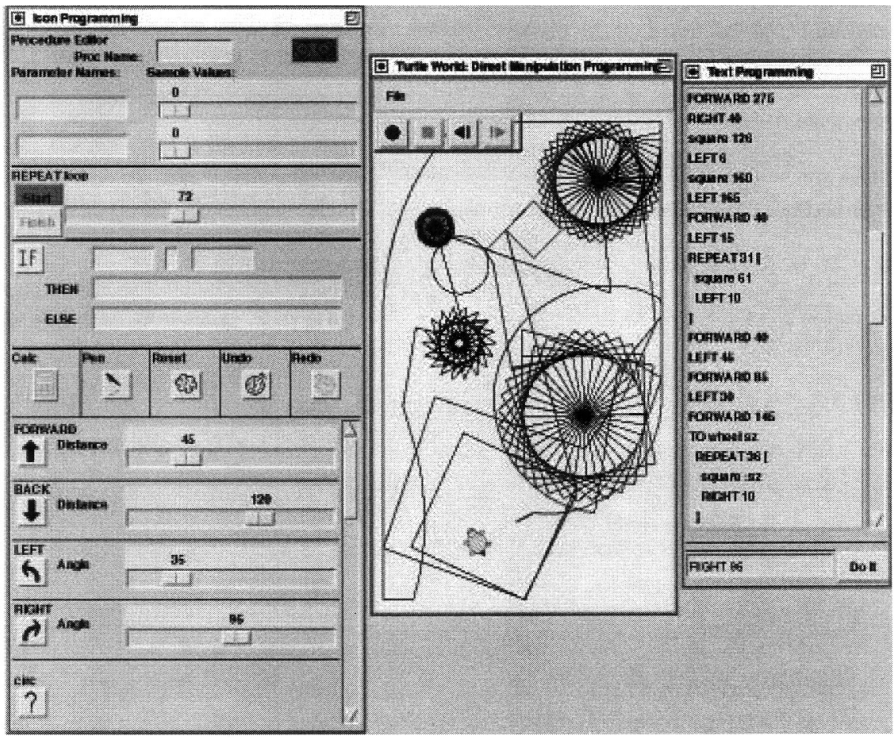


Figure 2.9: The Leogo programming environment. Users can manipulate any of three notations (textual, iconic, or directly-manipulate the turtle), and see the results immediately in the other two notations. For example, a user who wants to move the turtle forward 50 units can either drag the turtle forward 50 units and see the statement “FD 50” appear as well as the icon for forward movement depress and a slider advance to 50, or they can type “FD 50” and watch the changes in the iconic representation and the output domain.

textual notation). This motivation is similar to Prototype 2 (see Section 2.2.2). Both these environments provide multiple notations for reading and writing—one close to the task domain and one close to conventional code. The designers argue that this decision is useful as it can aid knowledge transfer from domain specific knowledge to knowledge about conventional code. Unfortunately the designers of the two environments did not perform an evaluation of the successes of the knowledge transfer.

2.2.4 *n*-Language Environments

The only environment we reviewed that supports an arbitrary number of notations is the Garden programming environment [149] (Garden is a successor of the Pecan programming environment, which is described in the previous section). Users of Garden write programs in one of several notations and can add new notations to the Garden environment. Reiss describes how a user can extend Garden so the user can program using petri-nets. Despite this expressive power of Garden, users must use the same notation to read their program as the one in which they wrote their program. This means that Garden provides n notations for writing but only one for reading—the notation that the user wrote their program in. We write Garden's Language Signature as [(WR+WR+. . .)/RE] rather than [RE/WR + RE/WR + . . .], as the latter conveys that users can move between the different notations after they had started writing code.

Unfortunately no user studies were performed on Garden, so we can neither analyse what the effects of user-extensible environments are on users nor examine what gulfs are created in a user-extensible programming environment.

2.2.5 Are There Any Missing Language Signatures?

As our taxonomy groups environments by how they use different notations for different tasks, it is natural to look for ways of using notations that are not implemented in any programming environments. We identified four unimplemented Language Signatures. An analysis of the signatures follow.

[RE/WR/WA + RE/WR/WA]: We imagine two feasible types of programming environment with this Language Signature. The first is a dual notation environment with watching support for both notations. The second combines a PBD environment and a conventional environment, and could create a gulf of representation if the notations are not kept consistent.

[RE + WR + WA]: These environments use different notations for each of the three fundamental tasks. Our review predicts that a [RE + WR + WA] environment risks creating: a gulf of expression because there is a read-only representation, a gulf of representation because there are different notations for reading and writing, and a gulf of visualisation because the representation is not animated at run-time. These gulfs would reduce the usability of a [RE + WR + WA] programming environment.

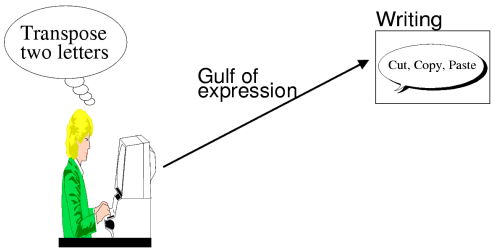
[RE/WR/WA + RE/WR/WA +WA] environments could be a dual notation environment in a task specific domain. Mulspren, the programming environment developed in this thesis (chapter 6), has a [RE/WR/WA + RE/WR/WA +WA] Language Signature.

[RE/WR/WA + RE/WR/WA + RE/WR/WA] could be an extension of the previous category to include programming by demonstration techniques. Our review indicates that environment developers should take care to keep all notations consistent at all times, and should provide mechanisms for users to move easily between the notations. It is also possible that an environment with three editable notations could create information overload for a user, and we suggest caution for any environment author considering creating an environment in this category.

2.3 Programming Gulfs

This section examines in depth the three cognitive programming gulfs: the gulf of expression, the gulf of representation, and the gulf of visualisation. To re-cap, the gulf of expression is the cognitive difference between a user's mental model of desired program behaviour and how they must express the program to the computer. The gulf of representation is the cognitive difference between a user's mental model of a program and how the program is represented to them for reading. The gulf of visualisation is the cognitive difference between a user's mental model of program behaviour and what the user sees when the program is executed. A visual representation of the gulfs is shown in Figure 2.6 (on page 20).

2.3.1 Gulf of Expression



A gulf of expression is created when programmers must translate their internal mental model of the desired program behaviour into a programming notation. The further the cognitive distance from the programmer's mental model of desired program behaviour to the notation they use to write the program, the greater the cognitive barrier programmers must overcome before they can write their program. Their internal model is influenced by several factors, some of which are artifacts of the programming environment and some of which are artifacts of the programmers themselves. We identified three factors that can affect a programmer's mental model of desired program behaviour: the program representation; the task domain; and the constraints imposed by the programming environment.

Factor: Task Domain

When an end-user is writing a program they begin with a model of desired program behaviour. For example, consider a user wanting to build a simulation of rabbits and wolves. Before they begin programming they have a model of how rabbits and wolves interact. They must then translate this model to the syntax and semantics of a programming notation. Unfortunately, designers of programming notations have no control over what types of programs users are going to use their environment to build, meaning that no matter how well a notation is designed, there will always be a gulf of expression for at least one user.

Much work has been done to reduce this gulf by moving programming environment notations to task specific domains, under the belief that the closer a notation is to a programmers internal model, the easier the notation will be to use. For example, Nardi argues that the main reason for the success of spreadsheet and CAD programming notations is the task domain [122]. Unfortunately, specialising a programming notation for a particular domain causes a trade-off between the level of abstraction of the notation's symbols and its potential generality. Conventional programming languages, such as C or Java, use abstract symbols for program expression, causing a large gulf between the notation used for writing (textual symbols) and the resultant program behaviour (possibly an animation or a graphical user interface), but the abstract symbols enable the language to program a wide range of different domains. In contrast, languages with a strong mapping between their symbols and the domain will normally be constrained to a

small set of domain-specific problems. For example, programming by demonstration systems, in which the input symbols are identical to the programming domain, are normally limited to a specific domain like creating user interfaces (see [138] for a discussion of domain independent programming by demonstration).

Other research examining the relationship between a programming notation and a user's mental model is Green and Petre's Cognitive Dimensions framework [65]. This framework introduces thirteen cognitive dimensions that can be used to analyse the usability of a programming environment. The dimension most relevant to the gulf of expression is *Closeness of Mapping*. This dimension represents the difference between a problem world and a domain world. Green and Petre argue that the closer the program world is to the problem world, the easier a programming notation will be to use. In our framework, we believe that *initially* the user's cognitive model of how to program it internalised in terms of the problem world. However, as a user spends more time solving a problem their cognitive model of their program will be affected by the programming representation, moving the gulf of expression away from the closeness of mapping dimension.

Factor: Environmental Constraints

Another factor influencing the gulf of expression is the extent to which a programming environment constrains (or determines) the types of expressions that the programmer can issue. Conventional programming languages typically under-determine the user by providing no constraints on the textual symbols that the user can enter. However, research systems, such as the Cornell Program Synthesizer [182], over-determine the programmer by requiring that statements be selected through the language's grammatical rules: the resultant programs are guaranteed to be syntactically correct, but the user is forced to work through possibly excessive constraints.

In 1995, Toleman and Welsh conducted a usability evaluation of structured editors [184]. They compared three different editors: a free form text editor, and a structured editor they designed, and the Cornell Program Synthesizer. Toleman and Welsh measured many variables including: task completion times, perceptions of the interface, and the preferences of their participants. Their participants were five final year honours students. Toleman and Welch found: while the participants preferred the Cornell editor, there was no reliable difference in speed or accuracy when using editors. Unfortunately, this study examined how expert programmers use structured editors: as experts are likely to know the syntax of a programming notation, we expect experts would make fewer errors than novices when modifying existing code. A structured editor

might help novices avoid errors more than it can help experts.

More recent work found constraints provided by structured editors do help novice programmers. The Alice programming environment uses drag-and-drop programming (to guarantee syntactical correctness) to overcome many problems beginner programmers have learning syntax [31]. In a similar vein, many commercial programming environments include type-ahead facilities that allow programmers to select from contextual information when available (such as the methods of an object of a particular class), while also allowing any symbols to be typed. Constraining the programmer by using type-ahead facilities or a syntax directed editor reduces the gulf of expression[4].

Factor: Program Representation

Our review found evidence that the program representation (the notation used for reading) affects the gulf of expression. The most compelling evidence for this unintuitive relationship, that the notation used for reading (the program representation) affects a programmer's internal mental model of program behaviour, comes from [WR/WA + RE] environments: two notation PBD environments with a read-only representation (subsection 2.2.2). This class of programming by demonstration environments provide a, typically textual, read-only representation of the generated program.

In an evaluation of Prototype-2, Gilligan found that "*The environment sorely needed unwind, delete, and editing facilities.*" [54] This is evidence that the notation used for reading can change a programmer's mental model more than the notation used for writing (otherwise Gilligan's users would have simply re-demonstrated their programs rather than wanted to edit the representation). Additionally, there is evidence that read-only notations help build internal models of other program notations, but only when the representation is transient.

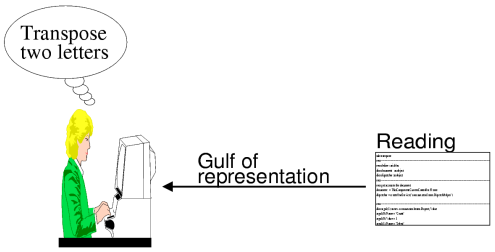
Reducing the Gulf of Expression

There are three mechanisms to reduce the gulf of expression, each related to a factor above. All mechanisms attempt to reduce the cognitive difference between the user's internal model of their program, and how they must express their program to the computer. First, and most important, programming environments should make every non-transient representation editable. Second, whenever possible, programming environments for end users should use symbols in the end-

[4] There is a correlation in an old saying: the hardest thing for a writer is a blank sheet of paper.

users' task domain. Finally, programming environments should provide a syntax directed editor to overcome problems learning syntax.

2.3.2 Gulf of Representation



The gulf of representation is the cognitive distance between the programmer's internal model of desired program behaviour and the notation that is presented to them for reading. Our review identified three ways this gulf can inhibit programmers: when there is no representation, when the representation used for writing is different to that used to reading, and when there are multiple notations for reading.

Factor: No Notation for Reading

Environments that do not provide a notation for reading create a gulf of representation: programmers do not want to execute their program as they do not understand what actions the program will perform when it is executed. This factor was introduced and described in Section 2.2.1. To reduce this factor, it is vital that programming environments show a representation of the program a programmer is writing.

Factor: Different Writing and Reading Notations

When a user writes their program using one notation and then reads their program using a different notation there is a risk of a gulf of representation. Users must map from the symbols they wrote the program in to the symbols they are now presented with they must overcome a cognitive gulf when performing the mapping. To overcome this problem, a programming environment should provide scaffolding to help a programmer understand the relationship between the notation they used to write the program and the notation they used to read the program.

Factor: Multiple Reading Notations

Subsection 2.3.1 describes how the notation used for reading affects a programmer's internal model of their program. When a programming environment provides multiple notations for reading it risks creating a gulf of representation. This gulf could confuse a programmer: they have to switch between multiple notations for reading and create a mental model encompassing all notations.

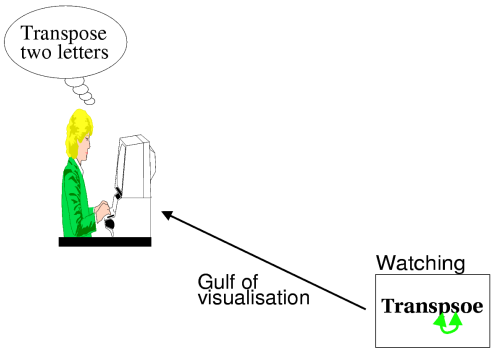
The gulf can be reduced by keeping the notations consistent at all times and providing support to transparently move from one notation to the other. This problem is exacerbated by the difficulty of browsing program code. Programs consist of interconnected procedures, rules, methods, and objects. A programmer must navigate through this rich hypertextual space to understand the program. While some work has been done solving this problem [29], providing support to help a programmer move easily between two complex, rich, and heavily interdependent hypertextual spaces is an area for future work.

Reducing the Gulf of Representation

There are three mechanisms to reduce the gulf of representation, each related to a factor above. First, programming environments must provide a notation for reading. Second, programming environments should provide support for users to move seamlessly between notations. Finally, these notations should be kept consistent at all times.

2.3.3 Gulf of Visualisation

A gulf of visualisation arises when a programmer has difficulty mapping between the observed behaviour of the running program and their internal model of their program. As the notation a programmer uses for reading heavily influences their internal model, we tend to use the program representation as a surrogate for a programmer's internal model. Programmers, therefore, must map between their program's observed behaviour and their internal model as encoded in the notation used for reading. The gulf of visualisation is not only a problem for novice programmers: it also causes problems for competent programmers who must use debuggers and other tools to overcome the difficulties of visualising the internal dynamic behaviour of the program.



There are two factors influencing the gulf of visualisation: understanding what the program is doing, and understanding how the environment is executing a program.

Factor: Understanding Program Behaviour

Researchers have designed many program animation tools to help programmers understand what their programs are actually doing [44,107,118,185]. These program animation (or visualisation) tools can help users understand problems in their code by creating a strong mapping between different notations.

Factor: Understanding Program Execution

This factor influences the gulf of visualisation when users do not understand how the programming environment is executing their program. For instance, Rader reported that children using StageCast had problems mapping between the agent-based representation of rules and their mental model of expected behaviour [145]. Within educational environments, overcoming the gulf of visualisation provides opportunities for scaffolding the student's understanding. Systems could allow learners to manipulate the visualised behaviour of the environment in a variety of ways, including changing and controlling the timing of execution of statements (for instance, stepping forwards and backwards through a series of instructions), and revealing internal structures that are not normally viewable (for instance, the state of the run-time stack). By providing controllable insights into the machine's state, we believe that the programming environment can encourage transfer effects between novice and more advanced programming concepts.

Reducing the Gulf of Visualisation

The gulf of visualisation can be reduced by improving the mapping between three elements: the visual display of the program's behaviour, the visibility of the state of the machine, and the representation of the program statements being executed. We hypothesise that by appropriately designing the visualisation capabilities of educational programming environments, it is possible to enhance understanding of programming concepts.

2.4 Competing Theories of Programming

This section examines several alternate theories of programming notation design and learning. The theories are: Cognitive Dimensions, ACT*, and DEFT.

2.4.1 Cognitive Dimensions

Cognitive Dimensions is a framework developed to analyse visual programming notations and the programming environments used to create, modify, and execute the notations [65]. The framework has thirteen dimensions, ranging from error-proneness (how easily can programmers make errors) to viscosity (how easily can programmers change an existing program). The framework was developed to create a set of heuristics that an expert can use to critique a programming notation. In this way, they are related to the heuristics used for ordinary user-interface evaluation [124], but are tightly focused on usability issues related to programming notations.

Cognitive Dimensions and Language Signatures examine different and complimentary aspects of notation use in programming environments. Whereas Cognitive Dimensions provides heuristics to examine the usability of a particular notation, Language Signatures provide heuristics to examine how multiple notations in an environment notations interact with each other and with the user. This difference in scope of the two frameworks means that both are useful when designing and building programming environments.

While the Cognitive Dimensions framework was created with the goal of analysing programming notations, some of the dimensions can be extended to analyse the the relationships between programming notations.

Abstraction Gradient. The Abstraction Gradient dimensions analyses the minimum and maximum levels of abstraction, and looks at the ability for a programmer to abstract fragments of a program. An extension of this dimension to multiple notations would examine the relationships between the abstraction gradient of each notation, and look what happens to one notation when a programmer abstracts part of the program in the other notation.

For example, consider a programmer who is working in a multiple notation programming environment and wants to create a new abstraction by refactoring some common code into a method. Conventionally, the abstraction gradient dimension would examine how much work the programmer must do to perform the refactoring. However, in a multiple notation programming environments, a environment designer must also consider the effects of the refactoring on the other notations.

Closeness of Mapping. This dimension examines the mapping between the problem world and the syntax and semantics of the programming notation. The extension for multiple notations also examines the closeness of mapping between the multiple notation: how much

cognitive effort a user must expend when switching, or moving between, notations.

Error-Proneness. This dimension examines how easy it is to make an error, and more importantly how easy it is to recover from an error. An extension of this dimension into a multiple notation system would examine the effects of making an error in one notation on the other notation.

For example, consider a user who makes an error in one notation of a multiple notation programming environment. An analysis of the Error-Proneness dimension in a multiple notation environment should consider questions including examining the effects of the error on the other notations, and the recoverability from the error using the other notations.

Hard Mental Questions. Multiple notation programming environments can create many additional cognitive tasks that users must overcome to be able to use the environment. These were enumerated by Ainsworth *et al* and include: understanding the relationships between representations and the domain, translating between representations, and, if designing representations, selecting and constructing an appropriate representation [3]. A cognitive dimensions analysis of multiple notation programming environments should include an analysis of how hard these tasks are for a user to perform.

Role Expressiveness. This dimension refers to the ease in which programs can be read (as opposed to Hard Mental Questions or Closeness of Mapping which refer to the ease in which a program can be written). In a multiple notation programming environment, users of this dimension should examine how easy it is for people to understand the relationship between the notations *as well as* read the individual notations.

Secondary Notation and Escape From Formalism. A secondary notation refers to extra information that is not part of the actual program: commenting and indentation. Green and Petre argue that support for secondary notation is important for programming notations. In a multiple notation programming environment, a programmer using this dimension to analyse the notations should consider the effects of modifying a notation on the comments and layout of the other notations.

Viscosity. Viscosity refers to a notation's resistance to local change, or to the ease in which a programmer can make small changes to a program. In a multiple notation programming

environment, small changes in one notation could lead to large changes in another notation. This high inter-notation viscosity is especially likely if the two notations use very different representations of the program. For example, consider a multiple notation programming environment with two representations of a program: a control flow representation and a data flow representation. A simple change in the data flow representation could equate to a large change in the control flow representation. Programmers analysing multiple notation programming environments should consider the effects of small changes in one notation on the other notation.

2.4.2 ACT*: Adaptive Control of Thought

ACT* is a theory of cognition: a theory of how the brain stores knowledge, retrieves knowledge, and learns new knowledge [5]. It is an interesting theory as it has been successfully applied to help understand how people learn to program computers [6].

Skill acquisition in ACT* is a simple sequential process. First, learners have knowledge in declarative form. For example, consider a learner programmer who knows that the statement $i = i + 1$ increments the value of i by one. At this stage of learning, the learner will use analogy to transform the concrete examples to generate new code—code to add 2 to i, or to increment k by one. In the second stage of learning, learners create a generalised form of this concrete rule, perhaps incorporating other rules as well. This generalised form is called a production rule. An example of a production rule is:

```
IF the goal is to increment a variable (X) by
   a certain quantity (q)
THEN enter X+=q;
```

After generating production rules, a learner's rules gain strength through practice. Ultimately a complex skill, such as programming, may consist of several hundred production rules and the act of programming requires sequential application of these rules.

One of the more interesting effects of this theory is that learning is not a function of how a student solves computer programming problems, or how much help a student has solving a problem. Learning is a function of how many problems they solve and how many times they see examples of particular production rules [6]. More precisely, Anderson writes:

> *The data... illustrate that amount of practice on specific productions is a strong determinant of performance.* [6]

This theory provides additional arguments about how multiple notations might help or hinder learner programmers.

Using different notations for different activities can hinder users

Consider a programming environment that uses different notations for reading and writing. In this environment, the rules learners learn by reading programs are no use for writing programs. Learners must generalise two sets of production rules: one for reading and one for writing. There are two reasons why learners of multiple notation environments will take longer to learn to program than learners of single notation programming environments. First, the actual generation of rules will take longer — there are more rules to generate. Second, the rules will take longer to become strong — users with two sets of rules will get less practice with each rule than a user learning only one set of rules.

Using different notations for the same task might help users transfer knowledge from one notation to the other

Consider a programming environment where users use two different notations for reading. As users interact with the environment, they will generate production rules for one of the notations but not the other notation (Cockburn and Bryant's research indicates that users of multiple notation programming environments tend to stick with the notation they first used [27]).

We conceive two situations where transferring the production rules from one notation to the other could be useful. First, if the two notations are sufficiently similar, the two notations might help generalise production rules earlier than a single notation environment. Second, if the two notations are sufficiently different, forcing a user to use the notation they have not generated production rules for, learner programmers might generalise more abstract production rules that help them learn new notations faster. However, we believe that a programming environment would need to provide much scaffolding to help a learner programmer generalise these abstract production rules.

Using similar (but different) notations for the same task might increase learning speed

As mentioned previously, the amount of practice learners have with specific productions (an instance of a production rule) is related to how well they can write programs:

The data... illustrate that amount of practice on specific productions is a strong determinant of performance. [6]

Compare a programming environment which provides multiple notations, where the two notations are very similar, with a single notation programming environment. Learners using the multiple notation environment will get at least as much contact with specific productions, and in best case, twice as much contact with specific productions. The increased contact with specific production rules will increase learning performance.

2.4.3 DEFT

Some environments let users perform one or more of the three fundamental tasks in multiple ways: they use semantic redundancy within a task. Sample environments include Agentsheets and Leogo. Our review describes how using multiple notations for reading, writing, and watching can aid program comprehension. Using multiple notations for writing can: avoid a gulf of expression (subsection 2.2.2), help people learn to program textually (subsection 2.2.3), and let users choose a notation that suits the problem they are trying to solve (subsection 2.2.4). Using multiple notations for reading can help users lever domain knowledge to learn to program (subsection 2.2.3). Many programming environments use multiple notations for watching. They provide a program visualisation component that helps users map between program code and behaviour, overcoming the gulf of visualisation.

There is also much research from the education community that examines how multiple notations can help people learn about problems and solve problems. One framework that examines how multiple representations are used in computer based learning applications is DEFT (DEsign, Functions, Tasks) [3]. This framework examines issues surrounding three aspects of multiple representation environments: the functions of multiple representations, the design parameters that environment designers must consider when designing multiple representational environments, and the additional cognitive tasks that users of multiple representation environments must perform. DEFT classifies different types of multiple representation learning environments based on the pedagogical purpose of using the multiple re[presentations (shown in Figure 2.10). The decomposition on pedagogical purpose is quite different from our taxonomy, which classifies multiple notation programming environments based on how notations are used in programming environments.

While DEFT provides solid guidelines about the use of multiple representations, and includes

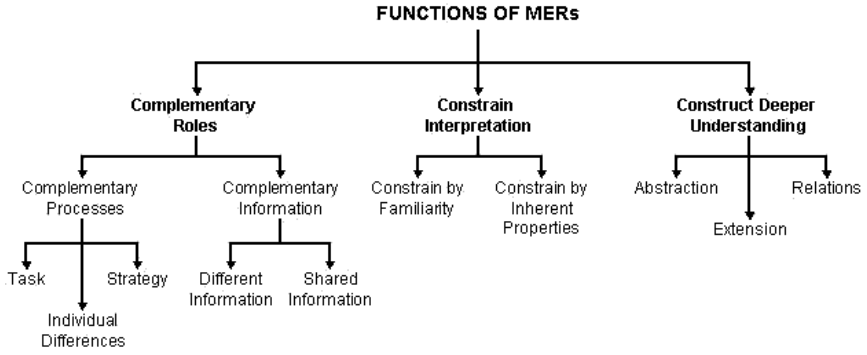


Figure 2.10: A decomposition of the purposes of multiple notation learning environments. This decomposition is used in the DEFT framework for understanding how to build multiple representation learning environments [3]. This picture was taken from [2].

a comprehensive list of the possible uses for multiple representations, our framework covers two areas that DEFT does not. First, DEFT can not let us reason about representations that are not displayed to a user. Unfortunately, systems including Prototype 2, Eager, and Gamut, use a notation for writing that is not displayed. Second, while DEFT helps us reason about visualisations of a problem and how users interact with multiple visualisations of a problem, DEFT does not help us reason about how users might create and modify these problems using multiple representations.

In our taxonomy, DEFT helps us reason about environments where different notations are used for the same task (i.e.: [WA + WA] or [RE + RE] environments). DEFT does not cover the effects of multiple notations for different tasks.

2.5 Additional Multiple Notation Systems

There are several other systems that use multiple representations. This section examines them and describes how they are useful. Additionally, we write Language Signatures for the systems and use the information from the previous sections to analyse the environments. In doing this we provide evidence that our framework is useful in domains that are not end-user programming environments.

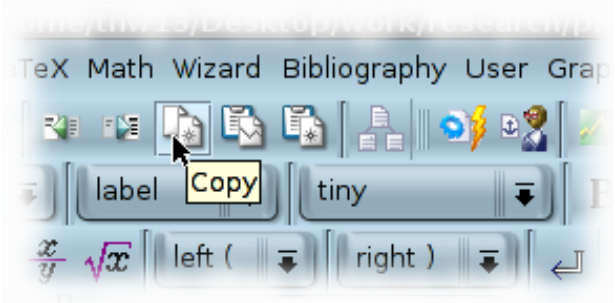


Figure 2.11: Example tool-tip. The tool-tip appears below and to the right of the mouse cursor.

2.5.1 Tooltips

Tooltips are a lightweight help mechanism [157]. When a user moves their mouse over an interface widget and pauses a small unornamented window appears near the mouse cursor. The unornamented window is called a tool-tip, and typically holds a textual description of what will happen if a user clicks on the widget. Tool-tips first appeared (as Balloon Help™) in Apple System 7.0 [157] and are also known as bubble help and pop-up hints. While there is little formal evaluation of balloon help, anecdotal evidence suggests that they are helpful when learning how to use an application and that users should be able to disable (and re-enable) the tips easily.

We can write a Language Signature to describe tooltips. They provide two notations for reading: the icon and the English description, and one notation for writing: users can click on and, in some environments, modify the look and feel of the icon. Their Language Signature is [RE/WR$_{iconic}$ + RE$_{English\ text}$]. This Language Signature suggests a gulf of expression (users will want to modify the tool-tip as well as the look and feel of the icon) which is avoided due to the transient nature of tool-tips (see subsection 2.3.1).

2.5.2 Literate programming

Literate programming (LP) is a software development methodology proposed by Knuth when he wrote the LaTeX typesetting environment [97]. Proponents of LP argue that a good program should read like a good book and that people reading programs should not have to examine the code to understand the program. This need for a program to read like a book requires that users create, modify, and browse two heavily related representations: the program code and the program comments. However, unlike multiple notation programming environments, where each

program representation is a view of an abstract program and the computer keeps the two consistent, the two LP representations must be kept consistent by a human. Fortunately, proponents of LP have created a host of tools to help users track the parallel representations and keep the two representations consistent; unfortunately they have performed few usability studies examining the effectiveness of their tools, the LP methodology, or even the effectiveness of providing multiple representations for program understanding [93].

Like tool-tips, we can write a Language Signature for LP. They provide two notations for reading and writing—conventional code and English comments. Both notations are editable, meaning LP tools have a [RE/WR$_{computer\ syntax}$ + RE/WR$_{English\ text}$] — the same signature as a visual interface to a textual language. Our analysis of Language Signatures and programming gulfs, presented in Section 2.3, suggests that literate programming tools could suffer a gulf of representation, and that the tools should provide support to help users switch notations and provide support to keep the notations consistent at all times.

2.5.3 Other Development Environments

Some new development environments also provide multiple notations. Two examples are Together[5] and Dreamweaver[6]. Programmers using Together manipulate class diagrams or conventional code[7] and see changes in the other representation immediately. Dreamweaver users can view both HTML source and the visual rendering of a web page. They can modify either representation and see changes immediately reflected in the other representation. Although there is little research examining how users interact with these highly complex multiple language software development environments, software engineers would only pay thousands of dollars for a single Together license if the software was useful and usable. Likewise, Dreamweaver users informally report that the multiple notation aspect of Dreamweaver is useful, only this aspect is very processor intensive.[8]

Many CASE (Computer Aided Software Engineering) tools provide multiple representations or visualisations of a program. Problems with these tools have been well discussed in the literature (for examples, see [48, 205]). Additionally, Paige, Ostroff, and Brooke write:

> *It is especially challenging and relevant when using the modelling language UML [13],*

[5] http://www.borland.com/together/
[6] http://www.macromedia.com/
[7] Together supports Java, C++, C♯, Microsoft Visual Basic 6, Visual Basic .NET, and IDL.
[8] Informal conversations with Dreamweaver users.

where five different and potentially conflicting views of a software system of interest can be independently constructed... The different descriptions must at some point be combined to form a consistent single model of the system that can be used to produce executable program code. The process of combining the descriptions should identify inconsistencies... [133].

To overcome these problems, we recommend keeping notations (or views) as consistent as possible at all times. Unfortunately, there might be cases where small changes in one notation create large changes in another notation. This could create a large gulf of representation for a user: a user might not understand why all the changed in one notation (or visualisation) were necessary.

2.5.4 Program Animation Tools

There has been much research into program animations to help beginner programmers understand varied aspects of programming. The aspects include: understanding what the code does, understanding how the code is being executed, and understanding a complex algorithm (like quick sort or binary search). Additionally, many researchers have built tools to help build complex program animations [44, 118]. These animations present, at a minimum, a static program representation that is animated as a program executes. Depending on the animation, the animation might also include a visual view of the data the program is working on or a visual representation of the state of the machine that is executing the code (the stack, registers, etc). These alternate representations are only shown as the program is being animated.

A Language Signature for program animations is [RE/WA$_{computer\ program}$ + WA$^{*}_{animations}$]. Our analysis of Language Signatures and programming gulfs, presented in Section 2.3, suggests that there will be a gulf of expression if users of program animations will want to edit the code (possibly to see what will happen differently), and that the different visualisations should be kept consistent to avoid creating a gulf of visualisation.

2.6 Summary

This chapter used a powerful description of how programming notations are used for different activities in programming environments. We used this description to classify to review programming environments. The description is called a Language Signature. Using this description we

grouped programming environments based on how they *used* notations rather than the *type* of notation they used. The analysis of programming environments based on how notations are used provides several useful insights:

↪ The majority of programming environments use more than one notation. Using more than one programming notation creates the risk of creating one of several cognitive programming gulfs. These cognitive programming gulfs can manifest themselves as usability problems.

↪ While some of these usability problems may seem to be self-evident (a good example is "*programming environments must provide a program representation*"), the fact that these usability problems are present in current programming environments means that designers of programming environments are not considering the effects of notations on users. The designers should consider these effects (and the chapter provides concrete advice for designers to avoid creating the cognitive gulfs).

↪ Several programming environments have attempted to use multiple notations as an aid to teach programming by using one notation to aid knowledge transfer to another notation (for example, Pascal or Logo). Unfortunately, the benefits of the transfer have not been empirically validated.

After describing Language Signatures, and how the use of notations can create usability problems, we examined three other programming theories: Cognitive Dimensions, ACT*, and DEFT. We found that these theories peacefully co-exist with Language Signatures, and we extended the Cognitive Dimensions framework to help analyse multiple notation programming environments.

Finally, we used Language Signatures to analyse other multiple notation systems, providing some evidence that this tool for analysing programming environments can also be used to analyse other types of environments. We believe that Language Signatures can be used to analyse any environment that contains an executable description of a dynamic process. These environments range from end-user programming environments to movies, cooking recipes, cartoons, and even configuring tools (like editing the configuration file of a web server).

While performing the review presented in this chapter, we noticed that there exist very few user-evaluations of the effects of interacting with multiple computer programming notations. The next chapter attempts to empirically validate the benefits of multiple notations.

Chapter 3

Multiple Notation Evaluation

The previous chapter classified, analysed, and reviewed many programming environments. One of the conclusions from the chapter was that multiple notations have potential to help people transfer knowledge from one domain to another and learn to program, but there was little evaluation of how well the transfer works. There was also little evaluation on the performance differences between different notations.

This chapter presents an evaluation of the use of multiple notations for *reading* computer programs. We are interested in two aspects of this task. First, we want to determine which of two different programming notations, or a combination of both notations, better helps users understand computer programs. Second, we want to determine if multiple notations can help people transfer knowledge from one domain to another. The particular type of transfer we are interested in is transfer from a domain that most people have knowledge of, natural language, to knowledge about a more conventional programming notation. For the natural lan-

guage, we used English, and for the conventional programming notation, we used a notation with a syntax similar to C++ or Java. We wanted to test transfer to a conventional notation because, despite the best efforts of programming environment researchers, the notations that many end-users have access to are conventional-style notations: the notation used for macro programming in their word processor or the mathematical notation in their spreadsheet.

To structure the two notations, we used research by Pane and Myers [136]. This research examined the structure and language used when non-programmers describe program behaviour. It found that the participants describe behaviour using an event driven paradigm and non-programmers use lists of objects with implicit iteration over the lists (with keywords *any*, *all*, and *the* to select the lists).

Our implementation of the multiple notation interface used a transient notation that was shown in a tooltip. Each tooltip provided a different version of one line of code. This approach is easy to integrate with other programming environments as it requires no additional screen real estate, and could also be used when browsing literate programs by adding tool-tips that contain the documentation associated with program regions [97].

The evaluation produced an interesting result: children (aged about 11) could read and understand the conventional-style notation faster than they could understand code written in English. There was no reliable difference in accuracy. The evaluation also found that most children preferred the English-like notation. These results provide motivation for multiple notation programming environments: we can provide users with access to both an efficient notation and a notation they prefer.

3.1 Motivation

There are two motivations for our experimental design. First, to control contributing factors, we only evaluate one task — the reading task. Second, we want to examine how well children can gain an understanding of conventional-style syntax when using different (or multiple) notations.

3.1.1 Evaluating Reading

The previous chapter found that many programming environments used multiple notations for different tasks and some programming environments used multiple notations for the same task. It also found that there was considerable potential to help programming by using multiple notations

for the same task. To reduce experimental variance, we choose to evaluate one of our three fundamental programming activities: reading. Our argument to evaluate reading follows:

1. Modifying a program is a more important activity than writing a new program.

MacLean *et al* performed a decomposition of the types of programming activities end-users perform [109]. They ordered these activities from easiest to hardest and argued that users will start programming at the easiest level and gradually move up the scale (see Figure 1.2 on page 5). In their taxonomy, program modification occurs before program creation, meaning that users cannot create programs unless they can successfully modify a program.

2. If users want to modify a program they must understand the programming notation.

Modifying a program risks creating syntactical and semantic errors. If a user does not understand a programming notation, they will not have the skills to fix their errors. In our taxonomy, there are two activities that aid program understanding: reading and watching.

3. Reading is a more interesting research area for program understanding than watching.

While little research has examined the effects of users reading multiple notations, much research has examined the effects on users of multiple watching notations: many program visualisation systems show program code and program output and provide mechanisms to help users link the two [44, 118].

4. The writing and watching tasks contain elements of the reading task.

When users write code they continually read and revise their code to find and fix syntactic and semantic errors. Similarly, when users are watching an animation of their program they must read the code that is being animated. By evaluating reading first we provide a baseline for the reading task that we can later use when evaluating the writing and watching tasks.

3.1.2 Teaching Conventional Code

We believe that people who to have an understanding of conventional-style code can achieve greater levels of efficiency because they are often confronted with notations designed for a computer rather than a user. They might encounter these notations after demonstrating a macro to

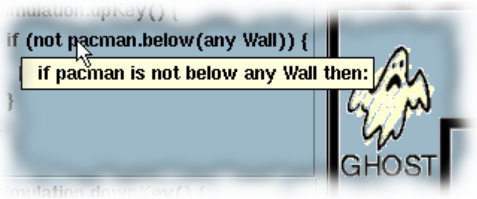


Figure 3.1: Multiple notation interface using tool-tips to show the English-like representation.

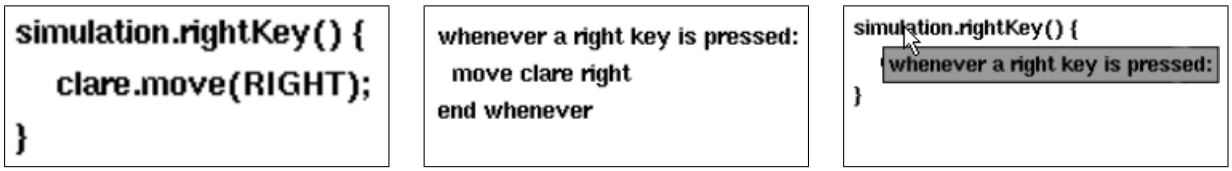
their word processor, writing formula in spreadsheets, or even viewing error messages returned by an email system. If users understand what the notation means then they can perform tasks ranging from understanding what went wrong to modifying and debugging their program.

We hypothesise that using multiple notations increases user's understanding of conventional notation because it will aid knowledge transfer from knowledge about English to knowledge about conventional code. This evaluation tests this hypothesis by dividing users into several groups and asking them several questions about static representations of computer programs. Each group is trained using one of three computer interfaces, and they are all tested on the same interface. The three training interfaces show users conventional-style versions of the computer program, English-like versions of the computer program, and both notations at the same time. In this way we can equitably compare the effect of different notations on understanding of conventional code.

We compared two textual notations that had a 1:1 semantic mapping. We made this decision for three reasons. First, using notations with a simple mapping should reduce cognitive load on a participant when mapping between the notations. Second, we wanted to investigate transfer effects to a conventional textual notation. Third, we wanted to use notations that were very similar so that production rules generated from one notation would be applicable to the other notation (Section 2.4.2 contains a discussion of production rules and programming notations).

3.2 Experimental Description

Each of the forty-six participants, aged ten and eleven, was asked a total of twenty questions about four different simulations. The evaluation was split into two phases. Fifteen questions were in a *training phase* and five questions in a *testing phase*. The second phase was shorter to measure the immediate benefits of transfer rather than how easy the initial notation made it for



(a) Notation used in the Conventional Condition.

(b) Notation used in the English Condition.

(c) Notations used in the Multiple Condition.

Figure 3.2: Screen snapshots of notations used in the three conditions. The notation used in the Multiple condition is the notation used in the Conventional condition with the notation used in the English condition available in a tool-top.

participants to learn the conventional notation. Participants in the conventional condition used the same notation in each phase to act as a control group.

In the the training phase participants in different conditions used different computer interfaces whereas in the testing phase all participants used the same interface. The interface used in the training phase in the first *Conventional* condition has the code is represented in a conventional-style syntax. In the *English* condition, participants answered questions about the same simulations, but the code in the training phase was represented using an English-like syntax. The third *Multiple* training interface combined the convention and English notations by placing English-like representation in tool-tips (see Figures 3.1 and 3.2). The Conventional and English conditions act as controls, ensuring that effects seen in the Multiple condition are due to users having access to both representations.

The notation used in the English condition was based on the notation described by Pane, Ratanamahatana, and Myers when they asked children to describe behaviour of particular visual simulations [136]. The conventional notation was created by taking the features of the English notation and making a notation similar to a conventional notation, using parenthesis, brackets, and method calls, but keeping semantic structures like the event driven paradigm and accessing agents using lists.

Fifteen participants were assigned randomly to each condition (using equal numbers of participants in each condition from each of the three school involved in the evaluation), unfortunately one participant in the Conventional condition elected to stop her involvement in the evaluation. Her data were discarded and another participant was assigned to her condition.

While the semantics of the two notations are identical, and are similar to the notation used in

Conventional-style	English-like

`any PacMan.contactWith(any PowerPill) {`
`  the PacMan.power = 10;`
`  the PowerPill.remove();`
`}`
	`whenever any PacMan touches any PowerPill:`
	`  set the PacMan's power to 10`
	`  remove the PowerPill`
	`end whenever`

Table 3.1: Example procedure in conventional-style and English-like notations

Hands [134], the syntax of the two notations differ. An example program with question is shown in Table 3.1. All the sample programs we used are listed in Appendix A.1. To ensure the English-like and conventional-style notations were consistent we used Lex (a scanner generator) and Yacc (a parser generator) to extend the context-free grammar for the conventional-style notation and produce a program that translates any program written in the conventional-style notation into the English-like notation. This transformation maps one statement in the conventional-style notation to one statement in the English-like notation.

After answering fifteen questions about three visual simulations in the training phase, all subjects, regardless of their training condition, moved to the testing phase where they answered five additional questions about a different visual simulation. The interface used in the testing phase was the same as the interface used in the conventional condition of the training phase. The testing phase was shorter in order to control the amount that participants learned during the testing phase. Switching from the three interfaces used for training to the one interface used for testing allows us to equitably compare how successfully the participants learned conventional code based on the interface they used in the training phase.

3.2.1 Hypotheses

We have several hypotheses for this experiment. The first set of hypotheses (H1a and H1b) examine program understanding in the first phase—we predict that participants in the English and Multiple conditions will complete tasks faster and more accurately than participants in the Conventional condition.

H1a: In the first phase, children in the English condition will complete tasks faster and more accurately than participants in the Conventional condition. We predict this hypothesis will hold as children have experience with English and not with conventional programming

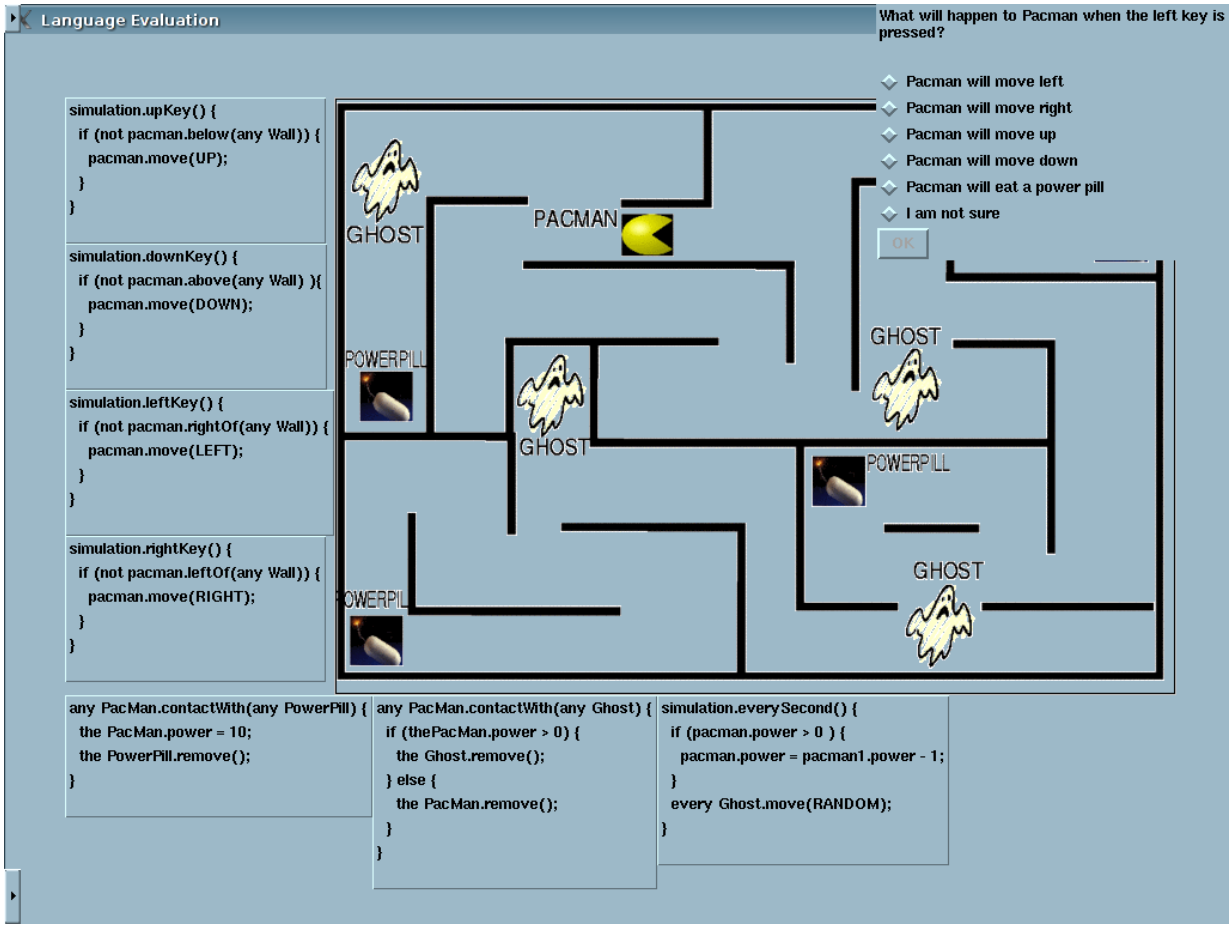


Figure 3.3: Single Notation Interface

syntax. This hypothesis is backed up by research indicating that people naturally express programs in English [22, 114, 136].

H1b: In the first phase, children in the Multiple condition will complete tasks faster and more accurately than participants in the Conventional condition. The participants will be faster because the interface is providing an English translation of the conventional code.

The second set of hypotheses (H2a and H2b) examine how well participants learn conventional syntax. They predict that participants in the Multiple and Conventional conditions will complete tasks faster and more accurately than participants in the English condition.

H2a: In the second phase, participants in the Conventional condition will complete tasks faster and more accurately than participants in the English condition. In the second phase participants in the English condition encounter conventional code for the first time. We predict that the new syntax will hinder participants' speed at understanding computer programs.

H2b: In the second phase, participants in Multiple condition will complete tasks faster and more accurately than participants in the English condition. We predict that the participants will be faster and more accurate because they have interacted with conventional code before.

When examined together, these two sets of hypotheses (H1 and H2) predict that the multiple condition has the advantages of both the Conventional and English conditions.

A further hypothesis, H2c, predicts that in the second phase, participants in the Multiple condition will complete tasks faster and more accurately than the other two conditions. If correct, this hypothesis provides strong motivation for adding multiple language support to programming environments.

H2c: Participants in Multiple condition will perform the second phase faster and more accurately than participants in the Conventional condition. Participants in both conditions have interacted with the conventional notation as much as each other, but participants in the multiple condition have had more help understanding the conventional notation.

3.2.2 Subject Details

The experiment was conducted at three primary schools in New Zealand chosen to have students from a similar socio-economic background. We asked the teachers to select children who were

in the middle of their class based on math scores. The evaluation was run at the participant's schools in a quiet room. Participants were run sequentially. The forty-six participants were allocated to one of the three training conditions, giving fifteen children per group, with gender balanced between conditions. One child in the conventional condition ended her participation after answering seven of the twenty questions and her data were discarded. Approximately one third of participants from each school were allocated to each training condition.

Approximately five minutes at the start of each evaluation was spent explaining the concepts behind visual simulations and the experimental interface to the participants. Each participant's involvement in the experiment lasted approximately twenty minutes.

3.2.3 Procedure

We designed four static visual simulations of comparable complexity (see Appendix A.1 for the simulations). Each simulation had seven rules and five questions. Each question could be answered using information from one rule. To answer the questions, participants were asked to predict program behaviour based on a static representation and a visual arrangement of agents. One of the questions is displayed in Figure 3.3. The remainder of the questions are in appendix A.2.

The first three simulations (those used in the training phase) are similar to each other but are not isomorphic. In each of those simulations, the rules control an agent that moves around the simulation and interacts with other agents. For example, the first simulation is a Pacman simulation where a Pacman moves around, eats power pills, and encounters ghosts.

The fourth simulation (used in the testing phase) was different. In the fourth simulation rules controlled how a ship moved along the bottom of the screen and shot aliens. This ship could move off one side of the screen and come back in the other side of the screen. This simulation was made different to better analyse transfer effects from the notations used in the first phase instead of transfer effects based on the type of simulation used in the first phase.

After each phase, participants were asked to rate two statements on the Likert scale. The two statements were: *I was confident with my answers* and *it was easy to complete the tasks*.

3.2.4 Apparatus

The experimental interfaces were implemented in Python/Tkinter [64]. Participants used a IBM R50 laptop running Debian GNU/Linux with a 15" LCD screen and a USB optical mouse. The screen's resolution was 1024×768. The interface logged: which questions each user answered,

Training Phase			
Interface	Conventional	English-like	Multiple
Training Phase Participants	s1–s15	s16–s30	s31–s45
Testing Phase Participants	s1–s45		

Table 3.2: Experimental Design. There were two factors: condition (interface used in the training phase) and phase (training or testing).

Speed	Accuracy	
	Low	High
Low	Lack of confidence; weak and incorrect mental model	Understand domain well; weak but correct mental model
High	Strong but incorrect mental model; participant not paying attention; over confidence; guessing	Strong and correct mental model

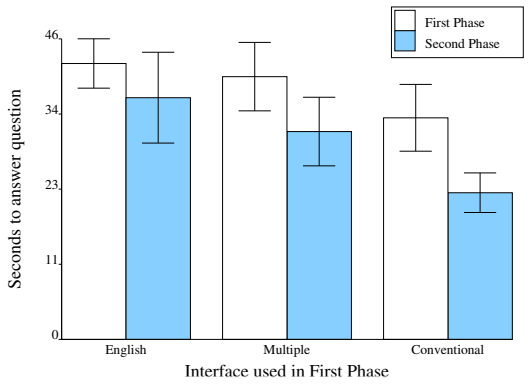
Table 3.3: Relationships between the dependent variables: time to answer a question, and the percentage of questions answered correctly.

the response to each question, and the time to answer each question.

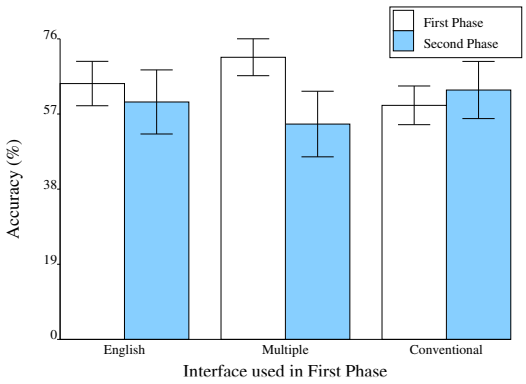
3.2.5 Data Analysis

The experiment was a analysis of variance (ANOVA) for factors training condition and phase. Training condition was a between-subjects factor with three levels: *Conventional*, *English*, and *Multiple*. Phase was a within-subjects factor with two levels: *training* and *testing*. Table 3.2 shows the design. This analysis was repeated for two dependent variables: time to answer a question and percentage of questions correct.

The dependent variables provide insight into how participants are answering the questions. For example, if a participant is taking a short time to answer the questions we can infer that they either have a strong mental model of what the code means, are just guessing, or are not paying attention. A long time to answer questions could imply that participants have a weak mental model of the code: participants are having to spend time reasoning about the code. It could also mean that participants have a lack of confidence in their skills and are continually changing their mind. We can further determine how participants are answering questions by examining their accuracy. These relationships are shown in Table 3.3



(a) Time



(b) Accuracy

Figure 3.4: Average time and accuracy taken to answer questions in the first and second phases. Error bars show standard error. There are reliable effects of both factors for the dependent variable time (phase: $_{F(1,42)}$ =21.5, p≤0.01, condition: $F_{(2,42)}$ =3.60, p≤0.05), but no interaction ($F_{(2,42)}$ =0.47, p=0.63). There are no reliable effects of either factor for the dependent variable accuracy (phase: $F_{(1,42)}$ =1.608, p=0.212, condition: $F_{(2,42)}$ =0.030, p=0.971), and no interaction ($F_{(2,42)}$ =1.682, p=0.198).

We log transformed the time data to stabilise the variance [46]. Unfortunately we could not log-transform the accuracy data: six participants spread among conditions answered all questions incorrectly in the testing phase (we could not log transform the data because the log of zero is undefined). In the case of the time data, all means and standard deviations we report are from the raw data, whereas F and p values are taken from the log-transformed data.

3.3 Results

3.3.1 Time

Anova showed reliable effects for both factors.

Phase: Participants in the first phase were slower (38.5 seconds) than participants in the second phase (30.2 seconds, $F_{(1,42)}$ =21.5, p≤0.01).

Condition: Participants needed a mean of 28.0 seconds to answer a question for the Conventional condition, 35.7 seconds in the Multiple condition, and 39.3 seconds in the English

	Training Phase	Testing Phase
English	64 σ5.6	60 σ8.1
Multiple	71 σ4.6	54 σ8.2
Conventional	59 σ4.9	63 σ7.2

Table 3.4: Mean accuracies and standard errors of the three conditions in both phases. Neither factor produced a reliable effect in Anova nor was there an interaction. The values are graphed in Figure 3.4b.

condition. The difference is reliable ($F_{(2,42)}$ =3.60, p$\leq$0.05). Post-hoc tests showed the difference in time was reliable between the English and Conventional conditions in both the first phase ($F_{(1,28)}$ =9.7,p$\leq$0.01) and in the second phase ($F_{(1,28)}$ =5.5,p$\leq$0.05). The differences between the Multiple condition and either the Conventional or the English conditions were not reliable in either phase.

There was no reliable interaction ($F_{(2,42)}$ =0.47, p=0.63). The means are displayed in Figure 3.4a.

3.3.2 Accuracy

Anova detected neither reliable effects of either factor for the dependent variable accuracy (phase: $F_{(1,42)}$ =1.608, p=0.212, condition: $F_{(2,42)}$ =0.030, p=0.971), nor any interaction ($F_{(2,42)}$ =1.682, p=0.198). Mean and standard error values can be found in Figure 3.4b.

3.3.3 Likert Questions

After each phase, participants were asked to rate two statements on the Likert scale. The statements were: *I was confident with my answers* and *it was easy to complete the tasks*. Likert-scale ratings for the three interfaces were reliably different only for the first question in phase two (Kruskal-Wallis test corrected for ties, H=6.55, df=2, N1=N2=N3=15, p$\leq$0.05): participants in the Conventional condition were very confident with their answers (median: 1), participants in the Multiple condition were confident (median: 2) and participants in the English condition weren't sure if they were confident or not (median: 3). All other medians are shown in Table 3.5.

I was confident with my answers

Condition	Training Phase	Testing Phase
English	2	2
Multiple	2	3
Conventional	2	1

It was easy to complete the tasks

Condition	Training Phase	Testing Phase
English	3	2
Multiple	3	3
Conventional	3	2

Table 3.5: Medians for Likert questions. A '1' indicates that the participant agrees with the statement, '5' indicates that they disagree. A Kruskal-Wallis test corrected for ties found only one of these differences is reliable: the first question in phase two: *I was confident with my answers* ($H=6.55$, $df=2$, $N1=N2=N3=15$, $p<0.05$).

3.3.4 Comments from Participants

After taking part, participants were asked for comments about the evaluation.

General Comments: four participants (spread among conditions) commented the questions were hard.

English Condition: Ten of the fifteen participants in the English condition found the English code easier, one found the conventional code easier, and one found them about the same. However, one of the participants who found the English notation easier also said they preferred the Conventional notation: they found it more fun because they had to think more.

Multiple Condition: Eleven of the fifteen participants in the multiple condition liked or found the tooltips useful. However, two participants said that it was more fun without the tooltips (the tasks were more challenging). One participant said having the tooltips in the first phase helped them understand the code in the second phase.

Conventional Condition: One of the fifteen participants in the Conventional condition commented that the notation was confusing: the lines between words, brackets, and dots had

no meaning. Another participant in the conventional condition mentioned they would like to see the programs run, and another mentioned "*it was easier than doing it on paper*" (we believe they were referring to answering the multi-choice questions).

3.3.5 Summary of Results

Participants in the Conventional condition were faster than participants in the English condition ($F_{(2,42)}$ =3.60, p$\leq$0.05). The speed of participants in the Multiple condition was between the Conventional and English conditions. There was no reliable difference in accuracy between any of the three experimental conditions ($F_{(2,42)}$ =0.030, p=0.971).

In the second phase, participants in the Conventional condition were more confident with their answers than other participants. Most participants who experienced different interfaces (those in the English or Multiple conditions) preferred the interface with an English representation. Some participants preferred having a Conventional interface as it made the questions "more fun."

3.4 Discussion

This evaluation was designed to examine two hypotheses. The first hypothesis predicted that children aged eleven would answer questions about computer programs with an English notation faster and more accurately than questions about computer programs without an English notation. The second hypothesis predicted that children who interact with a conventional-style notation will be faster and more accurate at answering questions about computer programs than children who do not interact with a conventional-style notation. The analysis of the data rejects our first hypothesis and supports our second hypothesis.

This rejection of the first hypothesis and the acceptance of the second hypothesis means that children are more efficient (when measured as time and accuracy) when understanding programs written in conventional code. We believe children reading English code carefully parse the English to construct understanding, while children reading the Conventional code scan over the code and gain enough understanding to answer the questions as accurately as children reading English code.

The increase in efficiency from conventional code draws doubt of the usefulness of English-like notations of computer programs. We believe that English-like notations are still useful:

comments from the children indicated that they preferred using English-like notations, and other research indicates that English-like notations are useful for the writing activity [114, 136]. However, English-like notations should only be provided in conjunction with a conventional-style notation.

We found the Likert data intriguing. In the testing phase, children who used the Conventional condition were most confident of their answers; children in the Multiple condition were least confident. This difference in confidence is reflected in the accuracy data: children in the Conventional condition (who were the most confident) answered a mean of 63% correct (σ7.2); children in the English condition answered a mean of 60% correct (σ8.1); and children in the Multiple condition (who were least confident in their answers) answered 54% correct (σ8.2). Unfortunately the difference in accuracy was not reliable ($F(2,42)=0.030$, $p=0.971$). One interpretation is that taking away a representation when a participant is used to multiple representations is more damaging to confidence than changing representations.

These results combined with the informal comments from the participants provide confidence in the value of multiple representations. Multiple representations provide access to a more efficient representation (conventional code) while providing access to a preferred representation (English). However, we note that once users have access to multiple notations the notations should not be taken away— although users should be able to view only one notation when they desire.

3.4.1 Limitations of Evaluation

We identified several limitations of this evaluation.

The primary limitation is that we only evaluated the reading task (our motivation for only evaluating reading is in Section 3.1.1). This is a limitation because we can not be sure that our results transfer to the writing or watching tasks: indeed there is other research indicating children when unconstrained (for example, when describing programs on paper with a pen) write programs in English [114, 136]. Because children naturally express programs in an English-like syntax we would expect them to be faster when using that syntax rather than using a conventional-style syntax. However, the previous chapter argued that programming environments should use a syntax-directed editor. Syntax directed editors constrain programmers and remove syntactical issues. We are unsure how the research on writing notations would apply when children are constrained: when they writing programs using a syntax directed editor.

Another limitation is that our results might not generalise to other populations. In this eval-

uation our participants were children aged 10 or 11 from three decile nine primary schools in Wellington. Decile nine indicates that the children are from high socio-economic areas[1]. Further research needs to predict how our result generalises to children from different age groups, socio-economic backgrounds, and different language backgrounds. We are also unsure if these results generalise to adults: adults have higher reading skills than children aged eleven so the parse time of the English notation might not slow them down.

Another limitation is we did not control for previous knowledge of programming nor knowledge of games similar to those being studied. Previous knowledge of programming could affect our results as children would understand the conventional notation more easily. Previous knowledge of the types of games being studied could result in children successfully guessing the correct answers to the questions without reference to the notations.

The final limitation is that we compared two textual notations that had a 1:1 semantic mapping. While we intentionally made this decision (see subsection 3.1.2), it would be interesting to repeat the evaluation when using notations that do not have a 1:1 semantic mapping.

3.5 Summary

This chapter described an evaluation examining how quickly and accurately children understand computer programs using different notations. We expected that children using an English-like notation would be faster and more accurate than children using a conventional notation. We observed the opposite: children interacting with a conventional notation understood computer programs faster than children interacting with an English notation. This result has implications for researchers who believe programming using English-like notations is better than programming using conventional-style notations: these researchers must show that use of an English-like notation aids the writing or watching tasks—this evaluation proves that an English-like notation can slow the reading task.

Comments from children indicate they preferred the English notation, although some preferred just having the conventional notation. This provides more motivation for multiple notation programming environments: people can have access to both a more efficient notation (the conventional code) and a notation they prefer (the English code). This result also influences the design of programming environments: because some users preferred just having the conven-

[1] New Zealand Schools are ranked from decile one to ten based on the population the school draws its students from. Decile one is the lowest and decile ten is the highest.

tional notation, users of multiple notation programming environments should be able to turn one notation off. This is easily achieved using tool-tips to show the additional notation.

There are several areas this evaluation could be improved or modified in future work. First, we believe that adults, who have stronger language skills than children, might perform differently when interacting with the conventional notation. Second, we would like to repeat the evaluation using simulations that are different to puzzles or games that the children have previously encountered. Third, we used math scores to select the participants, but it might be more appropriate to use English scores: especially as the evaluation is examining transfer from English to programming syntax. Fourth, it would be interesting to see if the results changed if the semantic 1:1 mapping between the notations was relaxed.

Part II

Collaboration

Chapter 4

Collaboration and Learning

Collaborative programming environments provide a representation of a program to multiple users. We expect children will collaborate using computers for three reasons. First, educational theorists believe that collaboration helps learning. Their reasons range from arguing that interacting with a teacher or more competent peer increases learning [186] to arguing that knowledge is only created by creating sharing meaning [100,172]. Second, research about professional programmers finds that that when programmers work in groups they produce more code with fewer bugs [126]. Third, many classrooms contain only one computer (the goal of a computer for every student is expensive), so we anticipate students will collaborate around the computer doing projects in groups rather than individually. This collaboration will create contention for the input devices (typically keyboard and mouse). We also anticipate that computers in classrooms in the future will have multiple input devices: the cost of an extra mouse or keyboard is far less than the cost of an extra computer.

This part of the thesis provides an overview of computer-supported collaboration and describes an empirical evaluation of how different ways of collaborating affect the performance and learning of children. It consists of two chapters. The first chapter presents an analysis of

computer support for collaborative learning and describes several findings. First, little empirical research examines how computer supported collaboration affects learning. Second, some modes of computer supported collaboration do not have parallels in the real world, so they might cause different learning outcomes that can not be inferred from the existing research about collaboration and learning. Third, little research examines how these new modes of collaboration affect learning. Fourth, professional programmers regularly collaborate, but few collaborative programming environments for children have been built. The next chapter describes an empirical evaluation of how different modes of collaboration affect learning.

In this chapter we first describe the support for collaboration in children's programming environments. We find that few programming environments for children support collaboration. Next we analyse what supports for collaboration a children's programming environment could provide. Finally, we describe research examining how these different supports can create different measurable learning outcomes.

4.1 Programming

While professional programmers often work in teams and use complex collaborative development environments, few collaborative programming environments have been built for children. We believe that collaborative programming environments should be built for children for two reasons: first, many educational theorists believe that collaboration is very important to create understanding and, second, professional programmers almost always work in teams. This section reviews collaboration by professional programmers as well as the collaborative programming environments developed for children. We find that while professional programmers regularly program collaboratively, little research has examined how children program collaboratively and few programming environments for children support collaboration. This section also extends Language Signatures (described in chapter 2) to describe how different environments support collaboration for different activities.

4.1.1 Professional Collaborative Programming

Professional programmers collaborate on both large and small scales. On a large scale, software engineering methodologies help programmers organise their problem and write code to solve their problem. There are several collaborative techniques to help programmers collabo-

rate, including design reviews, Fagan testing, structured code walkthroughs, and pair programming. This section examines two: pair programming [126], as an example of programmers working together to create programming artifacts (code and design); and Yourden's structured walkthroughs [203], as an example of programmers collaborating to examine programming artifacts after the artifacts have been created.

Pair programmers write code collaboratively on one computer: one programmer writes code and the other sits (or stands) behind them and comments on the code, notices bugs, and suggests improvements [10]. The two programmers swap roles regularly. There is evidence that pair programming is a powerful technique to reduce bugs, produce better designs, solve problems faster, increase enjoyment of work, and reduce reliance on a teacher [30, 126].

Yourden's structured code walkthroughs are used as part of program development: multiple programmers sit down with a copy of the code and collaboratively examine it line by line, looking for bugs and possible improvements. A code walkthrough typically finds many bugs (one review found 17 bugs in 13 lines of code [187]).

Many collaborative tools support collaboration by software engineers. These tools range from providing synchronous groupware support, where programmers can work on the same part of a program at the same time (Grundy and Hoskings provide a good overview of these tools [67]), to environments that provide asynchronous collaborative support, where programmers store code in a central repository either on the web or on a local server (Eclipse is an example of a development environment where programmers can collaborate asynchronously [78]).

There are several problems with asynchronous collaboration. As the different collaborators might not be aware of changes that other collaborators have made, it is easy for collaborators to make conflicting changes to any programming artifact (design, code, test cases, user interface, etc). To help avoid programmers overwriting changes made by other programmers or even writing code that conflicts with other programmers, most asynchronous software engineering tools provide support for versioning of code and merging conflicting code. To aid these tasks the environments need: awareness support, so programmers know what has changed and conflicts; conflict resolution tools, to find what code conflicts; traceability, so different programmers can figure out how different versions were merged; and communication support, so software engineers can ask each other what their changed code was trying to achieve. These problems do not occur to the same extent in a software engineering tool that supports synchronous collaboration — awareness support increases social protocols and helps programmers avoid conflicts.

4.1.2 Children's Collaborative Programming

There are few collaborative programming environments for children. We identified four: AlgoBlock [177], Cleogo [28], Moose Crossing [20], and the AgentSheets Behaviour Exchange [152]. All these environments aid small-scale collaboration, some of the environments help large-scale collaboration.

AlgoBlock is based on Logo, a programming environment developed in the late 1970's by Seymour Papert [137]. AlgoBlock statements are functionally equivalent to Logo statements, but they are represented as physical blocks instead of text. AlgoBlock users connect the blocks and watch a submarine navigate a maze (instead of a turtle draw lines). An evaluation of AlgoBlock found that children easily learned to program using the blocks, and that the blocks functioned "*as an open tool which facilitates interaction among learners.*" Unfortunately the AlgoBlock authors neither compared collaboration with solo work, nor compared children collaborating using AlgoBlock with children collaborating with a non-tangible version of Logo.

AlgoBlock supports small-scale collaboration. Users pair-program (modify the program together).

Cleogo is a groupware extension of Leogo [27]. Like AlgoBlock, Leogo is based on Logo, but Leogo provides multiple ways to interact with the turtle: users can directly manipulate the turtle, click on icons representing program statements, or type textual statements. Cleogo has support for multiple telepointers, and each user sees exactly the same version of the program at all times. Although Leogo (the single-user version) was evaluated, Cleogo (the collaborative version) was not.

Cleogo supports small-scale collaboration in two ways. First, users can perform a code walkthrough by collaboratively stepping through a program. They can use the walkthrough to track bugs or to understand a program. Second, users can pair-program (modify the program together).

Moose Crossing is a environment where children can write and interact with "*places, creatures, and other objects that have behaviours in a text-based multi-user virtual world (or 'MUD').*" [22] Because there has been success using MUDs (Multi-User Dungeon) as a vehicle for teaching programming [19], Bruckman developed Moose Crossing to help

children learn creative writing skills by programming [20]. In an evaluation of Moose Crossing Bruckman found that the children enjoyed using Moose Crossing and concluded that "*CSCL environments can help to foster and support collaborative learning in schools.*" [21]

AgentSheets Behaviour Exchange is an extension of AgentSheets where children can share programs they have written and extend the behaviour of programs other children have written. We can not find any papers describing how it was used.

Both Moose Crossing and AgentSheets support small-scale and large-scale programming. On a small-scale, users can share programs and perform code walkthroughs. On a larger-scale, users can download other users' code and incorporate it in their programs.

4.1.3 Extending Language Signatures

In subsection 2.1.4 we described Language Signatures: a concise way of describing how different notations are used for our three programming activities in a programming environment. This section extends the Language Signature syntax to describe how different notations are used for different synchronous collaborative activities in a programming environment.

Currently Language Signatures describe the notations used for reading, writing, or watching. To increase flexibility we extend the syntax to describe which notations support collaboration **and** for which activities each notation supports collaboration. In this extension, a notation supports collaboration for an activity if the programming environment provides more support for collaboration than a single-user computer with a single display, a single keyboard, and a single mouse. In particular:

↪ A notation supports collaboration for writing if multiple users can modify different parts of the notation at the same time.

↪ A notation supports collaboration for reading if: multiple users can view the notation at the same time in different places, or multiple users can view different parts of the notation at the same time in the same place.

↪ A notation supports collaboration for watching if multiple users can watch the animation of the notation at the same time when they are in different places or watch different animations of the same program at the same place.

To indicate if a notation supports collaboration for an activity we underline the supported activity. For example, Leogo lets multiple users interact with the same view of a program in different places. Leogo's extended Language Signature is [$\underline{\text{RE/WR/WA}}$ + $\underline{\text{WR/WA}}_{iconic}$ + $\underline{\text{WR/WA}}_{direct\ manipulation}$]. AlgoBlock lets users simultaneously modify a program, but they must be in the same place to read or watch the program, and all users must view the same representation of a program. AlgoBlock's extended Language Signature is [RE/$\underline{\text{WR}}_{physical\ blocks}$ + $\text{WA}_{turtle\ and\ lines}$].

4.2 Collaborative Applications (Groupware)

Although there are few collaborative programming environments for children, many researchers have examined how to build computer support for collaboration. This section examines the support for collaboration provided by groupware applications. Based on this examination we can determine what types of collaborative support children require in a collaborative programming environment.

Simply stated, groupware applications are applications where users are aware of being in a group. These users typically interact with a shared artifact ranging from a simple whiteboard or text document to a complex multi-level design such as a computer program or aircraft design. Groupware applications are typically classified based on two properties [9]. The first property is time: groupware applications where users collaborate at the same time are called synchronous groupware applications, whereas applications where users collaborate at different times are called asynchronous groupware applications. This chapter is primarily concerned with synchronous groupware applications. The second property is space: whether users must be in the same place or in different places. Table 4.1 shows the distinction.

While researchers have alleviated technical issues of application development for groupware applications [61, 142], developers still face other design issues. One important design issue with synchronous groupware applications is that people need to interact both as individuals and as members of their groups [69]. These different roles require different supports from a groupware application: when a user interacts as a member of a group, they need to know what other members are doing and when a user interacts as an individual, they need private spaces to carry out their own work. These two roles, group member and individual, are not separate: users may move from interacting as a group member to interacting as an individual and back depending on their tasks and goals. Researchers have named several classes of groupware applications that lie along

	Same Time	**Different Times**
Same Place	Face to Face Interactions ↪ Public Computer Displays ↪ Electronic Meeting Rooms	Remote Interactions ↪ Video Conferencing ↪ Collaborative Editors
Different Places	Ongoing Tasks ↪ Team Rooms ↪ Shift Work Groupware	Communication and Coordination ↪ E-Mail ↪ Version control

Table 4.1: The Space/Time Groupware Matrix, after Baecker *et al* [9].

this spectrum. These classes are summarised in Table 4.2 and are described below.

Strict-WYSIWIS applications (What You See Is What I See, [175]) lie at one extreme of the group-individual spectrum. This class requires all computer displays are identical at all times, creating problems as users are overloaded with awareness information (many telepointers and cursors) and are unable to use private information (like read email or perform other individual tasks).

Further along the spectrum are relaxed-WYSIWIS [174] and WYSIWITYS (What You See Is What I Think You See, [170]) applications. Users of Relaxed-WYSIWIS applications can move and resize application windows independently and can have different applications running. While users can access private information (like email) they can not do so using the groupware application: the groupware application shows the same view of the shared artifact. Users of WYSIWITYS applications view and interact with different parts of a shared artifact. The application provides awareness supports so each user knows which part of the shared artifact every other user is working on. The main awareness supports are multiple telepointers, multiple scrollbars, and gestalt views [71]:

Multiple Telepointers: Applications with multiple users should provide a separate telepointer (mouse cursor) for each mouse. Separate telepointers allow users to work independently and to make deictic references to parts of the artifact they are editing. A deictic refer-

Application Type	Description
Strict-WYSIWIS (What You See Is What I See)	The data in each user's physical display is identical
Relaxed-WYSIWIS	The data in each user's groupware application is identical. Users can move, resize, and scroll the application independently. Users can also use single-user programs.
WYSIWITYS (What You See Is What I Think You See)	Each user is viewing or modifying a different representation of the same shared artifact. Users can scroll independently. The groupware application needs to provide enough awareness that users know which part of the shared artifact all other users are working on.
Single User Applications	Each user works independently. Changes to shared artifacts need to be merged at a later time.

Table 4.2: Classes of groupware applications providing different supports for individual and group activities. The table is ordered by the amount of support for private information the application type provides. Applications on the top support only mainly work while applications on the bottom support mainly private work.

ence is a reference where a user says "*this part here*" while indicating the part with their telepointer.

Multi-user Scrollbars: Users should be able to easily find out what part of a document other users are editing. Multi-user scrollbars show every user every other user's scrollbar (but they can modify only their own scrollbar).

Gestalt Views: are a richer version of multi-user scrollbars requiring more screen real-estate. They show a miniature of the entire document overlaid with boxes representing the viewports of all users. Gestalt views are used primarily for 2D spaces.

Intention Awareness: Another interesting aspect of awareness support is *intention awareness* [76]. These supports help users of synchronous groupware systems determine what other users intend to do. The supports include buttons depressing when a user mouse clicks on a button but has not yet released the mouse and transparent menus or drop down lists so other users can see what menu choices a user is going to make. The transparency is important so that the intention information does not disrupt the users' workflow

Whereas awareness supports help users who are working as members of groups, privacy support helps users who are working as individuals. There are several reasons why privacy support is necessary: some users enjoy having a private space [163]; applications run out of screen real-estate if multiple users all have multiple window open; users can be overloaded with awareness information [69], and displaying all information about a complex artifact creates information overload [180]. Unfortunately providing privacy support reduces both the screen real-estate available for awareness support and the total amount of awareness provided by the application.

Providing private areas is technically easy with remote groupware applications (simply provide public and private areas) but much harder with co-located users who are sharing a display. Recent research has examined the effects of providing multiple displays for co-located users (different displays for private and public information) [150] and using modified 3D glasses so two users can view different images on the same display [163].

This tradeoff, between the amount of awareness information and the amount of private space, exists because users of groupware applications need to interact both as members of a group and as individuals. Researchers believe that support for both activities can be provided by examining user tasks, analysing the awareness requirements of the work situation and shared artifact, and evaluating which types of awareness and privacy supports best manage the tradeoff [69].

Gutwin, Stark, and Greenberg argue that these awareness supports are even more important in an educational setting because users have two tasks: to work together on a project *and* to learn [71]. They present a taxonomy of awareness support with four levels of awareness needs: social awareness (about group expectations and roles), task awareness (of how the task will be completed), concept awareness (how does this activity fit into their existing knowledge), and workspace awareness (what are other users are doing right now). Gutwin *et al* argue that different levels of supports are needed for each type of awareness, and, for workspace awareness, the same awareness/privacy tradeoff (identified previously) is evident with the same problems. Unfortunately they only concentrate on the information needed to provide workspace awareness in different collaborative settings. They do not examine the effects *on learning* of the different types of awareness/privacy support.

4.3 Impact of Different Modes of Collaboration on Learning

The previous section examined groupware applications. It described different types of collaboration (same or different space or time), and how different people's tasks need a different balance

between private spaces and awareness support. Another factor creating differences in collaboration types is the physical hardware used for input and output. We say that the "*modes of collaboration*" differ when differences in input and output devices, space, and time exist. Literature describing collaboration and learning does not always distinguish between issues of performance and learning in collaborative settings. However, many empirical studies measure performance as how well participants work in pairs, and measure learning as how well the participants perform once separated. We call this measure of learning "*measurable learning outcomes*." This section describes research that examines how different modes of collaboration affect measurable learning outcomes.

Corresponding to the research examining how people use different modes of collaboration in groupware applications (eg. [63, 66, 69, 70, 71]), much research has examined how computer supported collaboration affects learning (eg. [12, 38, 99, 130, 131]), however, much of the work examines process of collaboration than measuring empirical benefits of collaboration. For example, Koschmann, when describing different approaches to computer supported learning writes:

> *As a consequence, CSCL studies tend to be descriptive rather than experimental.* [99]

Even as recently as 2002, Bruckman, Jensen, and and DeBonte indicated that empirical studies are not common in this field:

> *We contrast these quantitative findings with our qualitative observations and conclude that quantitative analysis has an important role to play in CSCL research.* [23]

Additionally, the empirical benefits of collaboration are hard to determine. Blaye and Light's do not provide much confidence in collaboration when they argue that collaboration does not induce *worse* post-test performance than working along:

> *The benefits of collaboration were not always highly significant but, on the average, peer work never induces worse post-test performance than individual work.* [12]

However, this view is mitigated by Dillenbourg, Baker, Blaye, and O'Malley. They argue that collaboration has the potential to be useful, and our aim should be to find out when it is useful:

> *Collaboration works under some conditions, and it is the aim of research to determine the conditions under which collaborative learning is efficient.* [38]

Some of this research, which examines how different modes of computer support for collaboration affect learning, focuses solely on input device contention [1, 83, 84, 85, 86]. Unfortunately none of this research shows that collaboration increases measurable learning outcomes when compared to working solo (an evaluation was outlined by McGrenere [113], but the results were not published). To further the examination of collaboration creating different learning outcomes, the remainder of this section reviews the evaluations of collaborating with peers and finds that although peer collaboration does not create different measurable learning outcomes, different modes of computer support for collaboration might. They provide collaboration modes that can not happen without a computer. The next chapter describes an evaluation of how different modes of collaboration affect learning.

Researchers have performed several evaluations examining how different modes of computer-supported collaboration affect learning and performance. Results from these evaluations show that:

↪ *Children perform better when collaborating than when working alone.* Inkpen, Booth, Klawe, and Upitis describe a study where 435 children played a game called "The Incredible Machine." [83] The found that children who were using the computers collaboratively performed better than children who were solving the puzzles by themselves and children who were collaborating together on one machine performed better than children who were collaborating on two machines. During this evaluation, Inkpen *et al* noticed that the mouse control protocol affected performance: girls performed better when they gave mouse control to their partner and boys worked better when they took mouse control from their partner. There are two limitations of this evaluation: in the two machine condition, the software neither used groupware mechanisms to keep the two machines' displays consistent nor included any awareness support, and the evaluation did not test measurable learning outcomes: children were not tested by themselves *after* solving the puzzles collaboratively.

↪ *Different mouse control protocols affect performance and learning.* To further examine the differences in mouse control protocol identified by Inkpen *et al* [83], Inkpen, McGrenere, Booth, and Klawe describe an evaluation where 252 children played "The Incredible Machine." [85] In this evaluation, Inkpen *et al* did separate the pairs to test them individually after training, but did not have a solo group to compare the pairs' performance. This evaluation found that the time males had control of the mouse was a reliable indicator of how

well they could solve the puzzles alone. The effect was not reliable for females. This evaluation is important because it shows that different mouse control protocols can create different measurable learning outcomes.

↪ *Higher engagement with multiple mouse cursors.* Inkpen, Ho-Ching, Kuederle, Scott, and Shoemaker describe an evaluation where 40 children solved a pattern matching game [86]. This evaluation compares three interfaces to the game: a physical paper-based interface, a computer interface with one mouse and one mouse cursor, and a computer interface with two mice and two mouse cursors. The evaluation found that children interacting with two mice and two cursors were more engaged in the puzzle than children interacting with one mouse and one cursor. Unfortunately, they do not compare the engagement in the computer conditions with the engagement of children using the paper-based interface. They do report the data. In the computer-based interfaces, children were off-task for up to three minutes in a 10 minute session, whereas in the paper-based interface, children were only observed to be off-task 4 times (15 seconds) during the 10 minute session. This is a strong indication that computers and physical artifacts support collaboration in different ways. However, the primary limitation of the study is that children working collaboratively were not compared with children working alone: there was no control group.

↪ *Children collaborating with multiple input devices talk more about their tasks.* Abnett, Stanton, Neale, and O'Malley describe an evaluation where thirty-six children wrote stories using one of two collaborative computer interfaces [1]. In the first interface, children had access to only one mouse. In the second interface, children had access to two mice and two mouse cursors. They found that the two-mouse interface "*did not stop children discussing their joint work, but they talked more about what they themselves were doing.*" Unfortunately there are several limitations of this evaluation: there was no control group working alone and the two-mouse interface was more powerful:

> "*In this case, the children are not prevented from drawing as individuals, but they can gain additional benefit (new colours and filled areas) by working together.*" (the software they used is described in detail by Benford *et al* [11]).

These evaluations are important because they show that different modes of collaboration can create different measurable performance and learning outcomes. Unfortunately they do not

provide us with enough information to compare the measurable learning outcomes created by children working together with the outcomes created by children working alone.

Much research compares how children learn when working alone with children working together — it examines peer collaboration in children without computers rather than computer-supported peer collaboration. The empirical research examining how peer collaboration affects measurable learning outcomes concludes that collaboration does not affect learning. For example, in 1992 Elshout cited a 1976 a review of 22 experiments that found only two experiments with reliable results: one in favour of collaboration and one in favour of working alone ([79] cited in [45]). These experiments were wide ranging, using a variety of subjects from grade seven to first year university. They examined different forms of pair composition and different types of problems. Most experiments had 70 to 80 participants. Elshout also wrote that there was:

> *Nothing in the the literature of the 14 years since [Hoogstraten's 1976 review], that indicates that things have changed; or that they are different in other educational contexts.* [45]

More recent studies have found

> *No clear cognitive benefit from working in a pair in terms of pre- to post- and pre- to delayed post-test gains.* [87]

In the years since 1976 researchers have developed many computer supported collaborative tools. These supports range from allowing users in remote parts of the world to collaborate using multiple computers [66] to allowing users in the same place to collaborate using a touch sensitive surface [50]. Researchers have conducted many usability evaluations on these systems and some researchers have conducted studies evaluating how these new technologies affect learning. Generally, these studies are positive. For example, Blaye and Light describe two evaluations investigating how collaboration affects planning skills in adults and children [12]. The first of the studies is more important here. In this study, participants were asked to solve a planning problem in pairs or by themselves and then were separated and their planning skills were tested using a similar problem. Blaye and Light found that solving the problems collaboratively led to a statistically significant increase in the ability to solve problems individually: empirical evidence that collaboration can increase learning outcomes. Additionally, O'Malley describes several factors influencing effective computer supported collaboration [130]. These factors include group size, gender, ability mix of dyads, and type of task. O'Malley also notices that *conflict* is an important factor in creating learning opportunities.

Additionally, there are positive aspects of computer-supported collaboration. Stanton and Inkpen report higher levels of task engagement [86, 173], Scott reports that participants found the task easier [161] (although they did not perform measurably better), and Inkpen found that girls perform better when collaborating [83]. Intuitively we believe that collaboration has other important side-effects, like learning to work in a team and developing social skills.

There are several limitations of applying the human-supported collaboration research results to computer-supported applications. Users of groupware technology can interact and modify a shared artifact simultaneously while being aware of each other and being in different locations. This mode of collaboration does not exist in the real world. Additionally, users of groupware applications can interact with a shared artifact without contention for any input device while users with computer technology must engage in social protocols to negotiate for the input device. It is not immediately obvious that the research examining human-supported collaboration translates to computer-supported collaboration, especially when there are modes of computer-supported collaboration that do not exist without computers. The next chapter describes an evaluation of the different measurable learning outcomes provided by these modes of collaboration. In particular we examine the planning strategies that participants engage in while collaborating compared with the planning strategies they use when working alone.

4.4 Summary

This chapter examined usability and learning issues of collaborative applications. We found collaborative applications need to have a balance between group and individual work, and collaboration does not necessarily increase measurable learning outcomes. A review of collaborative programming environments found while collaborative programming is useful for professional programmers, few collaborative programming environments exist for children. The collaborative programming environments for children that do exist show that children can program collaboratively and that children enjoy programming collaboratively. However, it may not matter than few collaborative programming environments exist for children: it is feasible that children program as well grouped around one computer as children using groupware programming applications with multiple input devices.

The next chapter describes an evaluation that tests the hypothesis whether children generate different learning outcomes when collaborating with or without input device collaboration or when not collaborating. The evaluation finds that although there is no measurable difference

in learning outcome, there were performance differences. This result means that programming environment developers for children need not write groupware applications: children can collaborate as well around one computer as children using a groupware application.

Chapter 5

Collaboration Evaluation

As described in our previous chapter, many researchers have investigated how to use computers to aid collaboration. Additionally, much work has been done developing computer support for collaborative learning. Many of these systems overcome the limitation caused by input device contention by allowing multiple users to simultaneously work with a computer-supported artifact such as a puzzle, virtual world, or interactive story. As computers become more available in the classroom, groupware applications can feasibly be used for new styles of collaboration among local and remote students. However, little evaluative work has been carried out investigating how these new modes of collaboration affect learning.

This chapter reports an evaluation of three modes of computer supported collaboration: people working alone, people working in pairs on one computer, and people working in pairs on two adjacent computers. In the third mode, awareness supports ensure that the displays stayed consistent and that there was no contention for the mouse. The evaluation found no reliable effect on how well the participants learned to solve the puzzle, but there was a reliable effect of

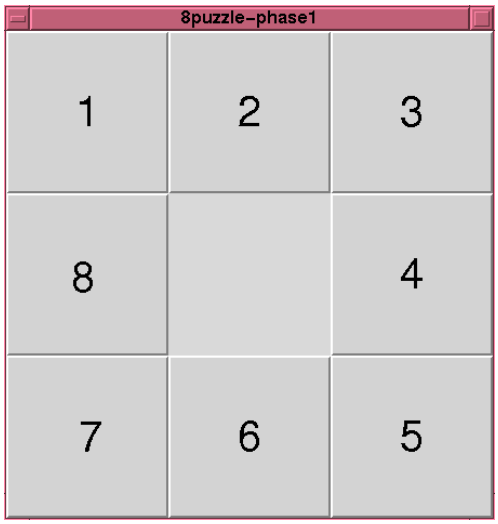


Figure 5.1: The 8-Puzzle. Users click on a piece to move the piece into an empty slot. Users can only move a piece into the empty slot if the piece is adjacent to the empty slot.

performance: females working alone solved the puzzle faster and in fewer moves than females working in pairs. An analysis of the data indicates that female participants feel unsure of their move sequences when working in pairs.

5.1 Motivation

Although researchers have so far failed to find empirical benefits of paired collaboration on learning (described in the previous chapter), they have found that other human-computer interaction techniques do empirically improve learning outcomes. The most promising of these is cognitive interface cost: studies show that people learning to solve a task using a high-cost interface learn to solve the task better than users who learn a task using a low-cost one. The types of cost examined include using a keyboard instead of a mouse [179], adding a delay to user actions [128], removing undo [129], using indirect instead of direct manipulation [60], and using meaningless labels [43]. These are costs that are generally considered bad in conventional user-interfaces — interfaces where designers are focused on users learning the interface rather than using the interface to solve a complex task [55].

The 8-puzzle is used in many of these evaluations (see Figure 5.1). It consists of a three by three grid with eight numbered pieces and one empty slot. Users work towards a particular

target configuration (such as the one shown in the figure) by sliding pieces into the one empty slot. The puzzle was examined in depth by O'Hara and Payne [128], who found that the more planning participants did, the better the participants learned the puzzle, and that a high interface cost promoted planning. The cost used in O'Hara and Payne's evaluation was implemented as a seven second delay

O'Hara and Payne used two measures to determine how much participants were planning. The first measure was how well participants learned to solve the puzzle and was determined by timing participants' performance after training. The second measure was the number of reversed move sequences: an indication that a participant has tried a sequence of moves and then reversed the sequence. O'Hara and Payne argue that a reversed move sequence is an indication that participants are tracking their thoughts using the puzzle interface and are not actively engaging in planning. They write: "*When the cost associated with an operator was relatively high, problem solving strategy became more plan-based, whereby search paths were considered and evaluated mentally.*" [128]

While many evaluations have attempted to quantify the effects of collaboration on learning, there is little evidence that the researchers have examined the evaluations from an interface cost perspective, and there are several reasons why such an examination could prove fruitful:

↪ In the tasks examined in previous experiments on collaboration, there is no evidence that increasing planning could create different learning outcomes.

↪ The effect of mouse control negotiation and communication could create a high cognitive cost of interaction. This high cognitive cost could cause increased planning.

While this puzzle does not directly examine programming behaviour, we believe that it shares features with programming behaviour. The analysis of the puzzle examines planning behaviour when collaborating or working alone. One of the problems learner programmers have when programming is composing programs [171]. Program composition is a problem that can be solved using two approaches: a situated action approach (write some code, check if it works, and then modify code), or a planned action approach (think about code, formulate plan, write code, check if it works, rethink plan, rewrite code). By using this puzzle, we can determine to what extent collaboration influences planning behaviour.

This experiment examines collaboration from a high/low cost perspective. We expect that collaboration will cause a high-cost interface, and predict these effects:

H1 Users who learn to solve the 8-puzzle in pairs will solve the puzzle in fewer moves after training than users who learn to solve the 8-puzzle individually.

H2 Users who learn to solve the 8-puzzle in pairs will solve the puzzle in less time after training than users who learn to solve the 8-puzzle individually.

H3 Users solving the 8-puzzle in pairs will have a higher inter-move latency than users solving the puzzle individually.

H4 Users solving the 8-puzzle in pairs will have fewer moves in reversed sequences than users solving the puzzle individually.

5.2 Experimental Design

The experiment investigates the effectiveness of three different modes of computer supported collaborative learning in supporting children learning to solve a particular puzzle. The puzzle used is the 'eight-puzzle', shown in Figure 5.1, which consists of a three by three grid with eight numbered pieces and one empty slot. Users work towards a particular target configuration (such as the one shown in the Figure) by sliding pieces into the empty slot. In our user interface, mouse clicking any tile that is adjacent to the empty slot causes the tile to slide into the vacant position. The tile's movement is rapidly and fluidly animated, providing a clear indication of the direction of motion. This puzzle has been used with success to evaluate the effects of computer interfaces on learning [128, 179].

The experimental design is similar to the evaluation described in chapter 3: each of the fifty participants, aged ten and eleven, was asked to solve the eight-puzzle a total of ten times, with five trials in a 'training' phase, and five trials in a 'testing' phase. Each participant was assigned to one of three collaboration conditions for the training phase, and in the testing phase all participants solved the puzzle alone using the single user version of the system. The first 'solo' training condition acts as a control, and involves using a single-user version of the puzzle. In the second 'contention' training condition, two participants shared access to the interface used in the 'solo' condition. In the third 'groupware' training condition, two participants, each beside the other with their own computer, screen, and mouse, shared access to a strict-WYSIWIS implementation of the puzzle. The only visual difference between the groupware interface and the solo one was the addition of telepointers, which reveal the location of the other user's cursor

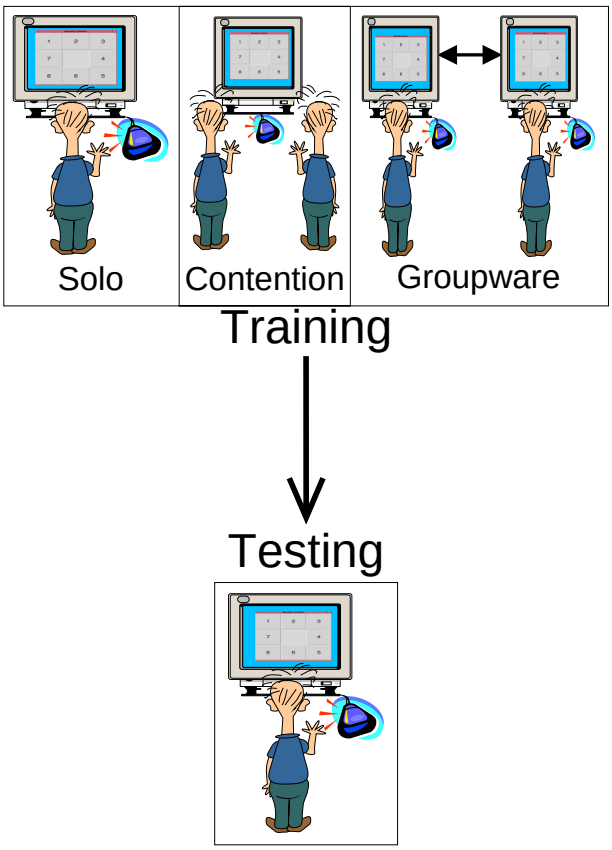


Figure 5.2: Experimental design: the participants were split into three groups for the training phase, but they all performed the testing phase individually.

in the display. In both collaborative conditions participants were located beside each other and could talk freely.

After solving the puzzle five times in the training condition, all subjects, regardless of training condition, moved to the testing phase where they solved the puzzle a further five times individually. The interface used in the testing phase was identical to that used in the solo condition in the training phase. Figure 5.2 summarises the difference between the three conditions used during the training phase and the one condition used during the testing phase.

Switching from the three conditions used for training to the solo condition used for testing allows us to equitably compare how successfully the participants learned to solve the puzzle during training.

5.2.1 Subject Details

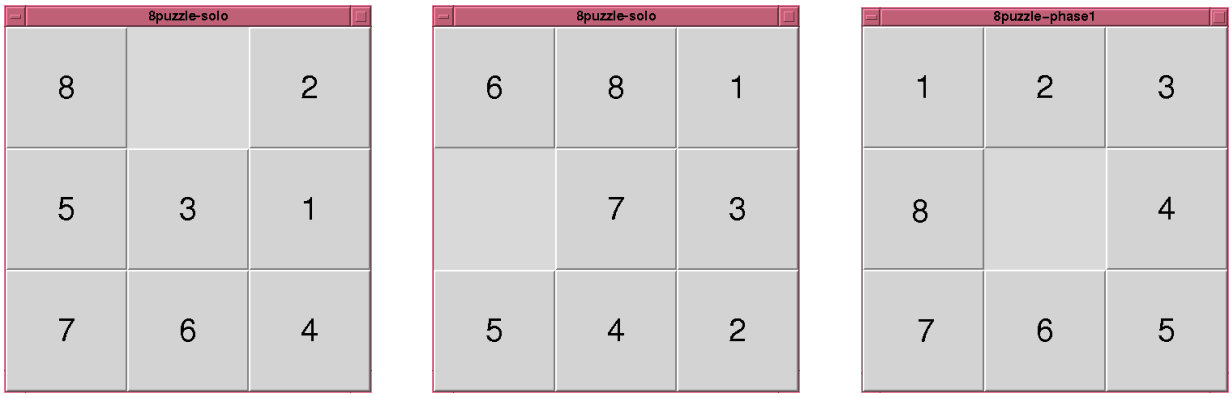
The experiments were conducted at three primary schools in Christchurch, New Zealand. We asked the teachers to select children who were in the middle of their class based on math scores. The participants were allocated to one of the three training conditions, giving sixteen children per group, eight males and eight females (data from three subjects was discarded, as discussed in the results). We used single gender pairs because prior work indicates mixed pairings can detrimentally affect learning [85, 108, 202]. We used equal numbers of each gender as previous work indicates males and females collaborate differently [83].

All participants had a large paper copy of the target puzzle configuration on their desk throughout the experiment (the target configuration is shown in Figure 5.1). Approximately five minutes at the start of each evaluation was spent introducing the puzzle and the interface to the participants. Particular care was taken in the groupware condition to ensure that the children understood the synchronous WYSIWIS properties of the interface.

In the collaborative conditions (contention and groupware) we stressed the importance of talking to the partner in order to negotiate moves in the interface. Each participant's involvement in the experiment lasted approximately twenty to thirty minutes.

5.2.2 Procedure

The goal configuration of the puzzle was the same for all ten trials (Figure 5.3c). The five training trials used the same starting configuration (Figure 5.3a). The five testing trials also used the same starting configuration, but a configuration that was different to the configuration used



(a) Starting configuration used during the training phase

(b) Starting configuration used during the testing phase

(c) Goal configuration used in both phases

Figure 5.3: Starting configurations used in the two phases and the goal configuration, which was the same for both phases. Each starting configuration had a minimum solution length of 17 moves to the goal configuration. All configurations are from [128].

for the training trials (Figure 5.3b). All puzzle configurations are shown in Figure 5.3. The training, testing and goal configurations are identical to the configurations used in the eight-puzzle experiment conducted by O'Hara and Payne [128] and described in section 5.1. The minimal solution length for both configurations is seventeen moves.

In the solo condition, having solved the puzzle five times in the training phase, the subjects paused briefly, then proceeded to the testing phase. In the collaborative conditions, the pair completed the testing phase sequentially: during trial runs we found that testing in parallel caused participants to feel uncomfortably pressured to complete the task as quickly as possible, as though racing their partner. To counter this effect one participant (chosen at random) was asked to play a 'snake' video game called `gnibbles` until their partner had completed the testing phase.

5.2.3 Apparatus

The computer interfaces for the solo and contention groups were identical and were implemented in Tcl/Tk [132]. If a participant clicked on an immovable tile, no feedback was given. When a puzzle was completed, the tiles briefly flashed green; then the participants clicked the mouse to

Phase one			
	Solo	Contention	CSCW
Male	s1–s8	s9–s16	s17–s24
Female	s25–s32	s33–s40	s41–s48

Phase two			
	Solo	Contention	CSCW
Male	s1–s8	s9–s16	s17–s24
Female	s25–s32	s33–s40	s41–s48

Table 5.1: Experimental Design. Both factors (condition in the first phase—see Figure 5.2—and gender) were between subject factors, and the two phases were analysed separately.

advance to the next puzzle. A screen snapshot of the interface is shown in Figure 5.3.

The interface in the groupware condition behaved identically to the solo and contention interface except for the addition of telepointers and concurrency control mechanisms. It was implemented using GroupKit [159] and Tcl/Tk.

The interfaces logged all user actions including: the number of moves per trial, the latencies between moves, the total time per trial, and a history of moves made. All experiments were recorded on video. In the groupware condition, the interface also recorded which user made which move.

5.2.4 Data Analysis

The experiment was designed as a two-factor randomised analysis of variance for the factors training condition and gender. Training condition was a between-subjects factor with three levels: solo, contention and groupware. Gender was a two-level between-subjects factor. We analysed each phase separately to separate the analysis of task performance from the analysis of learning outcomes. Table 5.1 shows the design.

This analysis was repeated for four dependent variables: total moves per phase, total time per phase, latency between moves, and number of reflected move sequences. A reflected move sequence is an indication that subjects tried a sequence of moves and reversed the sequence to return to a previous state, and indicates whether subjects are using a planned action or a situated action approach to solve the puzzle (see Section 5.1). Like O'Hara and Pane, we examine the reversed move sequences in three ways: number of moves in reflected move sequences, average

length of each reflected move sequence, and ratio of moves in reflected sequences to moves needed to solve a puzzle.

Each of these dependent variables provides a slightly different perspective on the nature of the subjects' interaction with the interface and their learning. As subjects' knowledge of the puzzle increases, it is reasonable to suspect that the total number of moves and the total solution time will decrease. High values for inter-move latency indicates that the participants are spending long periods in thought, and might be using a plan based approach to solve the puzzle. A high ratio of moves in reversed sequences to moves needed to solve the puzzle indicates that the participants are not planning move sequences, and are solving the puzzle by manipulating the user interface [129].

We analysed the data in the same way as O'Hara and Payne: we performed a multi-factor analysis of variance (Anova) analysis. Also, like O'Hara and Payne, we log transformed the data to stabilise the variance [46]. All means and standard deviations we report are from the raw data, whereas F and p values are taken from the log-transformed data. To further stabilise the variance, when possible, we group the data by phase: for example, we examine the amount of time to complete phase one, rather than the amount of time for each puzzle in phase one. This stabilised the variance: due to learning effects, the first puzzle a participant solves takes much longer (and more moves) than the 5th to solve. By analysing by 'time to complete a phase' rather than 'time to complete a puzzle' we remove much of this learning effect.

5.3 Results

5.3.1 Training Phase

Time

Anova showed no reliable main effect of time, but did reveal a reliable interaction between gender and collaboration configuration ($F_{(2,42)} = 3.22$, $p < 0.05$) for the time to complete phase one. This interaction (shown in Figure 5.4) indicates that collaborating females take longer than females working alone. A post-hoc test showed this intuition to be correct: there was a reliable effect of collaboration ($F_{(1,22)} = 4.90$, $p < 0.05$), where girls who collaborated took longer (24min 22sec for the phase) than girls who worked alone (15min 38sec). The difference for males was not reliable.

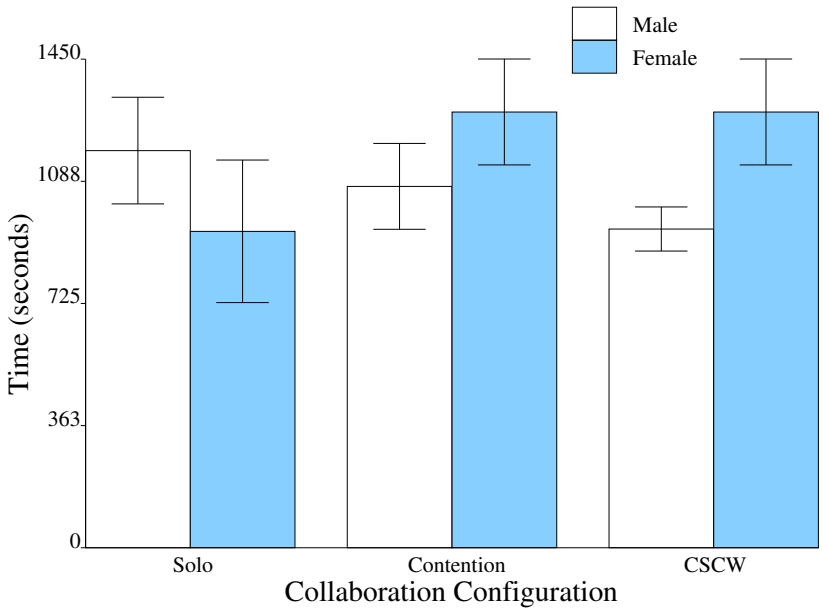


Figure 5.4: Time (in seconds) to complete phase one grouped by collaboration configuration and gender. There is a reliable interaction between these factors ($F_{(2,42)}$ =3.22, p<0.05). Error bars show standard error.

Number of Moves

Anova showed no reliable difference of the number of moves needed to complete phase one, but did show a marginal interaction between gender and collaboration configuration ($F_{(2,42)}$ =2.69, p=0.08). This interaction is shown in Figure 5.5. A post-hoc test showed that females who work in pairs need more moves than females who work alone (1206 moves instead of 776 moves, $F_{(1,22)}$ =5.99, p<0.05).

Inter-move Latencies

There was a reliable effect of gender ($F_{(1,42)}$ =4.46, p<0.05): males were clicking faster than females (a 1.03 second inter-move latency for males and a 1.12 second latency for females).

Number of Reflected Sequences

An analysis of the number of reflected sequences shows a marginal ($F_{(2,42)}$ =2.6, p=0.085) effect of collaboration configuration: participants in the CSCW configuration had the highest number of

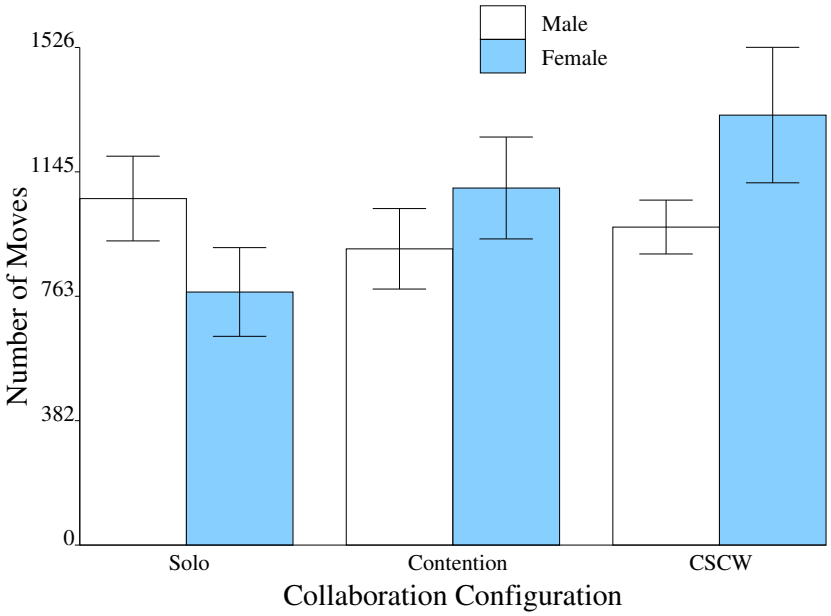


Figure 5.5: Number of moves needed to complete phase one grouped by collaboration configuration and gender. There is a marginal interaction between these factors ($F_{(2,42)}$ =2.69, p=0.08).

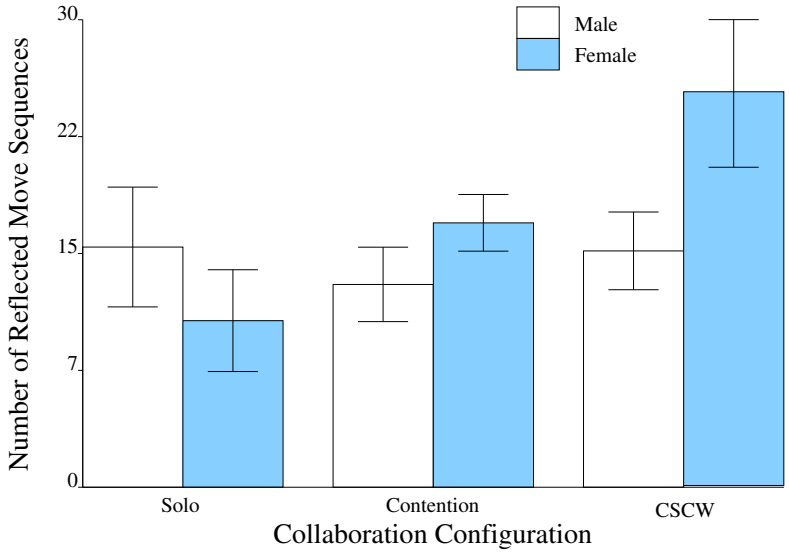


Figure 5.6: Number of reflected move sequences per puzzle in phase one. There is a marginal difference between collaborative configurations ($F_{(2,42)}$ =2.6, p=0.085).

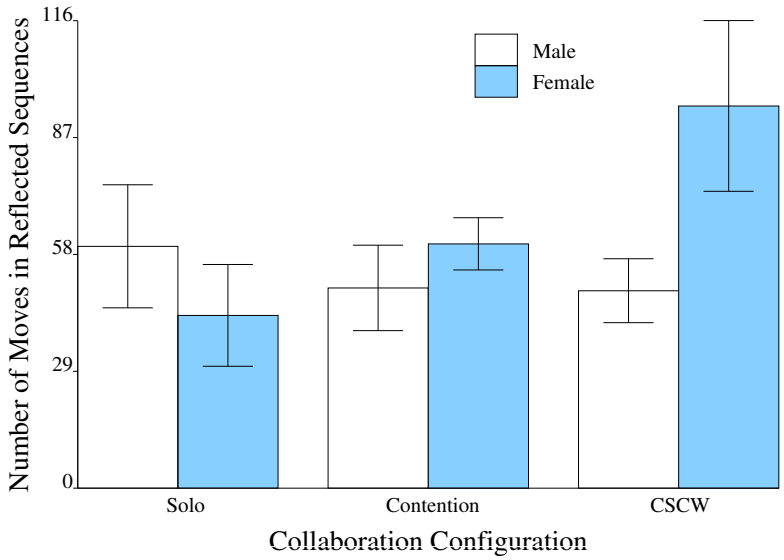


Figure 5.7: Average number of moves in reflected sequences in phase one grouped by collaboration configuration and gender. Anova showed a marginal interaction between these factors ($F_{(2,42)}$ =2.623, p=0.07).

reflected move sequences (20 sequences) whereas participants in the Solo configuration had the lowest (13 sequences, see Figure 5.6). The analysis also showed a marginal interaction between gender and collaboration configuration ($F_{(2,42)}$ =2.6, p=0.083). A post-hoc test confirmed a reliable effect of collaboration for females ($F_{(2,21)}$ =4.4, p<0.05). Female participants had an average of 25 reflected sequences in the CSCW configuration, 17 sequences in the Contention configuration, and 11 moves in the Solo configuration. This is an indication that collaboration is reducing the amount of planning female participants are engaging in (the opposite of what our hypothesis H4 predicted). This result is discussed in Section 5.4.

Number of Moves in Reflected Sequences

An analysis of the number of moves in reflected sequences shows a marginal interaction between gender and collaboration ($F_{(2,42)}$ =2.623, p=0.07). This interaction is shown in Figure 5.7. This interaction appears to be caused by females in the CSCW condition having more moves in reflected sequences than males. This effect was reliable ($F_{(2,21)}$ =4.661, p<0.05).

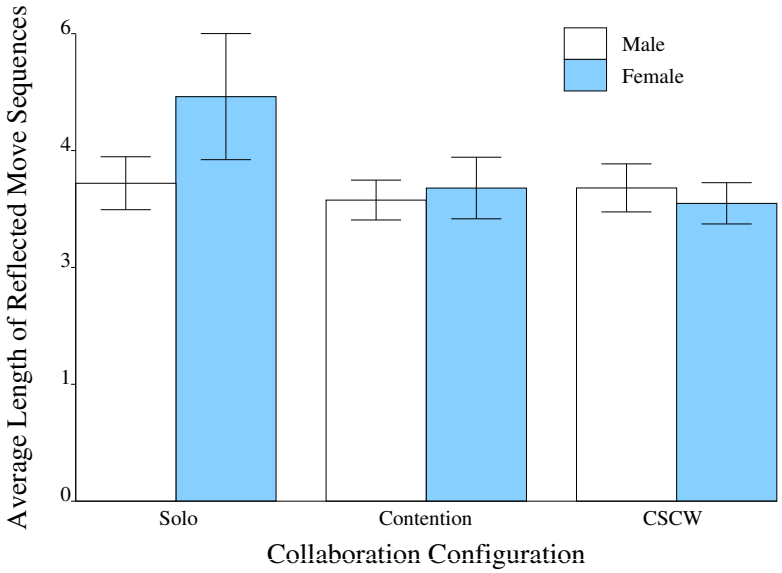


Figure 5.8: Average length of reflected move sequences for each collaboration configuration in phase one. Longer lengths of reflected move sequences is an indication that participants are engaging in more planning. There were neither reliable nor marginal results.

Average Length of Reflected Sequences

An analysis of the average length of reflected move sequences showed no reliable results. This is shown in Figure 5.8.

Percentage of moves in Reflected Sequences

An analysis of the percentage of moves in reflected sequences showed a marginal effect of gender ($F_{(1,42)}$ =3.557, p=0.07). Post-hoc tests showed this effect was marginal for females ($F_{(2,21)}$ =3.0, p=0.07) but unreliable for males ($F_{(2,21)}$ =0.24, p=0.79).

5.3.2 Testing Phase

Time

Although males took less time than females (655 seconds compared with 695 seconds), the effect was not reliable ($F_{(2,42)}$=0.22, p=0.64). There was also an unreliable difference in the time required in each of the three conditions: participants in the CSCW condition required the least

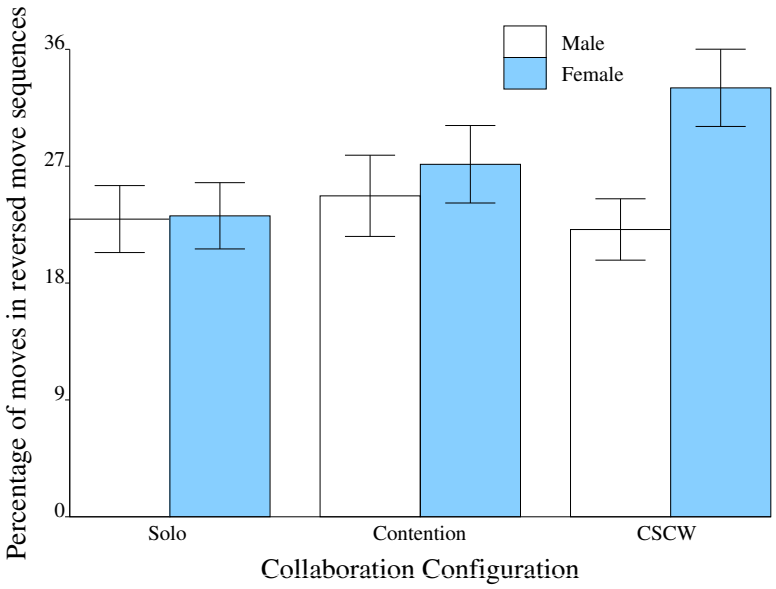


Figure 5.9: Percentage of moves in reversed move sequences in phase one. Shorter percentages of moves in reflected move sequences is an indication that participants are engaging in less situated action. There was a marginal effect of gender ($F_{(1,42)}$ =3.557, p=0.07).

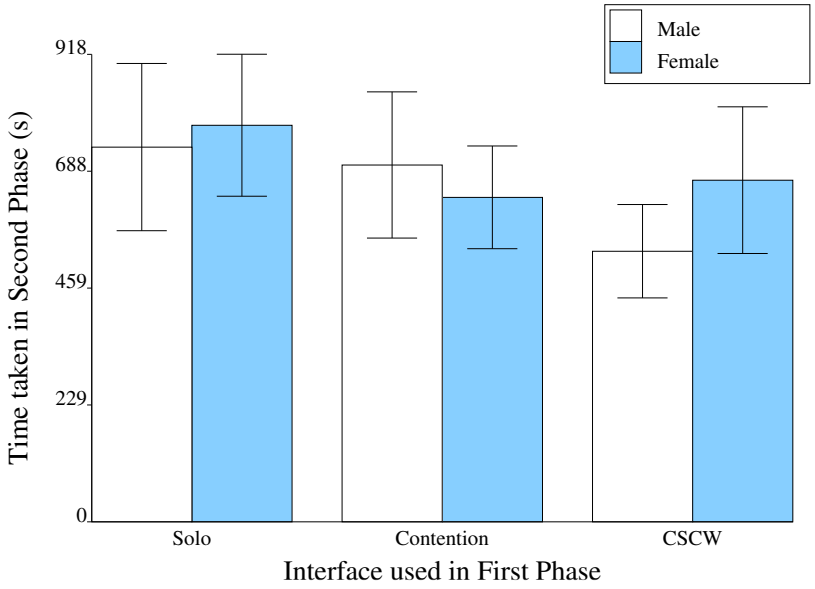


Figure 5.10: Amount of time needed to complete phase two grouped by collaboration configuration and gender. There were no reliable effects of Gender ($F_{(2,42)}$=0.22, p=0.64) or Condition ($F_{(2,42)}$=0.76, p=0.47), and no reliable interaction ($F_{(2,42)}$=0.29, p=0.75).

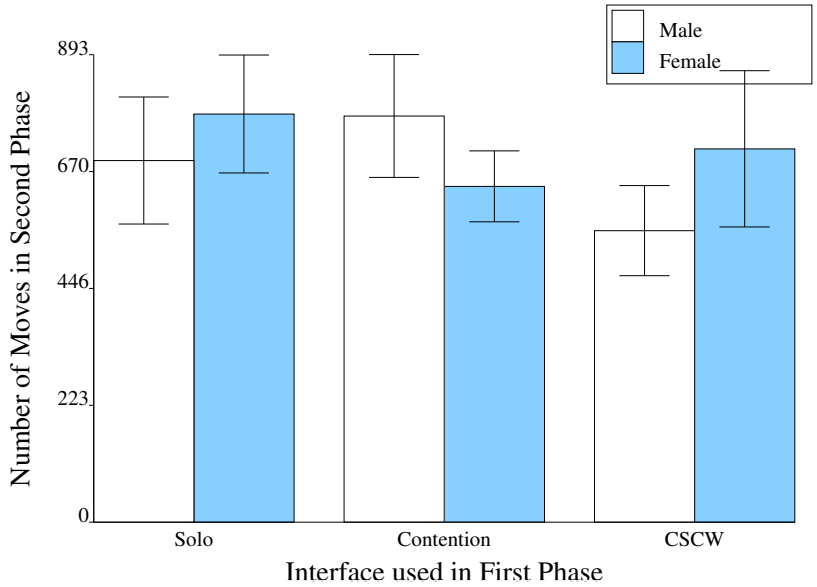


Figure 5.11: Number of moves needed to complete phase two grouped by collaboration configuration and gender. There were no reliable effects of Gender ($F_{(2,42)}$=0.05, p=0.82) or Condition ($F_{(2,42)}$=0.85, p=0.44), and no reliable interaction ($F_{(2,42)}$=0.92, p=0.40.

amount of time to complete the phase (601 seconds), followed by participants in the Contention condition (669 seconds), while participants in the Solo condition required the highest number of moves (757 seconds, $F_{(2,42)}$=0.76, p=0.47). This data is shown in Figure 5.10.

Number of Moves

Although males were making fewer moves than females (711 moves per phase compared with 674), the effect was not reliable ($F_{(2,42)}$=0.05, p=0.82). There was also an unreliable difference in the number of moves made in each of the three conditions: participants in the CSCW condition made the least number of moves (635 moves), followed by participants in the Contention condition (708 moves), while participants in the Solo condition required the highest number of moves (734, $F_{(2,42)}$=0.85, p=0.44). This data is shown in Figure 5.11.

Inter-move Latencies

Males were clicking faster with an average inter-move latency of 0.94 seconds. Females had an average inter-move latency of 1.03 seconds. The effect was reliable ($F_{(1,42)}$ =6.409, p<0.05).

5.4 Discussion

There are many reliable results in the data for phase one. The lack of results in phase two reinforces previous studies of collaboration reviewed in chapter 4: that collaboration does not affect measurable learning outcomes. That is, it neither helps nor hinders learning. The lack of results in phase two also provide reason to believe that collaboration neither promotes nor reduces planning. This is an indication that collaboration does not create a high-cost interface.

Our discussion examines the results from two perspectives. We first investigate indications of task-based learning and planning, and second examine effects of task-performance.

5.4.1 Task-based Learning and Planning

We view learning success as how quickly and in how many moves participants can solve the puzzle after training. Examining the results from phase two, we do not see any reliable effects of collaboration condition or gender. On average, participants learned to solve the puzzle equally well, regardless of which collaboration condition they learned in, and regardless of their gender. More precisely, there was no significant difference in the learning outcomes in each of the two-factors.

O'Hara and Payne argue that an indication of increased planning is an increase of inter-move latencies between training conditions in the training phase [129]. In our evaluation, the inter-move latencies between any of the three collaborative conditions were not reliably different in the first phase. We reject this as an indication that participants in collaborative conditions were doing equal amounts of planning: inter-move latencies in collaborative conditions represent the latency between any two moves, not between two moves by the same participant. That is, one participant in a dyad may have been planning while the other participant was trying out an idea.

When we examine the percentage of moves in reflected sequences, we see that females in the CSCW condition have a reliably higher percentage of moves in reflected sequences than females in the solo condition (Figure 5.9). This is an indication that females in the CSCW condition are tracking for their thoughts with the puzzle interface, either to explain their thoughts to their partner or as an indication of a reduction in planning.

Another explanation of this difference in percentage is that females are fighting for mouse control. If this were the case, we would expect a difference in average reversed sequence length: females who are fighting over mouse control would have a shorter reflected length. Although we saw indications of this effect, the indications were not statistically reliable (see Figure 5.8).

5.4.2 Task Performance

The results reliably show that females collaborating take more moves to complete a phase than females working alone. They also require more time to complete the phase. As the inter-move latencies are not reliably different between the collaborative and non-collaborative conditions, the extra time to solve the puzzle must be due to female participants taking more moves to complete the phase when collaborating.

The reliable data about females (in the first phase) shows that when they collaborate they take more moves to solve a puzzle, have a higher number of reflected move sequences, and have a higher percentage of moves in reflected sequences. This is a strong indication that when collaborating females track their thoughts using the puzzle and engage in less planning.

We suggest three potential explanations for this: the female participants are reversing each other's move sequences, reversing their own moves (this could be an indication that they are uncomfortable working in pairs), or communicating more. If female participants are reversing each other's move sequences or communicating more we would expect to see higher average inter-move latencies as they spend time explaining what they were doing. We did not see this effect, and are left with the conclusion that eleven year old girls reverse their own moves more regularly when working in pairs. Fortunately the discomfort did not reliably affect how well they learned the puzzle.

We note that further post-hoc tests showed the differences to be only significant between the solo and CSCW conditions. Examining the data for the Contention condition we find that the averages for Contention are in-between the Solo and CSCW conditions. This indicates that the Contention condition has elements of both the Solo and CSCW conditions, just not enough differences to be reliably distinguishable from either.

5.5 Limitations of Experiment

There are several limitations in our study. We examined one small puzzle, and while it is unclear how observations of learning in a small bounded puzzle transfers to larger unbound learning tasks, the effects of operator implementation cost on learning do generalise to larger problems, such as air traffic control [56]. Our metrics for learning are crude measures of task performance, and there may have been important learning factors that we failed to measure. Some learning factors include, for example, development of social skills, practice at negotiation, and practice at compromise. Despite these limitations, we believe it important to establish concrete empirical

foundations that attempt to characterise and clarify the relative merits of different modes of CSCL, even within restricted domains such as the one explored in this study.

The major limitation of the puzzle in the context of this thesis is that the puzzle does not directly examine programming behaviour. However, as considered in Section 5.1, we believe that this puzzle shares some features with programming. However, we realise much more work is needed to show transfer between the two domains and to safely extrapolate these results.

5.6 Summary

This chapter used a simple puzzle to look for evidence that collaboration can affect learning. Although we did not see any direct evidence that our interface helps or hinders learning, we did notice that female participants used more moves to solve a puzzle when collaborating than females working alone. This is an effect on performance rather than an effect on learning. An investigation into this result revealed indications that collaborative interfaces cause females to change their problem solving strategy: they use less of a planning approach and perform more of a situated action approach. We believe that this increase in situated actions is a result of females feeling unsure of their planning when collaborating.

Even though our collaborative interfaces did not help learning, they also did not hinder learning. We believe that collaborative computer interfaces still have value as there are other benefits to collaboration that we did not measure or test—like learning to work in a team or development of social skills.

The next chapter introduces Mulspren: our multiple notation programming environment.

Part III

Implementation

Chapter 6

Mulspren

Chapter 3 describes an evaluation of how well children can read and understand computer programs written using different notations. The evaluation found that children can read and understand a conventional-style notation more efficiently than a notation written with an English-like syntax. The evaluation also found that children prefer an English-like notation. These findings led us to reason that providing multiple notations is a good idea because they provides access to both a more efficient notation (conventional) and a preferred notation (English).

This chapter describes our programming environment called Mulspren[1]. Mulspren users program using multiple notations: a conventional-style notation and an English-like notation. Mulspren's notations contains a subset of the features described in Chapter 3: we implemented a subset of features to reduce the number of programming statements to five (a number suggested by a primary school teacher), and to reduce the complex hyperspace that programmers must navigate to understand programs. Another difference between Mulspren's interface and the interface described in the Chapter 3 is that notations are displayed side by side rather than in

[1] MUltiple Language Simulation PRogramming ENvironment

tooltips. We chose this layout for two reasons: so that users can use multiple notations for the writing and watching tasks, and so that different users can examine different notations and move between the multiple notations as they desire.

Chapter 2 describes how multiple notations are used in programming environments and noted several methods for using notations that are not used in current programming environments. Mulspren uses notations in a novel way: users program using dual notations where changes in one notation are immediately reflected in the other notation and users can move seamlessly between the two notations. We made this decision so that users can write using both notations rather than being constrained to only writing using one notation. Mulspren's Language Signature is [RE/WR/$WA_{English-like\ text}$ + RE/WR/$WA_{conventional-style\ syntax}$ + WA_{agents}].

Chapter 5 describes an evaluation of different modes of collaboration. The evaluation describes how participants collaborating using one computer (and contention for the mouse) learned to solve a problem not reliably differently to participants using two computers and two mice (with no contention for the mice). Mulspren uses this result: as input device contention does not reliably affect learning, we do not need to make Mulspren group-aware to lever the potential benefits of collaborative programming in a single-user application. These potential benefits are described in chapter 4

This chapter describes Mulspren. We start with an analysis of the requirements of Mulspren then describe the user interface. Next we illustrate the implementation considerations and finally report on two heuristic evaluations of Mulspren: one using cognitive gulfs as a heuristic tool (as described in section 2.3) and the other using cognitive dimensions (subsection 2.4.1 contains a description of the cognitive dimensions framework).

6.1 Requirements

This section discusses Mulspren's design requirements. We have two overriding design goals: to make programming interesting and to help children use their knowledge of English to learn a conventional-style notation. To help place the requirements in context, we use the second simulation described in Appendix A (on page 177) as an example program. This simulation is a mine game where players are acting as a monster-hunter who is trying to remove all monsters from a mine. In the game, players can only remove monsters when they have "anti-monster power". They receive five seconds of this power when they find and drink from a fountain. A player dies if they touch a monster when they have no anti-monster power. Figure 6.1 shows a

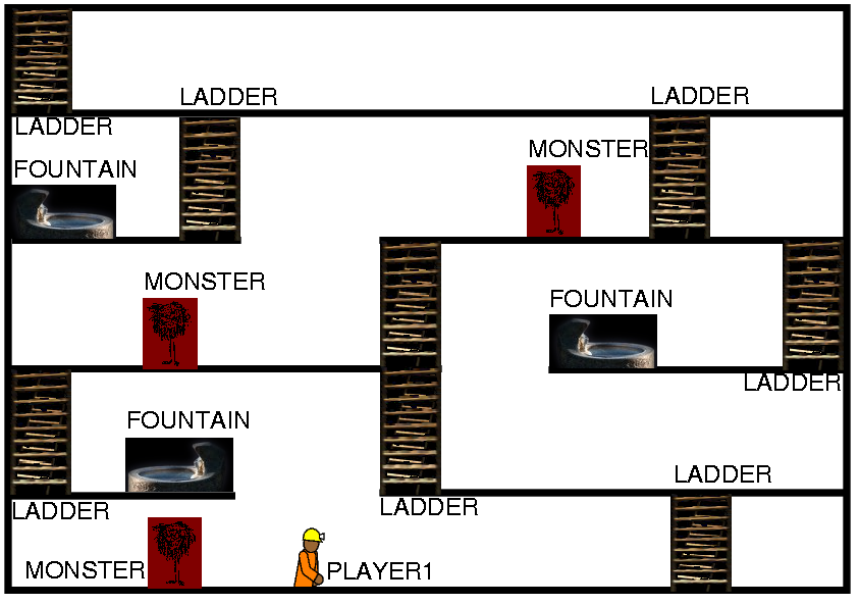


Figure 6.1: The Mine Game: players must move around the mine, gain power by touching fountains, then remove monsters using that power.

snapshot of the game.

This game has several agents (fountains, monsters, and a player agent), some of which are controlled by a user and some of which are controlled by the computer. A programmer of this game has to define how the computer should control the monsters, how the player agent should respond to user-generated events, and what should happen when the agents interact with each other. The obvious programming domain to let children build the example described in the previous section is the domain of 2D visual simulations. These simulations, which are used in many children's programming environments (e.g. [47, 75, 146, 162, 166]), have visual representations of agents in a 2D area and let programmers define how agents respond to events. Events might be user-generated events, interactions between other agents, or even regular computer generated events (such as timers).

One of our goals with this programming environment is to help children transfer their knowledge of English to knowledge of conventional-style computer syntax. This goal constrains our choice of notations in Mulspren: we need one notation to access their knowledge of English and another notation to help children transfer their knowledge to conventional code. The notations in Mulspren contain a subset of the programming statements provided by the English-like notation

and the conventional-style notation used in Chapter 3. To reduce the number of programming statements to five, as suggested by a primary school teacher, we choose to only support enumerated types and we removed the list comprehension features of the notations.

Additionally, we identified several concepts a programming environment for children should include.

Liveness. Cook, Burnett, and Book identify liveness as a good feature of programming environments [32]. They describe liveness as a feature of a programming environment where: immediate feedback is given about syntactic errors, programs can be modified at any time; and visual feedback about program semantics is always available.

Syntax-Directed (Constrained) Editor. Syntax directed editors are editors that use knowledge about syntax and semantics of a programming notation to help a programmer build a program [8, 33, 95, 188]. The editors can do things like highlighting incorrect code, offering type-ahead facilities, and even reducing the possibility of syntactically incorrect programs. Research into the usability of syntax directed editors has found that syntax directed editors have potential to help users [34, 182, 184], but if the environment is to restrictive, expert programmers will dislike the editors [115] — the environment forces the expert programmers to work differently than they usually would.

Poor Learning Environments. Rick, using theatre as an analogy, argues that programming environments for children should be poor [156]. By poor, he means poor in content and lacking intelligence: the user of a programming environment should be forced to create their own content and provide the intelligence. He also argues that there should be multiple representations of underlying concepts.

Mulspren includes all these concepts: it has liveness, uses a syntax-directed editor, and is a poor learning environment.

6.2 User-Interface

This section describes Mulspren's user interface. We first describe the programming domain then describe how programs are structured. Finally we describe the programming constructs available in Mulspren.

Figure 6.2: A screen snapshot of Mulspren: A MUltiple Language Simulation PRogramming ENvironment.

6.2.1 Programming Domain

Mulspren users build 2D visual simulations. We chose this domain for two reasons: first, research has shown that children enjoy writing 2D visual simulations and, second, many other programming environments for children use this domain (these environments include Playground [47], StageCast [168], AgentSheets [151], and Hands [136]). Research describing simulation programming environments describes that children enjoyed the domain and were enthusiastic about writing programs — even to the point of working on their simulations during their break time and staying late after school [57].

Visual simulations contain some (or many) interacting agents, where each agent has at least one visual representation (also called face or picture) and a location in a 2D space. Agents interact with each other and a user, and programmers write code that specifies what happens when agents interact. Pane found that children want to specify the direction and speed that an agent is moving in, where the agent is, and what it looks like [136]. These requirements make the domain of programming visual simulations fit well with object–orientated event–driven programming paradigms.

A sample visual simulation is shown in Figure 6.1.

6.2.2 Structuring Code

Programmers using Mulspren structure their code in an object–oriented event–driven style. Each agent in the simulation is represented as an object. Programmers write event handlers.

Object Oriented: Object orientation is a common programming paradigm, used by many languages including Java, Smalltalk, Python, Objective C, C++, PHP, and even Perl. Users build programs that consist of objects, where an object is the combination of related data and functionality. Objects are run-time instances of classes, where classes statically define object behaviour.

The domain of 2D visual simulations provides a close fit with the object-orientated programming metaphor. Each object in a program is represented as an agent, and objects can be instantiated by dragging them from a class list (or agent template list) onto the simulation area. Encapsulation, or data hiding, is implemented by allowing agents read-only access to other agents' state.

We do not include inheritance support for the reasons outlined by Reppenning and Perrone. They argue inheritance does not map easily onto the domain of visual simulations, users can easily create weak designs that are ontologically unsound, users need icons to represent abstract base classes, and it is too easy to over-generalise and create simulations with unexpected behaviour [153].

Event Handlers: When agents interact, events are triggered and are passed to an agent by invoking a method on that agent. In this programming style users need only define event methods and the code to react to the event—the event loop and the code to dispatch events are built into Mulspren. Many programming environments for children are event driven, and research indicates that this programming style is a natural style for children to express programs [136].

Agents in Mulspren respond to seventeen system events as well as any user defined events. The system events define how agents interact with a user, how agents interact with the simulation, and how agents interact with other agents. The events are summarised in Table 6.1.

6.2.3 Programming Constructs

Mulspren contains five programming constructs. We decided on five constructs after discussion with a primary school teacher, and selected the five so that beginner programmers could write interesting programs. The five statements we implemented are: selection, assignment, method call, agent creation, and agent destruction. The statements are described in Tables 6.2 through 6.6. We chose these statements by examining the types of statements present in other programming environments and selecting the five that provided the greatest functionality while remaining simple. There were six statements we identified as possible statements in Mulspren:

Assignment Statements. These statements let users change agent state (or variables). Each variable in Mulspren is associated with an object, and encapsulation is used to let objects examine the values of other objects variables, but objects can only change their own variables. All variables in Mulspren are enumerated types.

Selection Statements. Selection statements provide a way for programmers to specify control

Events generated by a user	
Event	**When Generated**
`Mouse Enter`	When the user's mouse moves from outside to inside an agent.
`Mouse Exit`	When the user's mouse moves from inside to outside an agent.
`Mouse Down`	When the user presses down the mouse on an agent.
`Mouse Up`	When the user releases the mouse on an agent.
`Mouse Click`	When the user clicks on an agent.

Events generated by Mulspren	
Event	**When Generated**
`Clock Tick`	Every second.
`Simulation Start`	When the simulation is started.

Events generated by interactions with agents	
Event	**When Generated**
`Am Near`	When an agent becomes near, or is no longer near, another agent.
`Am Not Near`	
`Am Left Of`	When an agent becomes left of, or is no longer left of, another agent.
`Am Not Left Of`	
`Am Right Of`	When an agent becomes right of, or is no longer right of, another agent.
`Am Not Right Of`	
`Am Above`	When an agent becomes above, or is no longer above, another agent.
`Am Not Above`	
`Am Below`	When an agent becomes below, or is no longer below, another agent.
`Am Not Below`	

Table 6.1: List of system generated events can respond to.

flow. The condition in a selection statement in Mulspren can test if any variable is equal or not equal to a particular value in an enumeration.

Method Call Statements. These statements let programmers invoke a method on another (or the same) object. Method calls are all asynchronous, meaning that the statement returns immediately and the method is put on a queue of things for the Mulspren scheduler to execute.

Agent Creation and Destruction Statements. These statements let programmers remove and add agents to a simulation.

Looping Statements. Looping statements let programmers specify that a certain piece of code should be performed a certain number of times or until a particular condition is false. We did not have looping statements in Mulspren for two reasons: first, they can be simulated using method calls and recursion, and second, we believe that they are the most complex statement type for children to understand.

To further aid programming, statements are inserted into a program (and modified once in the program) using a drag and drop syntax directed editor — Mulspren programs are created using only the mouse. A syntax directed editor increases environmental constraints. Avoiding the keyboard reduces coordination problems and increases programming accessibility: users do not need to use a soft keyboard and can even use advanced user input devices such as touchscreens or eye tracking devices. During implementation we found several other places a syntax-directed editor is useful: if statements can have the negation of the condition after the `else` keyword (see Table 6.2), and we can replace the `this` keyword with `my` or `me` to create a grammatically correct English-like notation.

In chapter 3, we wrote a program to produce the English-like notation by parsing the conventional-style notation and translating it into English. Mulspren's approach is different: Mulspren stores a program internally as an abstract syntax tree and produces both the conventional-style and English-like notations on demand. Modifications to the abstract syntax tree are immediately reflected on the computer's display using the Model-View-Controller paradigm [101]. This approach has several advantages over the translation approach. First, it is possible to create alternate notations based on the internal syntax tree (examples include flow charts or Nassi-Shneiderman diagrams) where modifications to these notations are automatically reflected in all

Conventional	English-Like

`if <object>.<variable>`
`=/!= value then`
`...`
`else`
`...`
`continue`
	`if <object>'s <variable>`
	`is/is not`
	`<value> then`
	`...`
	`otherwise (ObjectRef's variable`
	`is not/is value)`
	`...`
	`continue`

Table 6.2: Both representations of a selection statement. Users can select <object>, <variable>, <value>, and the comparison operator (= or !=) using drop down lists. Users fill out the list of statements in the true or false part of the statement (represented by '...' in the table) by dragging and dropping new statements into the if statements.

Conventional	English-Like
`<object>.<method>()`	`send <message> to <object>`

Table 6.3: Both representations of a method call statement. Users select <message> and <object> using drop-down lists.

other notations. Second, program visualisation is straightforward to implement. To execute the program we simply evaluate the abstract syntax tree. The abstract syntax tree (more precisely statements in the abstract syntax tree) notify interface components that a particular statement is being executed, and those interface components update their visual representation. We believe this approach to watching is easily extended to alternate multiple notations.

Conventional	English-Like
`<object>.<variable>= <value> [ (+|-) <object>.<variable> ]`	`set <object>'s <variable> to <value> [ (+|-) <object>'s <variable> ]`

Table 6.4: Both representations of an assignment statement

Conventional	English-Like
`new <class> ( <direction> )`	`create a new <class> <direction> [of] me`

Table 6.5: Both representations of a agent creation statement

Conventional	English-Like
`<object>.remove()`	`remove <object>`

Table 6.6: Both representations of an agent destruction statement

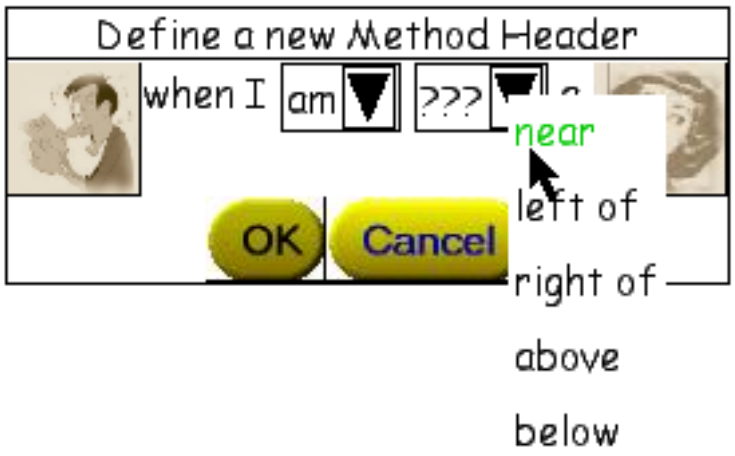
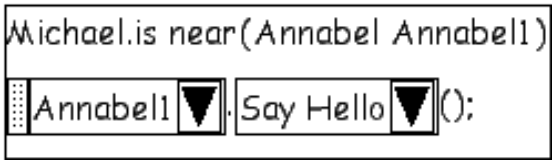


Figure 6.3: Dialogue to specify which locality to use

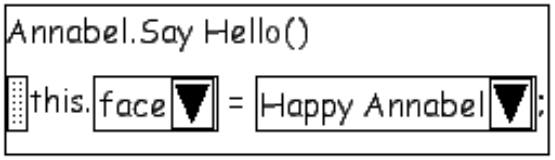
(a) English-like code



(b) Conventional code

Figure 6.4: English-Like Notation Window. A user has specified a `Say Hello` action to be performed on an Annabel agent whenever a Michael agent is near the Annabel agent.

(a) English-like code



(b) Conventional code

Figure 6.5: An Annabel agent's face changes to a happy representation whenever it receives a `Say Hello` message. The light grey handle (green when in colour) indicates that the statement is currently being executed.

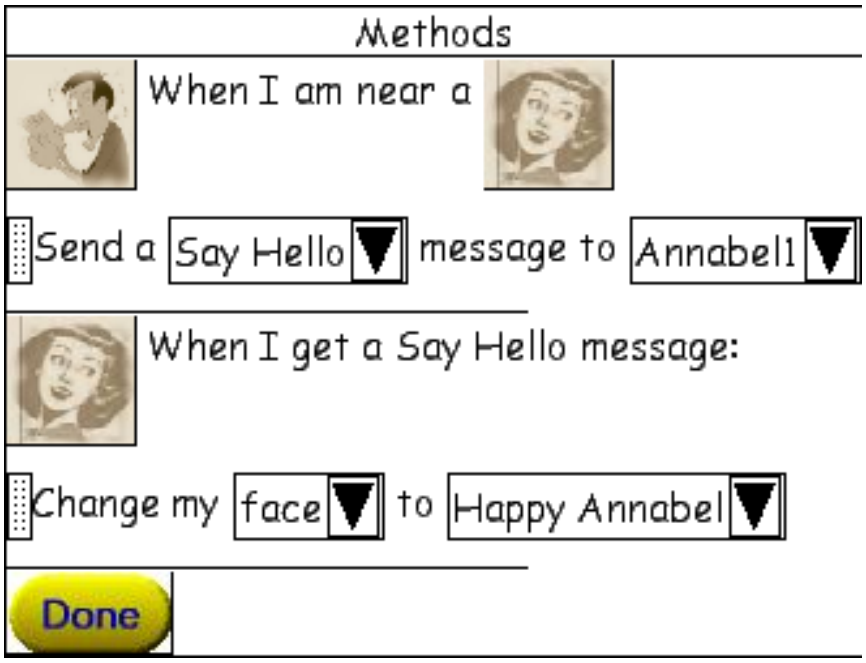


Figure 6.6: All methods for a simple simulation.

6.3 Implementation

6.3.1 Language and API

Mulspren is built on SDL [164], a multi-platform low-level graphics programming layer. Part of constructing Mulspren required building a C++ GUI toolkit on top of SDL. Before building Mulspren, we constructed a paper prototype [155]. The prototype is shown in Figure 6.7.

6.3.2 Software Design

Mulspren contains about 19 thousand lines of C++ and uses the Model-View-Controller paradigm to structure the code (Figure 6.8, [101]). The Model stores all information about the current program, and uses a combination of the composite pattern and the interpreter pattern to structure the code (Figure 6.9). In this combination of patterns each statement type has an associated class. When a program is parsed, each statement is represented by an object, and executing the program involves only invoking the execute method on the object representing the main function. Mulspren also uses the observer pattern to automatically update the multiple representations.

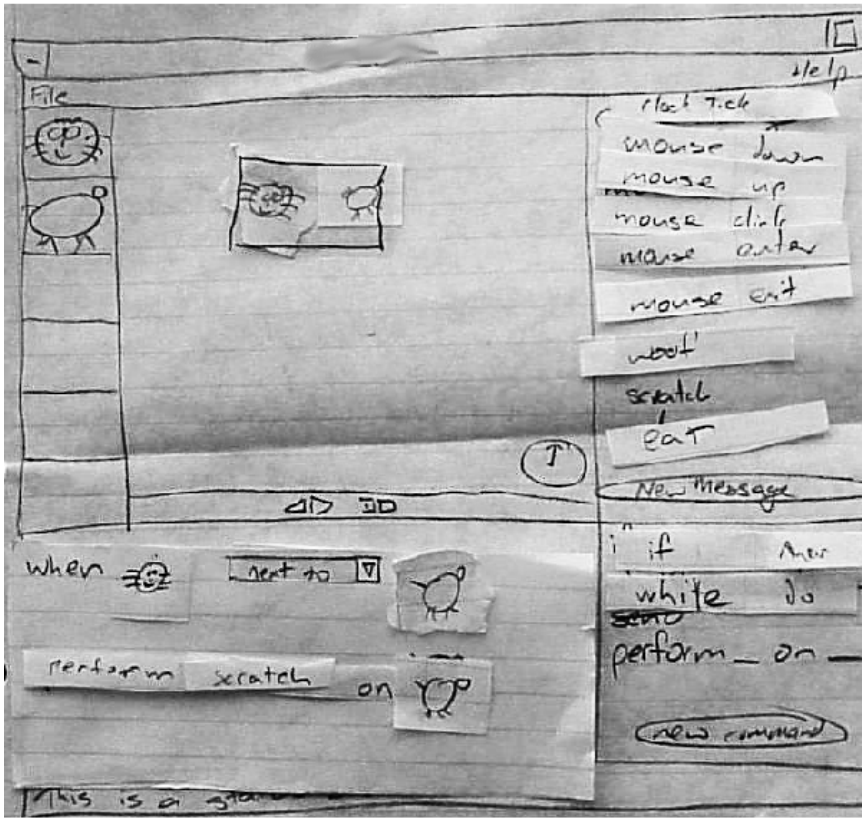


Figure 6.7: An early paper prototype. This prototype was used to determine the constraints of the syntax directed editor.

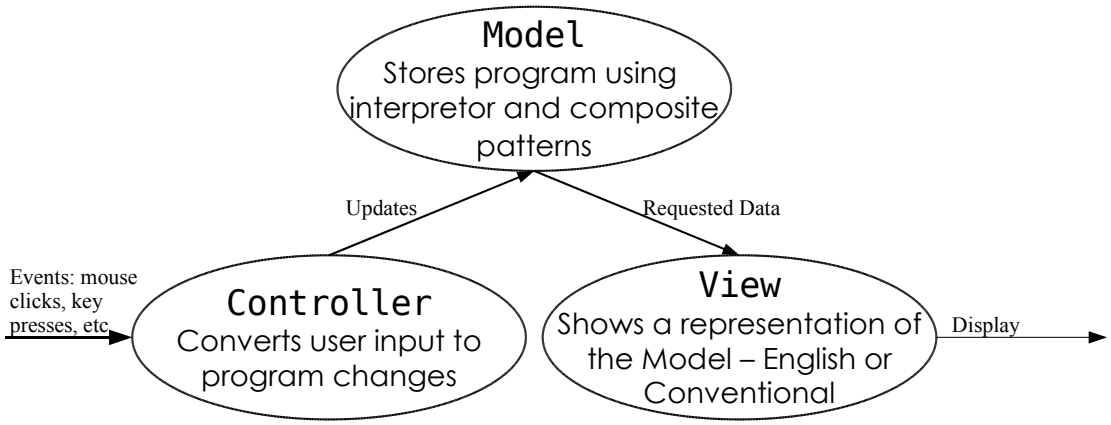


Figure 6.8: The Model-View-Controller paradigm [101]. The separation of data storage from data presentation allows multiple different presentations of the data, a design that powerfully supports multiple representations.

Each representation registers its interest in a part of the model (the programming statements that are being displayed), and when the model is changed, the model sends a notification message to the representation and the representation refreshes the on-screen display. We combined these two patterns, the observer and the interpreter, so that the model also sends notification messages when it is being executed. Views then have the information they need to include visualisation information (or watching support). Currently, there are two notation-views, one for the English-like notation and one for the conventional notation, however the separation of model and view makes adding new notations to the system relatively simple, and should make centralisation of shared data relatively easy using a groupware toolkit.

6.3.3 Implementation Limitations

Mulspren has several implementation limitations.

First, it provides little support for structuring and understanding highly complex programs. This limitation is present to encourage advanced users to program in more complex and powerful programming environments. Additionally, we expect users who have the skills to develop highly complex programs will likely become annoyed by the novice user supports (syntax di-

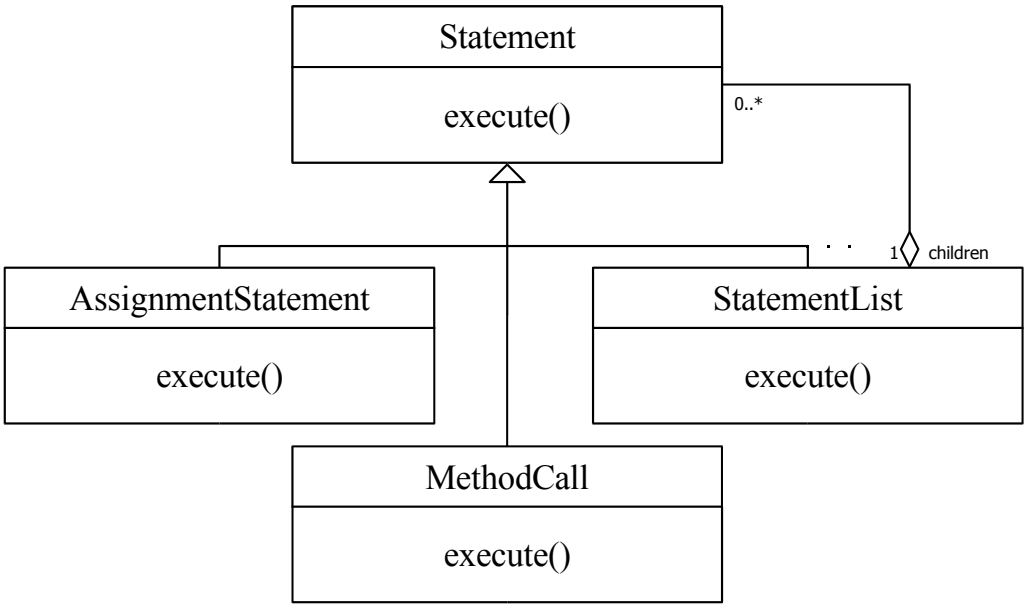


Figure 6.9: Structure of the Model. We used a combination of the composite and interpreter patterns to structure the model [52]. There is a base statement class from which all actual statement types extend, and a statementList class which has references to zero or more statements. Each statement can execute itself, meaning that executing a function is as simple as calling execute on the statementList for that function.

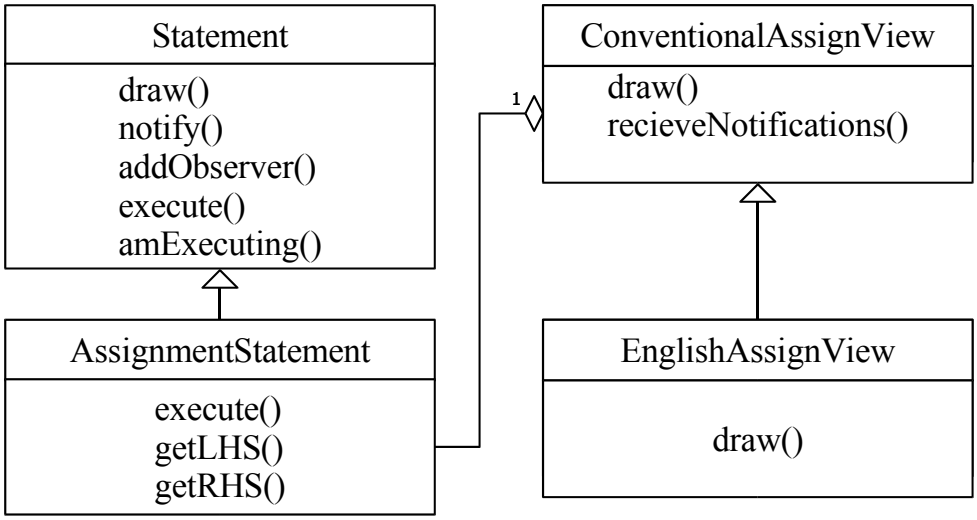


Figure 6.10: Structure of the Views. We used the observer pattern so the two notation views could automatically update themselves whenever the model is changed [52]. Each statement would send notifications to any observers of the statement (both the views) whenever the statement changed. The statement could change if a user modified a statement parameter or if the statement was executing. The views could automatically update themselves whenever the model changed, ensuring consistency between all views. This design decision also meant that programs can be edited as they are executed.

rected editor, multiple representations, no integer support) and move to different programming environments. However because Mulspren is a programming environment for novice programmers, we care less about advanced programmers than we care about novice programmers. Thus, Mulspren does not need features commonly found in programming environments like structuring complex code; searching for particular parts of a program, and an examination of how the environment features scale to huge programs. We leave this examination for developers who wish to build programming environments for expert users.

Second, all method calls in Mulspren are asynchronous. When a piece of code calls a method, that method is put on a queue of methods that are waiting to be executed and control is immediately returned to the method that made the call. It would be interesting to examine the difference that different method call semantics has on understandability and usability of a children's programming environment, but we leave this for future work.

6.4 Evaluation

This section describes two heuristic analyses of Mulspren. The first analysis uses the cognitive gulfs framework presented in chapter 2, and the second analysis uses the cognitive dimensions framework. When performing the cognitive dimensions analysis, we use the modifications proposed in subsection 2.4.1 (on page 56).

6.4.1 Cognitive Gulfs

The Gulf of Expression

To re-cap, the gulf of expression is the cognitive difference between a users mental model of desired program behaviour and the notation in which a user must express their program to the computer. In subsection 2.3.1 we identified three factors that can influence the risk of a gulf of expression. These factors are: the task domain, using read-only notations, and environmental constraints. Mulspren provides mechanisms to reduce the risk of the gulfs created by each these factors.

Task domain: This factor can create a gulf of expression when users want to write programs that are hard to express in a particular programming environment. One domain that researchers have found is close to users' mental model of programming is the domain of

visual simulations. Visual simulations have been found by other researchers to provide motivation for learner programmers to program [57].

Mulspren programs are visual simulations. Additionally, Mulspren users can use images from their own image library as agents. These two features should help reduce the cognitive distance between the problem space and Mulspren's notations.

Read-only notation: Our taxonomy found that programming environments that provide a read-only representation risk creating a gulf of expression. The gulf is created because users will read their program using one notation and then have to modify their program using a different notation. Mulspren avoids creating this gulf by ensuring all readable notations are editable. Mulspren allows editing of notations at all times—even when a program is executing. This should further reduce the risk of a gulf of expression as users can edit a program whenever they want.

Environmental constraints: Our taxonomy, presented in chapter 2, found that environmental constraints are a good way to help novice end-users write programs. Environmental constraints are functions of a program editor to avoid syntactical errors and provide information about what possible actions a user can make while programming.

Mulspren provides a syntax directed editor where users program using drag and drop mechanisms. This editor ensures that users can not write syntactically incorrect programs. Also, when a user creates a statement, the essential form of the statement is inserted into their program. For example, if a user creates a method call statement, the views will show: `send a ?  message to ?` and `?.?` (where `?` is a drop down list with all possibilities in the list). Users can also move entire statements around their program using drag and drop. This design avoids syntactical errors while providing users help structuring programs.

The Gulf of Representation

The gulf of representation is the amount of effort a user must expend to predict program behaviour, or the difference between a user's mental model of their program and how the program is represented to them for reading. In subsection 2.3.2 we identified three factors that can cause a gulf of representation: using multiple notations for reading, using no notation for reading, and

using different notations for reading and writing. Mulspren provides mechanisms to reduce the risk of the gulfs created by all these factors.

No notation for reading: This factor can create a gulf of representation because users have no means of predicting what will happen when their program is executed. To help people predict what will happen when their program is executed, Mulspren provides two notations. Users can use whichever notation is closest to their mental model and predict program behaviour based on that notation.

Different notations for reading and writing: This factor creates the risk of a gulf of representation when users must write their program in one notation and read their program using a different notation. It creates a gulf of representation because users must switch mental models of their program when performing the different tasks—something they must do regularly if they are editing a program.

Mulspren users are never required to switch notations for reading or writing. Although users can switch notations, Mulspren lets users make the choice when to change notations. By moving the decision of which notations to use onto the user, Mulspren reduces the risk of this factor creating a gulf of representation.

Multiple notations for reading: This factor creates the risk of a gulf of representation when users are presented with multiple inconsistent representations of their program. It creates the risk because users have more trouble predicting program behaviour: they are not sure which representation they should base their prediction on.

Mulspren avoids this gulf by letting users move seamlessly between the both notations whenever the user chooses to do so, and the notations are kept strictly consistent at all times. The strict consistency means that predictions of program behaviour should be the same when made from either of Mulspren's notations. If users make different predictions from the multiple notations it is an indication that either: their mental model of their program is incorrect (and they should modify their model using a visualisation); or an indication that they do not understand one of the notations (and they can use a visualisation to determine which notation they do not understand and increase their understanding).

The Gulf of Visualisation

The gulf of visualisation is the difference between a users mental model of program behaviour and what their program does as it executes. Section 2.3.3 identified two factors that can lead to a gulf of visualisation: program behaviour and program execution. The factor program behaviour risks creating a gulf of visualisation when a user does not have the information they need to understand what their program is doing while it executes. The other factor, program execution, risks creating the gulf when the user is not presented with enough information to understand what the programming environment is doing while their program is executing. Mulspren provides mechanisms to reduce the risk of the gulfs created by these factors.

Program Behaviour: This factor can lead to a gulf of visualisation when a user does not have enough information to understand the relationship between the program representation and their program behaviour. Mulspren provides two supports to reduce this factor: statement and object visualisation.

- ↪ **Statement visualisation:** To help users understand which program statements are causing which behaviour, Mulspren shows users which statement is being executed by turning on a green light in front of the statement when it is being executed. If the statement cannot be executed (for example, when the user has not chosen a value in an assignment statement) then a red light is displayed. See Figure 6.5 for a screen snapshot of Mulspren providing statement visualisation.

  This visualisation helps reduce the risk of this factor creating a gulf of visualisation by providing user-interface support to help a user map between the program representation and the program behaviour.

- ↪ **Object visualisation:** To help users understand how properties of their objects are changing, Mulspren provides support to see any hidden properties and experiment changing any properties to evaluate any change in program behaviour. Users inspect the state of objects by right-clicking on them and viewing a property sheet (see Figure 6.12 for a screen snapshot of a property sheet). As an object's properties change this is immediately reflected in the property sheet, and users may change the values of properties directly on the property sheet using a drop-down list to select possible values.

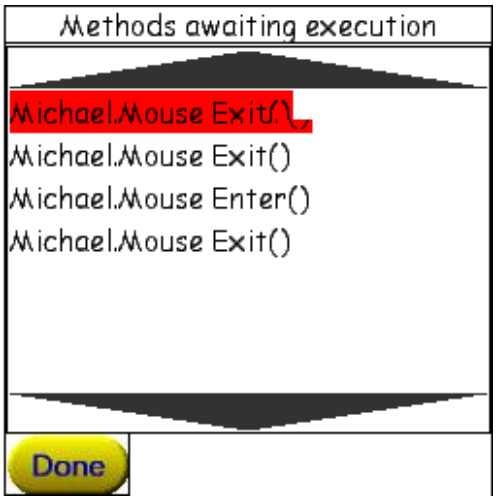


Figure 6.11: Method visualisation. This dialogue shows the list of methods that are in the queue awaiting execution. The highlighted method is currently being executed.

Program Execution: This factor can create a gulf of visualisation when a programming environment executes a user's program differently than a user anticipates. To help users understand how Mulspren is executing their program, Mulspren provides an animation of the program scheduler. Users can see which method calls are awaiting execution and in which order they will be executed. Figure 6.11 shows a screen snapshot of the program scheduler.

6.4.2 Cognitive Dimensions

The second heuristic evaluation we performed is a cognitive dimensions analysis of Mulspren [65]. This evaluation examines thirteen dimensions of programming notations to try to identify problems that users might have when using the programming notations. In subsection 2.4.1 we described several extensions to this framework to examine the relationships between dimensions in a multiple notation environment. The extensions are used in this section as part of this heuristic evaluation of Mulspren.

Abstraction Gradient. The Abstraction Gradient dimension analyses the minimum and maximum levels of abstraction, and looks at the ability for a programmer to abstract fragments of a program. The extension for multiple notations examines the relationship between the

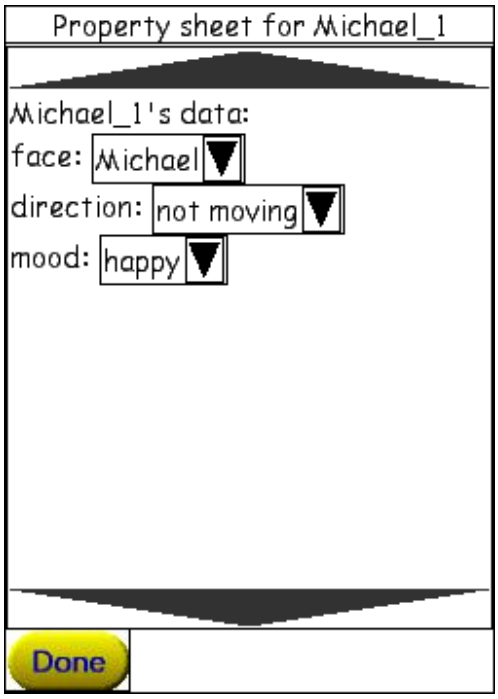


Figure 6.12: Object Visualisation. This dialogue shows the current states of variables for an object. Users can edit the values. `Face` and `direction` are standard variables, while `mood` is a user-defined variable and is defined in an external file.

minimum and maximum levels of abstraction in both notations and examines the effect of creating an abstract fragment of a program in one notation on the other notation.

Mulspren is abstraction-hating: users can not create abstract fragments of a program and Mulspren only provides two levels of abstraction: a function and a statement. This design decision was made to minimise the abstractional difference in Mulspren's two notations: program fragments in either notation can be translated to program fragments in the other notation using exactly the same level of abstraction.

Closeness of Mapping. This dimension examines the mapping between the problem world and the syntax and semantics of the programming notation. The extension for multiple notations also examines the closeness of mapping between the multiple notation: how much cognitive effort a user must expend when switching notations.

Research by Pane found that children naturally express programs in English and use an event driven paradigm to describe visual simulations. As Mulspren's English notation is similar to the notation described by Pane, we argue that Mulspren's English notation is close to a user's model of a program world. However, due to the extra syntactic features of the conventional-style notation, we believe that this notation is further away from a user's model of a program world.

The 1:1 mapping between statements in the two notations reduces cognitive load on a user.

Consistency. This dimension refers to "*a particular form of guessability: when a person knows some of the language structure, how much of the rest can be guessed successfully?*" [65] As Mulspren uses a syntax-directed editor that makes visible all possibilities for creating a program, it would be easy for users to guess the syntax of both notations. However, to properly determine consistency we should conduct formal usability studies of both the notations. We leave this for future work.

Diffuseness/Terseness. This dimension refers to both the number of symbols needed to create a program and the amount of screen real-estate needed to display a program. Unfortunately, Green and Petre do not give much information to evaluate the usability of a notation using this dimension. However, we can state that both of Mulspren's notations are textual notations (rather than visual notations), meaning that they have many symbols using little screen real-estate. Both notations are terse.

Error-proneness. This dimension refers to the consequences of making an error and the ease in which an error can be found. Green and Petre argue that textual programming notations are error-prone: it is easy to make small mistakes and hard to track these mistakes down. Mulspren helps users avoid making these mistakes by providing an editor that does not allow syntactic errors. Naturally, mistakes of program design are still possible (as they are in any programming environment), but the removal of syntax errors helps reduce the error-proneness of Mulspren: it becomes harder for users to make careless mistakes.

The removal of syntax errors also means that Mulspren does not need to show syntax errors of one notation in the other notation.

Hard Mental Questions. Some programming environments force users to play complex mental games to specify their programs. Green and Petre give this example:

> *Unless it is not the case that the lawn-mower is not in the shed, or if it* is *the case that the oil is not in the tool-box* and *the key is not in its hook, you will need to cut the grass…* [65] (original emphasis retained.)

This cognitive dimension, hard mental questions, refers to the programming games that users must perform to convert their mental model of desired program behaviour into the semantic structures provided by the programming environment. One way to understand this dimension is it is similar to the closeness of mapping dimension, but examining semantic structures rather than syntactic structures.

Both of Mulspren's notations are semantically based on the notation developed by Pane and Myers [134]. To create this notation, Pane and Myers asked children to describe behaviour of agents in a visual simulation and then examined the semantic features in the children's description and designed a notation based on these semantic features. As Mulspren's notations have similar semantic features to the language in which children naturally express programs, there is little risk of children having to solve hard mental problems when programming.

Hidden Dependencies. This dimension refers to how many relationships between components exist where the relationship is not fully visible. An example of a hidden dependency is a method in a conventional programming notation. Although it is easy to see which methods are called by a particular method, it is much harder to examine which methods call a

particular method: a question that programmers might want to know when changing the behaviour of a method (to avoid breaking their program in unexpected ways).

Mulspren does have hidden dependencies. While it is easy to look at a method in Mulspren and determine which methods it invokes, Mulspren provides no support for tracing method calls backwards and discovering which methods invoke a method that a programmer is interested in. There are two types of support that Mulspren could add to help reduce the effect of this dimension: Mulspren could provide a reversible debugger, so users can run their program backwards and see which methods are calling which other methods; and Mulspren could provide functionality for browsing code backwards. Vista is a sample environment that has this functionality: users can click on a method and see a list of all methods that invoke that method they clicked on [18].

Premature Commitment. This dimension refers to programming decisions that a programmer must make before having all information necessary to make the decision. Green and Petre give examples of this dimension referring to: commitment to layout, where a programmer must chose a location to place a visual component before knowing how it is going to interact with other components; commitment to connections, where a programmer must think heavily about future connections between components to avoid making their program look like spaghetti; and commitment to order of creation, where the order in which components are created affects the program execution.

This dimension refers primarily to factors influencing visual programming notations. Mulspren's notations are both textual notations and do not suffer the same commitment problems: statements can be moved around easily, the connections are all linear from top to bottom, and the order of statement creation does not affect the order the statements are executed. Additionally, Mulspren avoids premature commitment by allowing modification of programs that are not completely specified. For example, a user can add an `if` statement to a program then add several `assignment` statements and `method call` statements to the true or false parts of the `if` statement before specifying the condition in the `if` statement. An screen snapshot of a Mulspren program where a user has done this is shown in Figure 6.13.

Progressive Evaluation. This dimension refers to the ability of a programming environment to execute incomplete programs. Executing incomplete programs helps users evaluate their

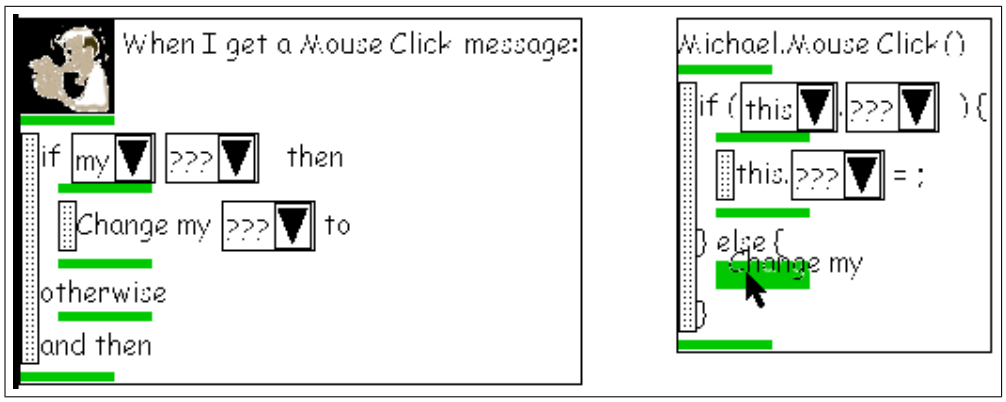


Figure 6.13: An incomplete `if` statement. A user has started filling out statements in the `true` and `false` parts of the if statement without defining the condition to test. This example shows how Mulspren avoids premature commitment when users are writing code.

progress at frequent intervals.

Mulspren supports progressive evaluation. Users can send messages to agents that don't know what to do with the messages and Mulspren can even execute programs with statements that are not completely defined—Mulspren will simply skip over that statement and move to the next statement (Mulspren will highlight the statement in red to indicate that there was a problem with the statement).

Role Expressiveness. This dimension refers to the ease in which a notation can be read and understood. This dimension can be contrasted with Hard Mental Questions and Closeness of Mapping: these dimensions refer to the ease in which a program can be written.

In chapter 3, we performed an evaluation of how quickly and accurately children can read and understand Mulspren's notations. We found that children could read the conventional notation faster than the conventional notation, but the two notations did not provide a reliable difference in accuracy. The evaluation provides confidence that children can understand programs written in either of Mulspren's dual notations and that children do not incur a high cost of translating between the two notations.

Secondary Notation and Escape from Formalism. This dimension refers to the ability of a programming notation to contain extra information; information that is ignored when the program is executed. This information could include comments about what a piece of

code should do, formatting of code to enable additional understanding; or even choice of statement type and choice of how programming statements are grouped. Allowing secondary notation and an escape from formalism provides an extra channel for programmers to communicate with someone who is trying to understand a program.

Mulspren allows neither secondary notation nor escape from formalism. We made this decision because our users are novice users. In an evaluation of how readership skills affect programming, Petre argues that while experts have much to gain from secondary notation, novices "*might benefit from a more constrained system in which secondary notation is minimised, in order to reduce the richness and the potential for mis-curing and misunderstanding*." [141]

Viscosity: Resistance to Local Change . This dimension refers to the amount of work a programmer must expend to perform a minor change to their program. Green and Petre note that the viscosity of a programming notation can often be related to the editor used to modify the notation rather than any actual properties of the notation.

Mulspren programs are easily changed and have low viscosity. First, statements can be modified at any time by selecting new values from drop-down lists. Second, statements can be removed by dragging them out of the method they are in and new statements can be added by dropping a new statement into a program. Third, statements can also be moved easily by dragging a statement to a new place. These changes can be made while a program is executing.

Visibility and Juxtaposability. This dimension measures two aspects of how users can browse visual programming notations. The first aspect is Visibility and refers to the number of steps needed to display a particular item in the program. The second aspect is Juxtaposability, and refers to the ability of the programming environment to show two different parts of a program on the display at the same time.

In Mulspren, there are two ways for a user to view a particular item in the program. Users can either open the All Methods dialogue (Figure 6.6) and scroll to the method they want or they can open the method for editing, an operation that can take one to four user interactions (however, we should note, that users can edit the methods in the All Methods dialogue directly). In Mulspren, users who want to view two different methods at the same

time can search for one method in the All Methods dialogue and show the other method in the dual notation area. Users can not view three or more methods at the same time.

This flat hierarchy of methods (users do not need to drill down through a tree-like structure to find information), combined with an ability to juxtapose different methods, should help users of Mulspren who are trying to understand and debug computer programs.

6.5 Summary

This chapter described a visual programming environment, Mulspren. Mulspren programs are built using only the mouse, and are represented using two notations: an English-like notation and a conventional-style notation. Chapter 3 showed that children can read and understand conventional-style notation faster than than the English-like notation with no reliable effect on accuracy, yet children preferred reading the English-like notation. Mulspren provides both notations to gain the efficiency of the conventional-style notation but provide a notation that children prefer. Mulspren's two notations contain a subset of the features described in chapter 3. We chose this subset to reduce the complex hypertextual space created by a program: to make programs easier to navigate and understand.

Mulspren's design was influenced by other chapters. Chapter 4 described that collaborative programming can be useful, and chapter 5 performed an evaluation showing that programming environments do not need to be group-aware to gain from the effects of collaboration. Chapter 2 described how programming environments risk creating several cognitive gulfs. Mulspren was designed to minimise the risk of these gulfs.

To evaluate Mulspren's effectiveness as a programming environment we performed two different heuristic evaluations. The first evaluation was a cognitive gulfs evaluation. In this evaluation, we used our theory of multiple-notation programming environments described in chapter 2. The second evaluation was based on a set of heuristics for programming notations. The heuristics were originally developed by Green and Petre [65], but we used an extended version for multiple notation programming environments. The extension is described in subsection 2.4.1.

Chapter 7

Conclusion and Future Work

This thesis presented an investigation into collaborative and multiple notation programming environments for children. First, we investigated multiple-notation programming. This investigation identified three fundamental programming activities and used these three activities to review many end-user programming environments. The review found evidence for several cognitive gulfs that can hinder programming. It also found that using different notations for the same activity had potential to create knowledge transfer and aid learning. Based on the review we performed an evaluation of using multiple notations for the reading task. The evaluation found that children could answer questions faster (with the same level of accuracy) about programs written in a conventional-style programming notation than programs written in an English-like programming notation. It also found evidence that children preferred the English-like notation. Next, we investigated collaborative programming to determine which modes of collaboration a programming environment should support. This investigation showed no reliable difference in measurable learning outcomes between children working together on one computer and children working together on two computers using a groupware application. Based on the analysis of programming environments and the two evaluations, we developed a multiple notation programming environment for children. The environment is a dual notation programming environment: both notations are displayed at all times and the children can move between and edit the representations as they wish.

There were two themes running through the thesis. The first theme examined multiple notations. It developed two sets of usability heuristics for multiple programming environments

and applied the heuristics to a new type of programming environment. We developed the set of heuristics using two mechanisms: first, by classifying multiple notation programming environments and determining which usability problems were due to the use of notations, and, second, by extending an existing set of heuristics from a single-notation domain to a multiple notation domain. Language Signatures lie at the core of our classification scheme: they precisely specify how notations are used in a programming environment and let us group programming environments that use notations in a similar way.

The second theme examined collaboration. It identified different modes of collaboration and evaluated how different modes might affect learning. Unfortunately we could not use Language Signatures to classify usability problems in collaborative programming environments: there was not enough supply of collaborative programming environments to determine common usability problems.

More precisely, the contributions of this thesis are:

↪ We introduce a method to concisely describe how different notations are used in programming environments. We call the description a *Language Signature*. Language Signatures are used to classify programming environments based on how the environment uses notations rather than on surface features of the notations. For example, Language Signatures for two programming environments might be the same even if one environment uses physical blocks for programming and the other environment uses textual symbols.

↪ We use Language Signatures to classify, review, and assess end-user programming environments. During the review we identify usability problems related to how the programming environments used notations. These problems are taken from literature describing usability studies of the environments.

↪ We identify several cognitive gulfs that exist in programming environments, describe how these gulfs can hinder user's programming experience, and catalogue ways of using programming notations that lead to these gulfs. We call these *factors*, and we used the factors as heuristics to analyse the usability of multiple programming notations in a programming environment. These gulfs and factors were constructed from the notation-related usability problems discovered during the review of Language Signatures.

↪ Through an empirical study we show that children can understand computer programs rep-

resented with multiple notations and that children understand code written in a conventional-style faster than code written in English, with no reliable difference in accuracy.

↪ Through another empirical study we show that children create the same measurable learning outcomes whether they are collaborating with one computer and a single user application or two computers and a groupware application. This contribution has implications for computer application designers wanting to increasing learning by levering collaboration: the designers do not need to implement collaborative support in the application to gain potential benefits from collaboration.

↪ We describe a programming environment called Mulspren. As Mulspren's Language Signature is different to every other programming environment's Language Signature we can confidently state that Mulspren uses notations differently to every other programming environment. To evaluate Mulspren we used two sets of heuristics: cognitive gulfs and an extension of Green and Petre's cognitive dimensions framework. Both are designed to evaluate multiple-notation programming environments.

7.1 Future Work

There is much scope for future work.

The evaluation described in chapter 3 found that children can understand programs written using a conventional-style notation faster with no reliable difference in accuracy than programs written using an English-like notation. We need to investigate this result further. In particular, we need to determine whether the children were faster because there is less information to parse or because they have a stronger mental model. We also need to determine whether this difference in efficiency is present in our other two programming activities: writing and watching. Particular research questions include:

↪ While we have justified the usability of Mulspren using two different sets of usability heuristics (cognitive dimensions and the gulfs we developed in chapter 2), heuristics can never identify all usability problems. To overcome this limitation, future work needs to examine the usability of Mulspren.

↪ Additional further work needs to examine how children interact collaboratively with multiple notation programming environments. In particular, we believe that a collaborative learning evaluation would be an excellent area for further research.

↪ Chapter 3 found that conventional-style notations were more efficient than English-like notations. We need to determine whether the difference in efficiency of conventional-style notations was due to a longer parse time of English code or due to a stronger mental model. There are several ways to answer this question. The first is to modify the English-like and conventional-style notations so that they contain the same number of English words and re-run the evaluation. The second is to re-run the evaluation using adults. Adults can read English faster than children so the difference in parse-time should be less noticeable. Re-running the evaluation with adults would also tell us if the result transfers to adults.

↪ Related to the previous point, we found that conventional-style notations were more efficient than English-like notations. We need to determine if the difference in efficiency transfers to the writing and watching activities. To answer this question we need to run evaluations concentrating on the effects of multiple notations on the writing and watching activities. Unfortunately when people are watching or writing a program they are also reading the program: to figure out what the program is doing and to track down bugs. The evaluation described in chapter 3 gives us a baseline for each notation and we can use this baseline to examine the effect of a notation on just the writing or watching tasks.

References

[1] ABNETT, C., STANTON, D., NEALE, H., AND O'MALLEY, C. The effect of multiple input devices on collaboration and gender issues. In *Euro-CSCL '01* (Universiteit Maastricht, Maastricht, Netherlands, March 20–21 2001), pp. 29–36.

[2] AINSWORTH, S. A functional taxonomy of multiple representations. *Computers and Education 33*, 2/3 (1999), 131–152.

[3] AINSWORTH, S., AND VAN LABEKE, N. Using a multi-representational design framework to develop and evaluate a dynamic simulation environment. In *Dynamic Information and Visualisation Workshop* (Tuebingen, July 2002).

[4] ANDERSON, J. J. ChipWits: Bet You Can't Build Just One. *Creative Computing* (December 1985), 76–79.

[5] ANDERSON, J. R. *The Architecture of Cognition*. Harvard University Press, 1983.

[6] ANDERSON, J. R., AND CORBETT, A. T. Acquisition of lisp programming skill. In *Foundations of Knowledge Acquisition: Cognitive Models of Complex Learning* (Hingham, MA, 1992), S. C. amd A. Meyrowitz, Ed., Kluwer.

[7] APPLE COMPUTER INC. *Apple Macintosh HyperCard user's guide*. Apple Computer Inc, 1987.

[8] AREFI, F., HUGHES, C. E., AND WORKMAN, D. A. Automatically generating visual syntax-directed editors. *Commun. ACM 33*, 3 (1990), 349–360.

[9] BAECKER, R. M., GRUNDIN, J., BUXTON, W. A. S., AND GREENBERG, S., Eds. *Readings in Human Computer Interaction: Toward the Year 2000*. Morgan Kaufmann Publishers, 1995, ch. 11.

[10] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[11] BENFORD, S., BEDERSON, B. B., ÅKESSON, K., BAYON, V., DRUIN, A., HANSSON, P., HOURCADE, J. P., INGRAM, R., NEALE, H., O'MALLEY, C., SIMSARIAN, K. T., STANTON, D., SUNDBLAD, Y., AND TAXÉN, G. Designing storytelling technologies to encouraging collaboration between young children. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (2000), ACM Press, pp. 556–563.

[12] BLAYE, A., AND LIGHT, P. Collaborative problem solving with hypercard: The influence of peer interaction on planning and information handling strategies. In *[131]*. Springer-Verlag, 1995, pp. 3–22.

[13] BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. *The UML Reference Guide*. Addison-Wesley, 1999.

[14] BORNING, A. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Computer-Human Interaction 3*, 4 (October 1981), 353–387. Also appears in [59].

[15] BORNING, A. Graphically defining new building blocks in thinglab. *Human-Computer Interaction 2*, 4 (1986), 269–295. Also appears in [59].

[16] BORNING, A. H. *Thinglab–a constraint-oriented simulation laboratory.* PhD thesis, Stanford University, 1979.

[17] BRIDGELAND, D. Simulacrum: A system behaviour example editor. In *Visual Languages and Applications*, T. Ichikawa, E. Jungert, and R. R. Korfhage, Eds. Plenum, 1990, pp. 191–202.

[18] BROWN, J., GRAHAM, T. N., AND WRIGHT, T. The Vista Environment for the Coevolutionary Design of User Interfaces. In *Human Factors in Computing Systems: CHI '98 Conference Proceedings* (USA) (1998), pp. 376–383.

[19] BRUCKMAN, A. Programming for fun: Muds as a context for collaborative learning. In *Proceedings National Educational Computing Conference* (Boston, MA, June 1994).

[20] BRUCKMAN, A. *MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids*. PhD thesis, Massachusetts Institute of Technology, 1997.

[21] BRUCKMAN, A. MOOSE Goes to School: A Comparison of Three Classrooms Using a CSCL Environment. In *Proc. Computer Supported Cooperative Learning 1997 (CSCL '97)* (Toronto, Canada, 1997), pp. 20–26.

[22] BRUCKMAN, A., AND EDWARDS, E. Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language. In *Human Factors in Computing Systems: CHI '99 Conference Proceedings* (USA) (1999), pp. 207–214.

[23] BRUCKMAN, A., JENSEN, C., AND DEBONTE, A. Gender and Programming Achievement in a CSCL Environment. In *Proc. Computer Supported Cooperative Learning 2002 (CSCL '02)* (2002), pp. 119–127.

[24] BURNETT, M., ATWOOD, J., DJANG, R., GOTTFRIED, H., REICHWEIN, J., AND YANG, S. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming 11*, 2 (March 2001), 155–206.

[25] BURNETT, M., CHEKKA, S. K., AND PANDEY, R. FAR: An End-User Language to Support Cottage E-Services. In *IEEE Symposia on Human-Centric Languages and Environments* (Stresa, Italy, September 2001), pp. 195–202.

[26] CHAMBERLIN, D. D., KING, J. C., SLUTZ, D. R., TODD, S. J., AND WADE, B. W. Janus: An interactive system for document composition. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation* (Portland, Oregon, United States, 1981), pp. 82–91.

[27] COCKBURN, A., AND BRYANT, A. Leogo: An equal opportunity user interface for programming. *Journal of Visual Languages & Computing 8*, 5–6 (1997), 601–619.

[28] COCKBURN, A., AND BRYANT, A. Cleogo: Collaborative and multi-paradigm programming for kids. In *APCHI'98: Asia Pacific Conference on Computer Human Interaction. Japan. July 15–17* (1998), IEEE Computer Society Press, pp. 187–192.

[29] COCKBURN, A., AND SMITH, M. Hidden messages: Evaluating the effectiveness of code elision in program navigation. *Interacting with Computers: The Interdisciplinary Journal of Human-Computer Interaction. 15*, 3 (2003), 387–407.

[30] COCKBURN, A., AND WILLIAMS, L. The costs and benefits of pair programming. In *Proc. eXtreme Programming and Flexible Processes in Software Engineering XP2000* (2000).

[31] CONWAY, M., AUDIA, S., BURNETTE, T., DURBIN, J., GOSSWEILER, R., KOGA, S., LONG, C., MALLORY, B., MIALE, S., MONKAITIS, K., PATTEN, J., SHOCHET, J., STAAK, D., STOAKLEY, R., VIEGA, J., WHITE, J., WILLIAMS, G., COGROVE, D., CHRISTIANSEN, K., DELINE, R., PIERCE, J., STEARNS, B., STURGILL, C., AND PAUSCH, R. Alice: Lessons Learned from Building a 3D System for Novices. In *Human Factors in Computing Systems: CHI 2000 Conference Proceedings* (USA) (April 2000), pp. 486–493.

[32] COOK, C., BURNETT, M., AND BOOM, D. A Bug's Eye View of Immediate Visual Feedback in Direct-Manipulation Programming Systems. *Empirical Studies of Programmers* (October 1997).

[33] COOK, P., AND WELSH, J. Incremental parsing in language-based editors: user needs and how to meet them. *Software Practice & Experiance 31*, 15 (2001), 1461–1486.

[34] COOPER, S., DANN, W., AND PAUSCH, R. Teaching Objects-first In Introductory Computer Science. In *SIGCSE* (2003).

[35] COX, P. T., AND PIETRZYKOWSKI, T. Using a pictorial representation to combine dataflow and object-orientation in a language independent programming mechanism. In *Proceedings International Computer Science Conference* (Hong Kong, 1988), pp. 695–704. Also appears in [59].

[36] CYPHER, A. *Watch What I do: Programming by Demonstration*. MIT Press, 1993, ch. 9: Eager: Programming Repetitive Tasks by Demonstration.

[37] DECORTE, E., LINN, M. C., MANDL, H., AND VERSCHAFFEL, L., Eds. *Computer-Based Learning Environments and Problem Solving*. Springer-Verlag, 1992.

[38] DILLENBOURG, P., BAKER, M., BLAYE, A., AND O'MALLEY, C. The evolution of research on collaborative learning. In *Learning in Humans and Machines: Towards an Interdisciplinary Learning Science*, P. Reimann and H. Spada, Eds. Pergamon, 1996, pp. 189–211.

[39] DISESSA, A. A., AND ABELSON, H. Boxer: a Reconstructible Computational Medium. *Communications of the ACM 29*, 3 (September 1986), 859–868.

[40] DRUIN, A., Ed. *The Design of Children's Technology*. Morgan Kaufmann Publishers, 1999.

[41] DUISBERG, R. A. Animation using temporal constraints: An overview of the animus system. In *Visual Programming Environments, Paradigms and Systems*, E. P. Glinert, Ed. IEEE Computer Society Press Tutorial, 1990, pp. 484–590.

[42] EDAL, M. The tinkertoy graphical programming environment. In *IEEE Proceedings COMPSAC* (Chicago, Illinois, 1986), pp. 466–471. Also appears in [59].

[43] EHRET, B. D. Learning Where to Look: Location Learning in Graphical User Interfaces. In *Human Factors in Computing Systems: CHI 2002 Conference Proceedings* (USA) (Minneapolis, Minnesota, april 2002), pp. 211–218.

[44] ELLERSHAW, S., AND OUDSHOORN, M. Program visualization - the state of the art. Tech. Rep. TR 94-19, Department of Computer Science, University of Adelaide, 1994.

[45] ELSHOUT, J. J. Formal Education Versus Everyday Learning. In *[37]*. Springer-Verlag, 1992, pp. 5–17.

[46] ERICSSON, K. A. Problem-Solving Behaviour with the 8-puzzle II: Distribution of Latencies. Tech. Rep. 432, Department of Psychology, University of Stockholm, 1974.

[47] FENTON, J., AND BECK, K. Playground: An Object Oriented Simulation System with Agent Rules for Children of All Ages. In *Proc. OOPSLA '89* (New Orleans, Louisiana, United States, 1989), pp. 123–137.

[48] FINKELSTEIN, A., GABBAY, D., HUNTER, A., KRAMER, J., AND NUSEIBEH, B. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering 20*, 8 (1994), 569–578.

[49] FINZER, W. F., AND GOULD, L. Programming by rehearsal. In *[59]*. IEEE Computer Society Press Tutorial, 1990, pp. 356–366.

[50] FISCHER, G. Shared Understanding, Informed Participation, and Social Creativity — Objectives for the Next Generation of Collaborative Systems. In *Proc. COOP 2000, Sophia, Antipolis, France* (May 2000). Invited Talk.

[51] FURNAS, G. New Graphical Reasoning Models for Understanding User Interfaces. In *CHI '91* (New Orleans, Louisiana, United States, 1991), pp. 71–79.

[52] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley, 1995.

[53] GEIGER, C., MUELLER, W., AND ROSENBACH, W. SAM — An Animated 3D Programming Language. In *IEEE Symposium on visual languages* (Nova Scotia, Canada, September 1998), pp. 228–235.

[54] GILLIGAN, D. An Exploration of Programming by Demonstration in the Domain of Novice Programming. Master's thesis, Victoria University of Wellington, August 1998.

[55] GILMORE, D. J. Interface Design: Have we got it wrong? In *Proc. INTERACT '95, Lillehammer, Norway* (1995).

[56] GILMORE, D. J. The Relevance of HCI Guidelines for Educational Interfaces. In *[77]* (1998).

[57] GILMORE, D. J., PHEASEY, K., UNDERWOOD, J., AND UNDERWOOD, G. Learning graphical programming: An Evaluation of KidSim. In *Proc. INTERACT '95, Lillehammer,*

Norway (1995), K. Nordby, P. Helmersen, D. Gilmore, and S. A. Arnesen, Eds., Chapman and Hall.

[58] GILNERT, E. P., AND TANIMOTO, S. L. Pict: An interactive graphical programming environment. In *[59]*. IEEE Computer Society Press Tutorial, 1990, pp. 265–264.

[59] GLINERT, E. P., Ed. *Visual Programming Environments, Paradigms and Systems*. IEEE Computer Society Press Tutorial, 1990.

[60] GOLIGHTLY, D. Harnessing the Interface for Domain Learning. In *CHI '96 Doctoral Consortium* (Vancouver, British Columbia, Canada, 1996). http://www.acm.org/sigchi/chi96/proceedings/doctoral.htm.

[61] GRAHAM, T. C. N., AND GRUNDY, J. C. External requirements of groupware development tools. In *Proceedings of the IFIP TC2/TC13 WG2.7/WG13.4 Seventh Working Conference on Engineering for Human-Computer Interaction* (Crete, Greece, 1999), Kluwer, B.V., pp. 363–376.

[62] GRAHAM, T. C. N., MORTON, C., AND URNES, T. ClockWorks: Visual programming of component-based software architectures. *Journal of Visual Languages & Computing 7*, 2 (June 1996), 175–196.

[63] GRAHAM, T. C. N., URNES, T., AND NEJABI, R. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proceedings of the 9th annual ACM symposium on User interface software and technology* (Seattle, Washington, United States, November 1996), ACM Press, pp. 1–10.

[64] GRAYSON, J. E. *Python and Tkinter Programming*. Manning Publications Co., January 2000.

[65] GREEN, T., AND PERTE, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing 7* (1996), 131–174.

[66] GREENBERG, S., Ed. *Computer supported cooperative work and groupware*. Computer and People Series, Academic Press, London, 1991.

[67] GRUNDY, J., AND HOSKING, J. Engineering plug-in software components to support collaborative work. *Software - Practice and Experience 32*, 10 (August 2002), 983–1013.

[68] GUINDON, R., KRASNER, H., AND B.CURTIS. A model of cognitive processes in software design: An analysis of breakdowns. In *Proceedings of Interact'87 - 2nd IFIP Conference on Human-Computer Interaction* (Stuttgart, Germany, September 1987).

[69] GUTWIN, C., AND GREENBERG, S. Design for individuals, design for groups: tradeoffs between power and workspace awareness. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work* (Seattle, Washington, United States, 1998), ACM Press, pp. 207–216.

[70] GUTWIN, C., AND GREENBERG, S. The Effects of Workspace Awareness Support on the Usability of Real-Time Distributed Groupware. *ACM Transactions on Computer-Human Interaction 6*, 3 (September 1999), 243–281.

[71] GUTWIN, C., STARK, G., AND GREENBERG, S. Support for workspace awareness in educational groupware. In *The first international conference on Computer support for collaborative learning* (Indiana Univ., Bloomington, Indiana, United States, 1995), Lawrence Erlbaum Associates, Inc., pp. 147–156.

[72] HALBERT, D. C. *Programming by Example*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkley, June 1984.

[73] HANCOCK, C. Flogo: A Robotics Programming Language for Learners. In *HCC01 Special Event—Children's Programming Odyssey* (Stresa, Italy, September 2001).

[74] HANCOCK, C. Toward a Unified Paradigm For Constructing and Understanding Robot Processes. In *IEEE Symposia on Human-Centric Languages and Environments* (Arlington, Virginia, September 2002), pp. 107–109.

[75] HARADA, Y., AND POTTER, R. Fuzzy Rewriting — Soft Program Semantics for Children. In *IEEE Symposia on Human-Centric Languages and Environments* (Auckland, New Zealand, 2003), pp. 39–46.

[76] HILL, J., AND GUTWIN, C. Awareness support in a groupware widget toolkit. In *Proceedings of the 2003 ACM Conference on Group Work (Group'03)* (Sanibel Island, FL, 2004), pp. 258–267.

[77] HIRSCHBUHL, J. J., AND BISHOP, D., Eds. *Computers in Education*, 8th ed. Dushkin/McGraw-Hill, 1998.

[78] HOLZNER, S. *Eclipse*. O'Reilly, 2004.

[79] HOOGSTRATEN, K. *Alleen of met zijn tweeën*. PhD thesis, University of Amsterdam, 1976. "Alone or Pairwise. Five field experiments with programmed material". Dutch text with English summary.

[80] ICHIKAWA, T., JUNGERT, E., AND KORFHAGE, R. R., Eds. *Visual Languages and Applications*. Plenum, 1990.

[81] INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. Back to the future: the story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Atlanta, Georgia, United States, 1997), ACM Press, pp. 318–326.

[82] INGALLS, D. H. H. Design principles behind smalltalk. *Byte Magazine* (August 1981).

[83] INKPEN, K., BOOTH, K. S., KLAWE, M., AND UPITIS, R. Playing together beats playing apart, especially for girls. In *The first international conference on Computer support for collaborative learning* (Indiana Univ., Bloomington, Indiana, United States, 1995), Lawrence Erlbaum Associates, Inc., pp. 177–181.

[84] INKPEN, K., GRIBBLE, S., BOOTH, K., , AND KLAWE, M. Give and take: Children collaborating on one computer. In *Human Factors in Computing Systems: CHI '95 Conference Proceedings* (Denver, CO, USA, May 7–11) (1995), pp. 258–259.

[85] INKPEN, K., MCGRENERE, J., BOOTH, K. S., AND KLAWE, M. The effect of turn-taking protocols on children's learning in mouse-driven collaborative environments. In *Proceedings of the conference on Graphics interface '97* (Kelowna, British Columbia, Canada, 1997), Canadian Information Processing Society, pp. 138–145.

[86] INKPEN, K. M., LING HO-CHING, W., KUEDERLE, O., SCOTT, S. D., AND SHOEMAKER, G. B. "This is Fun! We're All Best Friends and We're All Playing.": Supporting Children's Synchronous Collaboration. In *Proc. Computer Supported Cooperative Learning 1999 (CSCL '99) at Stanford* (1999).

[87] ISSROFF, K., SCANLON, E., AND JONES, A. Two empirical studies of computer-supported collaborative learning in science: methodological and affective implications. In *Proc. Computer Supported Cooperative Learning 1997 (CSCL '97)* (Toronto, Canada, December 1997), pp. 117–123.

[88] JACKSON, R. N., AND FINZER, W. F. *Watch what I do: programming by demonstration*, vol. 1. MIT Press, 1993, ch. 13: The Geometer's Sketchpad: Programming by Geometry, pp. 292–307.

[89] KAHN, K. Drawings on Napkins, Video-Game Animation, and Other Ways to Program Computers. *Communications of the ACM 39*, 8 (1996), 49–59.

[90] KAHN, K. A Computer Game To Teach Programming. In *Proc. of the National Educational Computing Conference* (1999).

[91] KAHN, K. Generalizing by Removing Detail. *Communications of the ACM 43*, 3 (March 2000), 104–106.

[92] KAHN, K., AND SARASWAT, V. Complete visualizations of concurrent programs and their executions. In *Proceedings of the IEEE Visual Language Workshop.* (Skokie, Illinois, 1990), pp. 7–15.

[93] KAKFOGETIES, A. Theme-based literate programming. Master's thesis, Department of Computer Science, University of Canterbury, 2002.

[94] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*. Prentice Hall, 1978.

[95] KHWAJA, A. A., AND URBAN, J. E. Syntax-directed editing environments: issues and features. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing* (New York, NY, USA, 1993), ACM Press, pp. 230–237.

[96] KIMURA, T. D., CHOI, J. W., AND MACK, J. M. Show and Tell: A Visual Programming Language. In *Visual Programming Environments: Paradigms and Systems*, E. P. Glinert, Ed. IEEE Computer Society Press, 1990, pp. 397–413.

[97] KNUTH, D. Literate programming. *The Computer Journal 27*, 2 (1984), 91–111.

[98] KOPACHE, M. E., AND GLINERT, E. P. C^2: A mixed textual/graphical environment for c. In *IEEE Proceedings Workshop on Visual Languages* (Pittsburgh, PA, USA, 1988), pp. 231–238. Also appears in [59].

[99] KOSCHMANN, T. Paradigm shofts and instructional technology. In *CSCL: Theory and Practice*, T. Koschmann, Ed. Lawrence Erlbaum Associates, Inc, 1996, ch. 1, pp. 1–23.

[100] KOSCHMANN, T. Dewey's contribution to the foundations of cscl research. In *Proc. Computer Supported Cooperative Learning 2002 (CSCL '02)* (Boulder, Colarado, USA, January 2002), G. Stahl, Ed., pp. 17–22.

[101] KRASNER, G., AND POPE, S. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (August/September 1988), 26–49.

[102] KURLANDER, D. *Watch what I do: programming by demonstration*. MIT Press, 1993, ch. 12: Chimera: Example-based Graphical Editing.

[103] LEWIS, C. Nopumpg: Creating interactive graphics with spreadsheet machinery. In *Visual Programming Environments: Paradigms and Systems*, E. P. Glinert, Ed. IEEE Computer Society Press Tutorial, 1990, pp. 526–546.

[104] LIBERMAN, H. Tinker: Example-based programming for artificial intelligence. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence IJCAI* (Vancouver, August 1981), p. 1060.

[105] LIEBERMAN, H. *Watch what I do: programming by demonstration*, vol. 1. MIT Press, 1993, ch. 2: Tinker: A Programming By Demonstration System For Beginning Programmers, pp. 49–64.

[106] LIEBERMAN, H. *Watch what I do: Programming by Demonstration*. MIT Press, 1993, ch. 16: Monderain: A Teachable Graphical Editor.

[107] LIEBERMAN, H., AND FRY, C. Bridging the Gulf Between Code and Behaviour in Programming. In *Human Factors in Computing Systems: CHI '95 Conference Proceedings* (Denver, CO, USA, May 7–11) (1995), pp. 480–486.

[108] LOCKHEED, M., AND HALL, K. Conceptualising sex as a status characteristic: Applications to develop leadership training strategies. *Journal of Social Issues 32*, 3 (1976), 111–124.

[109] MACLEAN, A., CARTER, K., LÖVSTRAND, L., AND MORAN, T. User-Tailorable Systems: Pressing the Issues with Buttons. In *Human Factors in Computing Systems: CHI '90 Conference Proceedings* (Seattle, WA, USA) (1990), pp. 175–182.

[110] MALONEY, J. H., AND SMITH, R. B. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology* (Pittsburgh, Pennsylvania, United States, 1995), ACM Press, pp. 21–28.

[111] MCDANIEL, R. G., AND MYERS, B. A. Getting More Out Of Programming By Demonstration. In *Human Factors in Computing Systems: CHI '99 Conference Proceedings* (USA) (1999), pp. 442–449.

[112] MCDANIEL, R. G., AND MYERS, B. A. Gamut: Creating Complete Applications Using Only Programming-by-Demonstration. Awaiting publication, http://www-2.cs.cmu.edu/~amulet/papers/#gamut, 2000.

[113] MCGRENERE, J., INKPEN, K., BOOTH, K., , AND KLAWE, M. Experimental design: Input device protocols and collaborative learning. Tech. Rep. 96-11, Department of Computer Science,University of British Columbia, 1996.

[114] MILLER, L. A. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal 20*, 2 (1981), 184–215.

[115] MINÖR, S. Interacting with structure-oriented editors. *Int. J. Man-Mach. Stud. 37*, 4 (1992), 399–418.

[116] MYERS, B., MCDANIEL, R., AND KOSBIE, D. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of INTERCHI '93* (Amsterdam, April 1993), pp. 293–300.

[117] MYERS, B. A. Creating interaction techniques by demonstration. In *Visual Programming Environments: Paradigms and Systems*, E. P. Glinert, Ed. IEEE Computer Society Press Tutorial, 1990, pp. 378–387.

[118] MYERS, B. A. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages & Computing 1* (1990), 97–123.

[119] MYERS, B. A. Graphical techniques in a spreadsheet for specifying user interfaces. In *Human Factors in Computing Systems: CHI '91 Conference Proceedings* (New Orleans, LA, USA) (April 28–May 2 1991).

[120] MYERS, B. A. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology* (UIST '91, Hilton Head, SC, USA, Nov. 11–13) (Nov 11–13 1991).

[121] MYERS, B. A. *Watch what I do: programming by demonstration*, vol. 1. MIT Press, 1993, ch. 10: Garnet: Uses of Demonstrational Techniques, pp. 218–236.

[122] NARDI, B. A. *A Small Matter Of Programming*. MIT Press, 1993.

[123] NELSON, G. Juno, a constraint-based graphics system. In *ACM Proceedings Computer Graphics (SIGGRAPH)* (1985), pp. 235–243. Also appears in [59].

[124] NIELSEN, J. *Usability Engineering*. Morgan Kauffman, 1993.

[125] NORMAN, D. *The Psychology of Everyday Things*. London: Basic Books, 1988.

[126] NOSEK, J. T. The Case for Collaborative Programming. *Communications of the ACM 41*, 3 (March 1998), 105–108.

[127] OBJECTIME LIMITED, 340 MARCH ROAD, KANATA, ONTARIO, CANADA, K2K 2E4. ObjecTime Developer User Guide, August 1998. 5.2 edition.

[128] O'HARA, K. P., AND PAYNE, S. J. The Effects of Operator Implementation Cost on Planfulness of Problem Solving and Learning. *Cognitive Psychology 35* (1998), 34–70.

[129] O'HARA, K. P., AND PAYNE, S. J. Planning and the user interface: the effects of lockout time and error recovery cost. *International Journal of Human-Computer studies 50* (1999), 41–59.

[130] O'MALLEY, C. Designing computer systems to support peer learning. *European Journal of Psychology of Education 7*, 4 (1992), 339–352.

[131] O'MALLEY, C., Ed. *Computer Supported Collaborative Learning*. Springer-Verlag, 1995.

[132] OUSTERHOUT, J. K. *An Introduction to Tcl and Tk*. Reading, MA: Addison-Wesley, 1993.

[133] PAIGE, R., OSTROFF, J., AND BROOKE, P. A test-based agile approach to checking the consistency of class and collaboration diagrams. In *UK Software Testing Workshop II* (University of York, 4-5 September 2003).

[134] PANE, J., MYERS, B., AND MILLER, L. Using HCI Techniques to Design a More Usable Programming System. In *IEEE Symposia on Human-Centric Languages and Environments* (Stresa, Italy, 2002), pp. 198–206.

[135] PANE, J. F. Human-Centered Design of a Programming system for Children. HCC01 Special Event—Children's Programming Odyssey, September 2001.

[136] PANE, J. F., RATANAMAHATANA, C. A., AND MYERS, B. A. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies 54* (2001), 237–264.

[137] PAPERT, S. *Mindstorms — Children, Computers, and Powerful Ideas*. Harvester Press, Brighton, 1980.

[138] PAYNTER, G. *Domain Independent Programming By Demonstration*. PhD thesis, Department of Computer Science, University of Waikato, New Zealand, 2000.

[139] PAYNTER, G. W., AND WITTEN, I. H. Automating iterative tasks with programming by demonstration: a user evaluation. Tech. Rep. 99/7, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1999.

[140] PAYNTER, G. W., AND WITTEN, I. H. Developing a practical programming by demonstration tool. In *Proceedings OZCHI* (Sydney, Australia, 2000), pp. 307–314.

[141] PETRE, M. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM 38*, 6 (June 1995), 33–44.

[142] PHILLIPS, G. Architectures for synchronous groupware. Tech. Rep. 1999-425, Department of Computing and Information Science, Queen's University, 1999.

[143] PIERNOT, P. P., AND YVON, M. P. *Watch what I do: programming by demonstration*, vol. 1. MIT Press, 1993, ch. 18: The AIDE Project: An Application-Independent Demonstrational Environment, pp. 382–401.

[144] POTTER, R. *Watch What I do: Programming by Demonstration*. MIT Press, 1993, ch. 17: Triggers: Guiding automation with Pixels to Achieve Data access.

[145] RADER, C., BRAND, C., AND LEWIS, C. Degrees of Comprehension: Children's Understanding of a Visual Programming Environment. In *Human Factors in Computing Systems: CHI '97 Conference Proceedings* (USA) (March 1997), pp. 351–358.

[146] RADER, C., CHERRY, G., BRAND, C., REPENNING, A., AND LEWIS, C. Designing Mixed Textual and Iconic Programming Languages for Novice Users. In *IEEE Symposium on Visual Languages* (Halifax, Nova Scotia, 1998), IEEE Computer Society.

[147] RAYMOND, E. S. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 2001.

[148] REISS, S. Pecan: Program development systems that support multiple views. In *[59]*. IEEE Computer Society Press Tutorial, 1990, pp. 324–333.

[149] REISS, S. Working in the garden environment for conceptual programming. In *[59]*. IEEE Computer Society Press Tutorial, 1990, pp. 334–345.

[150] REKIMOTO, J. A multiple device approach for supporting whiteboard-based interactions. In *Human Factors in Computing Systems: CHI '98 Conference Proceedings* (USA) (Los Angeles, CA, 1998), pp. 18–23.

[151] REPENNING, A. Agentsheets : an Interactive Simulation Environment with End-User Programmable Agents. In *Proceedings of the IFIP Conference on Human Computer Interaction* (INTERACT '2000, Tokyo, Japan) (2000).

[152] REPENNING, A., AND AMBACH, J. The agentsheets behavior exchange: Supporting social behavior processing. In *CHI 97 Electronic Publications* (1997). http://www.acm.org/sigchi/chi97/proceedings/demo/ar.htm.

[153] REPENNING, A., AND PERRONE, C. Programming by Analogous Examples. *Communications of the ACM 43*, 3 (March 2000), 90–97.

[154] RESNICK, M., BRUCKMAN, A., AND MARTIN, F. Constructional Design: Creating New Construction Kits for Kids. In *[40]*. Morgan Kaufmann Publishers, 1999, ch. 7.

[155] RETTIG, M. Prototyping for Tiny Fingers. *Communications of the ACM 37*, 4 (April 1994), 21–27.

[156] RICK, J. Understanding Children's Programming as Poor Learning Environments. HCC01 Special Event—Children's Programming Odyssey, September 2001.

[157] ROSE, C., HACKER, B., AND INC., A. C. *Inside Macintosh Volume VI*. Addison-Wesley, 1985.

[158] ROSE, D. Apprenticeship and exploration: A new approach to literacy instruction. In *CAST Literacy research papers (www.cast.org)*, no. 6 in 1. New York: Scholastic, 1995.

[159] ROSEMAN, M., AND GREENBERG, S. Groupkit: A groupware toolkit for building real-time conferencing applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4) (1992), pp. 43–50.

[160] SCHIFFER, S., AND FRÖHLICH, J. H. Concepts and Architecture of Vista - a Multi-paradigm Programming Environment. In *Proceedings of 1994 IEEE Symposium on Visual Languages* (St. Louis USA, Oct 1994), pp. 40–47.

[161] SCOTT, S. D., MANFRYK, R. L., AND INKPEN, K. M. Understanding children's interactions in synchronous shared environments. In *Proc. Computer Supported Cooperative Learning 2002 (CSCL '02)* (Boulder, Colorado, 2002), pp. 333–341.

[162] SHEEHAN, R. Turning ICE into Icicle. In *Proceedings of ED-MEDIA 2002: World Conference on Educational Multimedia, Hypermedia & Telecommunications* (Denver, Colorado, USA, June 24–29 2002), pp. 1796–1797.

[163] SHOEMAKER, G. B. D., AND INKPEN, K. M. Single display groupware: Augmenting public displays with private information. In *Human Factors in Computing Systems: CHI 2001 Conference Proceedings* (USA) (Minneapolis, Minnesota, 2001), pp. 522–529.

[164] SIMPLE DIRECTMEDIA LAYER (SDL) HOME PAGE. http://www.libsdl.org.

[165] SMITH, D. C. *Watch what I do: programming by demonstration*, vol. 1. MIT Press, 1993, ch. 1: Pygmalion: An Executable Electronic Blackboard, pp. 18–48.

[166] SMITH, D. C., AND CYPHER, A. Making Programming Easier for Children. In *[40]*. Morgan Kaufmann Publishers, 1999, ch. 9.

[167] SMITH, D. C., CYPHER, A., AND SPOHRER, J. KidSim: Programming Agents without a Programming Language. *Communications of the ACM 37*, 7 (July 1994), 54–67.

[168] SMITH, D. C., CYPHER, A., AND TESLER, L. Novice Programming Comes of Age. *Communications of the ACM 43*, 3 (March 2000), 75–81.

[169] SMITH, D. N. The interface construction kit. In *SIGGRAPH Symposium on User Interface Software* (October 17–19 1988), pp. 144–151. Also appears in [80].

[170] SMITH, R. B. What you see is what i think you see. *SIGCUE Outlook 21*, 3 (1992), 18–23.

[171] SOLOWAY, E. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM 29*, 7 (July 1986), 634–632.

[172] STAHL, G. Introduction: Foundations For A CSCL Community. In *Proc. Computer Supported Cooperative Learning 2002 (CSCL '02)* (2002), G. Stahl, Ed., pp. 1–2.

[173] STANTON, D., NEALE, H., AND BAYON, V. Interfaces to support children's co-present collaboration: Multiple mice and tangible technologies. In *Proc. Computer Supported Cooperative Learning 2002 (CSCL '02)* (Boulder, Colorado, 2002), pp. 342–351.

[174] STEFIK, M., BOBROW, D., FOSTER, G., LANNING, S., AND TATAR, D. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Transactions on Office Information Systems 5*, 2 (1987), 147–167.

[175] STEFIK, M., FOSTER, G., KAHN, K., BOBROW, D., LANNING, S., AND SUCHMAN, L. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. In *Computer Supported Cooperative Work: A Book of Readings*, I. Greif, Ed. Morgan Kaufmann, 1988, pp. 334–366.

[176] SUTHERLAND, I. E. Sketchpad: A man-machine graphical communication system. In *AFIPS Conference Proceedings, Sprint Joint Computer Conference* (1963), pp. 2–19. Also appears in [59].

[177] SUZUKI, H., AND KATO, H. Interaction-Level Support for Collaborative Learning: *AlgoBlock* — An Open Programming Language. In *Proc. Computer Supported Collaborative Learning '95* (Bloomington, Indiana, October 1995), J. L. Schnase, Ed., pp. 349–355.

[178] SVANÆS, D. Settling for less than the Holy Grail? CHI99 Position Paper for Workshop on End-User Programming and Blended-User Programming, 1999.

[179] SVENDSEN, G. B. The influence of interface style on problem solving. *International Journal of Man-Machine Studies 35*, 3 (September 1991), 379–297.

[180] TANI, M., HORITA, M., YAMAASHI, K., TANIKOSHI, K., AND M.FUTAKAWA. Courtyard: Integrating shared overview on a large screen and per-user detail on individual

screens. In *Human Factors in Computing Systems: CHI '94 Conference Proceedings* (Boston, MA, USA, Apr. 24–28) (1994), pp. 44–50.

[181] TANIMOTO, S. L., AND RUNYAN, M. PLAY: An iconic programming system for children. In *Visual Languages*, S. Chang, T. Ichikawa, and P. A. Ligomendies, Eds. New York: Plenum, 1986, pp. 191–205.

[182] TEITELBAUM, T., AND REPS, T. The Cornell Program Synthesizer: A Syntax Directed Programming Environment. *Communications of the ACM 24* (1981), 563–573.

[183] THIMBLEBY, H., COCKBURN, A., AND JONES, S. Hypercard: An object-oriented disappointment. In *Building interactive systems: architectures and tools*, P. Gray and R. Took, Eds. Springer-Verlag, 1992, pp. 35–55.

[184] TOLEMAN, M. A., AND WELSH, J. An empirical investigation of language-based editing paradigms. Tech. Rep. 95—45, School of Information Technology and Electrical Engineering, University of Queensland, 1995.

[185] TUDOREANU, M. E., WU, R., HAMILTON-TAYLOR, A., AND KRAEMER, E. Empirical evidence that algorithm animation promoted understand of distributed algorithms. In *IEEE Symposia on Human-Centric Languages and Environments* (Stresa, Italy, 2002), pp. 236–243.

[186] VYGOTSKY, L. *Mind in Society*. Harvard University Press, 1978.

[187] WEINBERG, G. M. *The Psychology of Computer Programming: Silver Anniversary Edition*. Dorset House Publishing, 1998.

[188] WELSH, J., BROOM, B., AND KIONG, D. A design rationale for a language-based editor. *Software Practice & Experiance. 21*, 9 (1991), 923–948.

[189] WITTEN, I., AND MO, D. *Watch what I do: Programming by Demonstration*. Morgan Kaufmann Publishers, 2001, ch. 8: TELS: Learning Text Editing Tasks by Examples.

[190] WOLBER, D. Pavlov: An Interface Builder for Designing Animated Interfaces. *ACM Transactions on Computer-Human Interaction* (December 1997).

[191] WOLBER, D., AND MYERS, B. *Your Wish is my Command*. Morgan Kaufmann Publishers, 2001, ch. Stimulus-Response PBD: Demonstrating 'When' as Well as 'What'.

[192] WRIGHT, T. Pattern programmer. Submitted for publication.

[193] WRIGHT, T., AND COCKBURN, A. Writing, Reading, Watching: A Task-Based Analysis and Review of Learners' Programming Environments. In *IWALT 2000* (Auckland, New Zealand, December 2000), pp. 167–170.

[194] WRIGHT, T., AND COCKBURN, A. Evaluating Computer-Supported Collaboration for Learning a Problem Solving Task. In *ICCE '02* (December 2002), pp. 266–267.

[195] WRIGHT, T., AND COCKBURN, A. Mulspren: a MUltiple Language Simulation PRogramming ENvironment. In *IEEE Symposia on Human-Centric Languages and Environments* (Arlington, Virginia, September 2002), pp. 101–103.

[196] WRIGHT, T., AND COCKBURN, A. Solo, Together, Apart: Evaluating Modes of CSCL for Learning a Problem Solving Task. In *Proc. Computer Supported Cooperative Learning 2002 (CSCL '02)* (January 2002), pp. 552–553.

[197] WRIGHT, T., AND COCKBURN, A. A Language and Task-based Taxonomy of Programming Environments. In *IEEE Symposia on Human-Centric Languages and Environments* (Auckland, New Zealand, 2003), pp. 192–194.

[198] WRIGHT, T., AND COCKBURN, A. An evaluation of the effects of different forms of computer supported collaboration on problem solving strategies. In *Proceedings of CHINZ'03* (Dunedin, New Zealand, 2003), pp. 99–104.

[199] WRIGHT, T., AND COCKBURN, A. Mulspren: a multiple language programming environment for children. In *Proceedings of CHINZ'03* (2003), pp. 21–26.

[200] WRIGHT, T., AND COCKBURN, A. Evaluation of two textual programming notations for children. In *The Australasian User Interface Conference (AUIC)* (2005).

[201] WYETH, P., AND PURCHASE, H. C. Programming Without a Computer: A New Interface For Children Under Eight. *Proc. Australian Computer Science Conference 22*, 5 (2000), 141–148.

[202] YELLAND, N. Collaboration and Learning with Logo: Does Gender Make a Difference? In *Proc. Computer Supported Collaborative Learning '95* (Bloomington, Indiana, 1995), pp. 397–401.

[203] YOURDON, E. *Structured walkthroughs: 4th edition*. Yourdon Press Computing Series, 1989.

[204] ZANDEN, B. V., AND MYERS, B. A. The lapidary graphical interface design tool. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (1991), ACM Press, pp. 465–466.

[205] ZAVE, P., AND JACKSON, M. Conjunction as composition. *ACM Trans. Softw. Eng. Methodol. 2*, 4 (1993), 379–411.

Appendix A

Simulations

This chapter contains all the simulations and questions used in the evaluation of multiple notations presented in Chapter 3. The simulation rules in both notations are presented first. The questions and screen layouts follow.

A.1 Rules

Code used in the first simulation

Conventional	English-like

`simulation.upKey() {`
`  if (not pacman.below(any Wall)) {`
`  pacman.move(UP);`
`  }`
`}`
	`whenever an up key is pressed:`
	`  if pacman is not below any Wall then:`
	`  move pacman up`
	`  end if`
	`end whenever`

continued on next page. . .

. . . continued from previous page

Conventional	English-like

`simulation.downKey() {`
`  if (not pacman.above(any Wall) ){`
`  pacman.move(DOWN);`
`  }`
`}`
	`whenever a down key is pressed:`
	`  if pacman is not above any Wall then:`
	`  move pacman down`
	`  end if`
	`end whenever`

`simulation.leftKey() {`
`  if (not pacman.rightOf(any Wall)) {`
`  pacman.move(LEFT);`
`  }`
`}`
	`whenever a left key is pressed:`
	`  if pacman is not right of any Wall then:`
	`  move pacman left`
	`  end if`
	`end whenever`

`simulation.rightKey() {`
`  if (not pacman.leftOf(any Wall)) {`
`  pacman.move(RIGHT);`
`  }`
`}`
	`whenever a right key is pressed:`
	`  if pacman is not left of any Wall then:`
	`  move pacman right`
	`  end if`
	`end whenever`

`any PacMan.contactWith(any PowerPill) {`
`  the PacMan.power = 10;`
`  the PowerPill.remove();`
`}`
	`whenever any PacMan touches any PowerPill:`
	`  set the PacMan’s power to 10`
	`  remove the PowerPill`
	`end whenever`

continued on next page. . .

. . . continued from previous page

Conventional	**English-like**

`any PacMan.contactWith(any Ghost) {`
`  if (thePacMan.power > 0) {`
`  the Ghost.remove();`
`  } else {`
`  the PacMan.remove();`
`  }`
`}`
	`whenever any PacMan touches any Ghost:`
	`  if thePacMan's power is greater than 0`
	`then:`
	`  remove the Ghost`
	`  otherwise:`
	`  remove the PacMan`
	`  end if`
	`end whenever`

`simulation.everySecond() {`
`  if (pacman1.power > 0 ) {`
`  pacman1.power = pacman1.power - 1;`
`  }`
`  every Ghost.move(RANDOM);`
`}`
	`every second:`
	`  if pacman1's power is greater than 0`
	`then:`
	`  subtract 1 from pacman1's power`
	`  end if`
	`  move every Ghost random direction`
	`end every`

Table A.1: Code used in the first simulation: the Pac-man simulation.

Code used in the second simulation

Conventional	English-like

`simulation.upKey() {`
`  if (player1.on(any Ladder)) {`
`  player1.move(UP);`
`  }`
`}`
	`whenever an up key is pressed:`
	`  if player1 is on any Ladder then:`
	`  move player1 up`
	`  end if`
	`end whenever`

`simulation.downKey() {`
`  if (player1.on(any Ladder)) {`
`  player1.move(DOWN);`
`  }`
`}`
	`whenever a down key is pressed:`
	`  if player1 is on any Ladder then:`
	`  move player1 down`
	`  end if`
	`end whenever`

`simulation.leftKey() {`
`  if (not player1.rightOf(any Wall)) {`
`  player1.move(LEFT);`
`  }`
`}`
	`whenever a left key is pressed:`
	`  if player1 is not right of any Wall then:`
	`  move player1 left`
	`  end if`
	`end whenever`

`simulation.rightKey() {`
`  if (not player1.leftOf(any Wall)) {`
`  player1.move(RIGHT);`
`  }`
`}`
	`whenever a right key is pressed:`
	`  if player1 is not left of any Wall then:`
	`  move player1 right`
	`  end if`
	`end whenever`

continued on next page. . .

. . . continued from previous page

Conventional	**English-like**

`player1.contactWith(any Fountain) {`
`  player1.power = player1.power + 5;`
`}`
	`whenever player1 touches any Fountain:`
	`  add 5 to player1's power`
	`end whenever`

`player1.contactWith(any Monster) {`
`  if (player1.power > 0) {`
`  the Monster.remove();`
`  } else {`
`  player1.remove();`
`  }`
`}`
	`whenever player1 touches any Monster:`
	`  if player1's power is greater than 0`
	`then:`
	`  remove the Monster`
	`  otherwise:`
	`  remove player1`
	`  end if`
	`end whenever`

`simulation.everySecond() {`
`  if (player1.power > 0 ) {`
`  player1.power = player1.power - 1;`
`  }`
`  every Monster.move(RANDOM_DIRECTION);`
`}`
	`every second:`
	`  if player1's power is greater than 0`
	`then:`
	`  subtract 1 from player1's power`
	`  end if`
	`  move every Monster random direction`
	`end every`

Table A.2: Code used in the second simulation: the mine simulation.

Code used in the third simulation

Conventional	English-like

`simulation.upKey() {`
`  if (not lostPerson.below(any Wall)) {`
`  lostPerson.move(UP);`
`  }`
`}`
	`whenever an up key is pressed:`
	`  if lostPerson is not below any Wall then:`
	`  move lostPerson up`
	`  end if`
	`end whenever`

`simulation.downKey() {`
`  if (not lostPerson.above(any Wall) ){`
`  lostPerson.move(DOWN);`
`  }`
`}`
	`whenever a down key is pressed:`
	`  if lostPerson is not above any Wall then:`
	`  move lostPerson down`
	`  end if`
	`end whenever`

`simulation.leftKey() {`
`  if (not lostPerson.rightOf(any Wall)) {`
`  lostPerson.move(LEFT);`
`  }`
`}`
	`whenever a left key is pressed:`
	`  if lostPerson is not right of any Wall then:`
	`  move lostPerson left`
	`  end if`
	`end whenever`

`simulation.rightKey() {`
`  if (not lostPerson.leftOf(any Wall)) {`
`  lostPerson.move(RIGHT);`
`  }`
`}`
	`whenever a right key is pressed:`
	`  if lostPerson is not left of any Wall then:`
	`  move lostPerson right`
	`  end if`
	`end whenever`

continued on next page. . .

. . . continued from previous page

Conventional	**English-like**

`lostPerson.contactWith(any Map) {`
` lostPerson.happiness = 10;`
` lostPerson.colour = GREEN;`
`}`
	`whenever lostPerson touches any Map:`
	`  set lostPerson's happiness to 10`
	`  set lostPerson's colour to GREEN`
	`end whenever`

`lostPerson.contactWith(any Flag) {`
`  if (lostPerson.happiness > 0) {`
`  the Flag.colour = RED;`
`  }`
`}`
	`whenever lostPerson touches any Flag:`
	`  if lostPerson's happiness is greater than 0 then:`
	`  set the Flag's colour to RED`
	`  end if`
	`end whenever`

`simulation.everySecond() {`
`  if (lostPerson.happiness > 0 ) {`
`  lostPerson.happiness = lostPerson.happiness - 1;`
`  }`
`  if (lostPerson.happiness == 0 ) {`
`  lostPerson.colour = BLUE;`
`  }`
`}`
	`every second:`
	`  if lostPerson's happiness is greater than 0 then:`
	`  subtract 1 from lostPerson's happiness`
	`  end if`
	`  if lostPerson's happiness is equal to 0 then:`
	`  set lostPerson's colour to BLUE`
	`  end if`
	`end every`

Table A.3: Code used in the third simulation: the maze simulation.

Code used in the fourth simulation

Conventional	English-like

simulation.leftKey() {
  if (ship.touching(leftOfScreen)) {
  ship.move(LEFTOF, rightOfScreen);
  } else {
  ship.move(LEFT);
  }
}
	whenever a left key is pressed:
	  if ship is touching leftOfScreen then:
	  move ship left of rightOfScreen
	  otherwise:
	  move ship left
	  end if
	end whenever

simulation.rightKey() {
  if (ship.touching(rightOfScreen)) {
  ship.move(RIGHTOF, leftOfScreen);
  } else {
  ship.move(RIGHT);
  }
}
	whenever a right key is pressed:
	  if ship is touching rightOfScreen then:
	  move ship right of leftOfScreen
	  otherwise:
	  move ship right
	  end if
	end whenever

simulation.spaceBar() {
  if (ship.numberBullets > 0) {
  new Bullet;
  the Bullet.move(ABOVE, ship1);
  ship.numberBullets =
ship.numberBullets - 1;
  }
}
	whenever the space bar is pressed:
	  if ship’s numberBullets is greater than
	0 then:
	  create a new Bullet
	  move the Bullet above ship1
	  subtract 1 from ship’s numberBullets
	  end if
	end whenever

any Bullet.contactWith(topOfScreen) {
  theBullet.remove();
  ship.numberBullets =
ship.numberBullets + 1;
}
	whenever any Bullet touches topOfScreen:
	  remove theBullet
	  add 1 to ship’s numberBullets
	end whenever

continued on next page. . .

. . . continued from previous page

Conventional	**English-like**

`any Bullet.contactWith(any Alien) {`
`  the Bullet.remove();`
`  the Alien.remove();`
`  ship.numberBullets =`
`ship.numberBullets + 1;`
`  ship.score = ship.score + 100;`
`}`
	`whenever any Bullet touches any Alien:`
	`  remove the Bullet`
	`  remove the Alien`
	`  add 1 to ship's numberBullets`
	`  add 100 to ship's score`
	`end whenever`

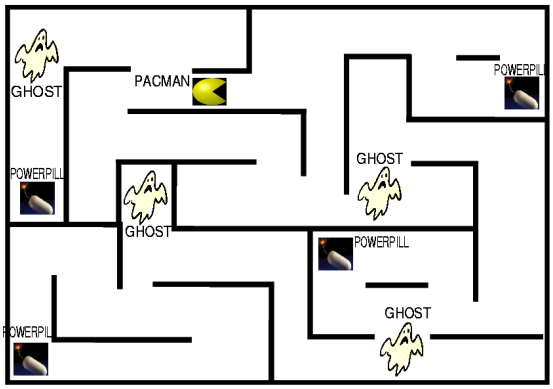
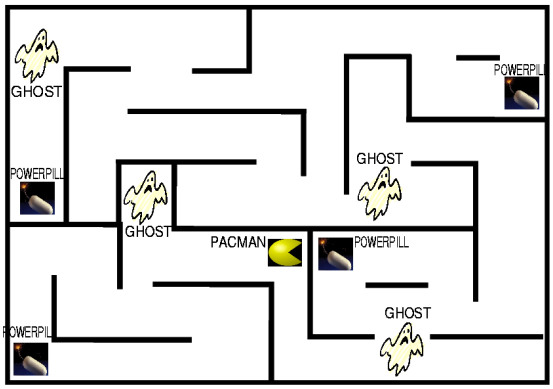
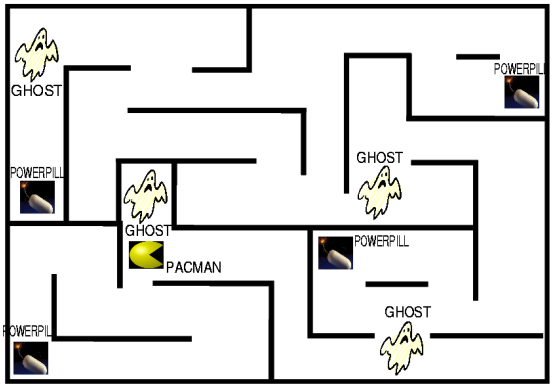
`ship.contactWith(any Alien) {`
`  ship.remove();`
`  simulation.restart();`
`}`
	`whenever ship touches any Alien:`
	`  remove ship`
	`  restart the simulation`
	`end whenever`

`simulation.everySecond() {`
`  any Bullet.move(UP);`
`  any Alien.move(RANDOM);`
`  ship.score = ship.score - 1;`
`}`
	`every second:`
	`  move any Bullet up`
	`  move any Alien random direction`
	`  subtract 1 from ship's score`
	`end every`

Table A.4: Code used in the fourth simulation: the space invaders simulation.

A.2 Questions and Screen Snapshots

Questions in the first simulation

Question	**Simulation Layout**
What will happen to Pacman when the left key is pressed? ↪ Pacman will move left (correct) ↪ Pacman will move right ↪ Pacman will move up ↪ Pacman will move down ↪ Pacman will eat a power pill	
What will happen to Pacman when the right key is pressed? ↪ Pacman will not move (correct) ↪ Pacman will move right ↪ Pacman will move left ↪ Pacman will move down ↪ Pacman will eat a power pill	
What will happen to Pacman when Pacman touches a Ghost? (Pacman's power is 0) ↪ pacman will be removed (correct) ↪ pacman will not move ↪ pacman will move down ↪ pacman will move left ↪ nothing will happen	

continued on next page…

. . . continued from previous page

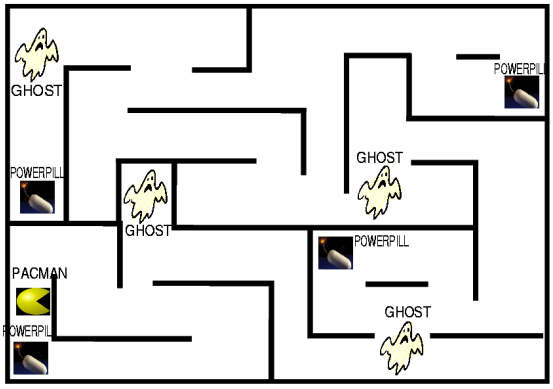
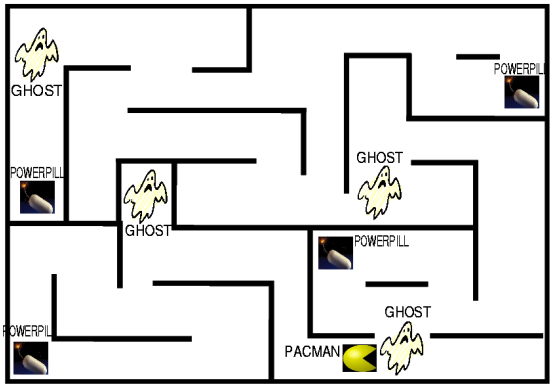
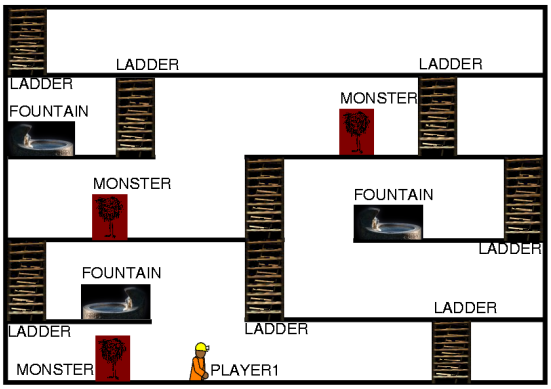
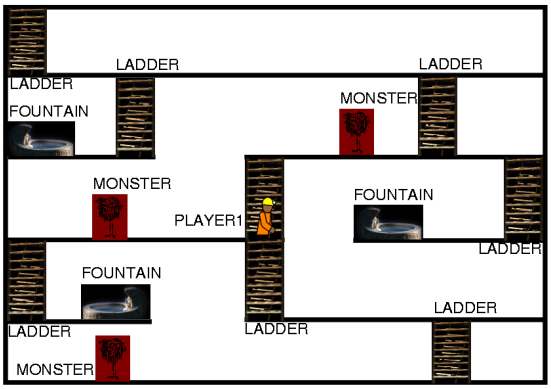
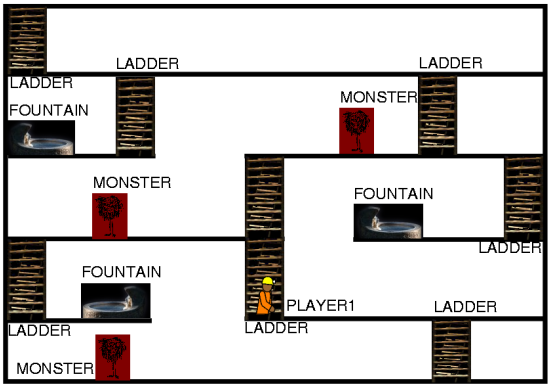
Question	**Simulation Layout**
What will happen to a Powerpill if Pacman touches a Powerpill? ↪ the powerpill will be removed (correct) ↪ nothing will happen ↪ the powerpill will move up ↪ the powerpill will change colour ↪ the powerpill will move down	
What will happen to a ghost if Pacman touches the ghost? (Pacman's power is 5) ↪ The ghost will be removed (correct) ↪ The ghost will move down ↪ The ghost will move up ↪ Nothing will happen ↪ The ghost will move left	

Table A.5: Questions asked in the first simulation: the Pacman simulation.

Questions in the second simulation

Question	Simulation Layout
What will happen to player1 when the up key is pressed? ↪ player1 will not move (correct) ↪ player1 will move up ↪ player1 will move left ↪ player1 will move down ↪ player1 will be removed	
What will happen to player1 when the down key is pressed? ↪ player1 will move down (correct) ↪ player1 will not move ↪ player1 will move left ↪ player1 will move up ↪ player1 will be removed	
What will happen to player1 when the right key is pressed? ↪ player1 will move right (correct) ↪ player1 will not move ↪ player1 will move left ↪ player1 will move down ↪ player1 will be removed	

continued on next page. . .

. . . continued from previous page

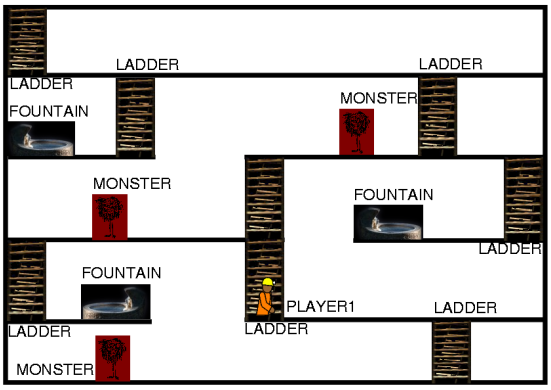
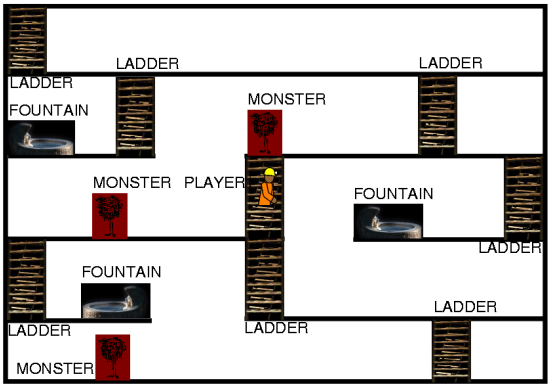
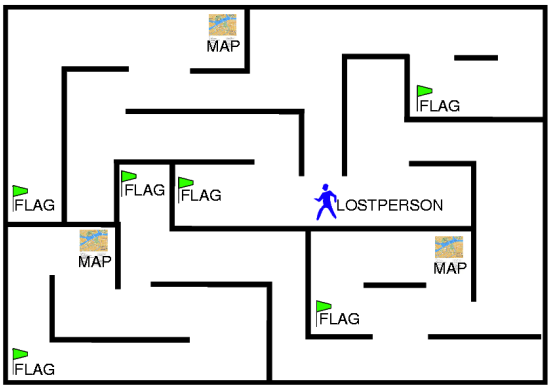
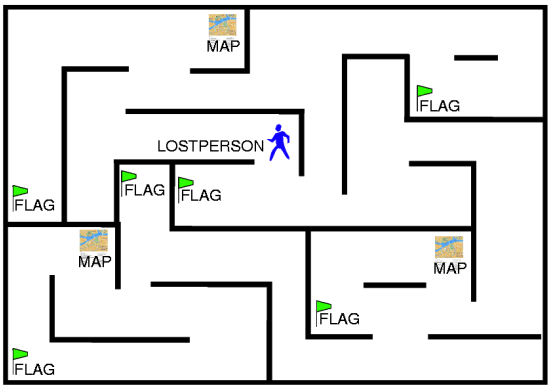
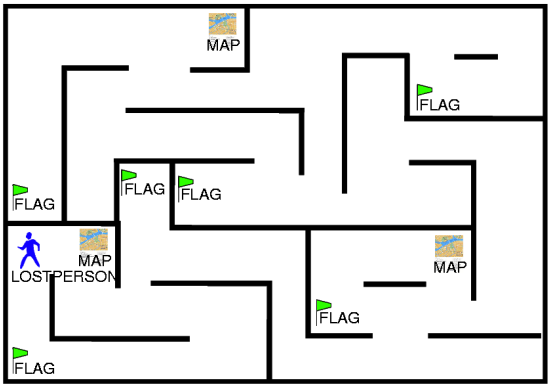
Question	**Simulation Layout**
What happens to Monsters every second? ↪ Monsters move in a random direction (correct) ↪ Monsters move up ↪ Monsters do not move ↪ The rules do not give me enough information to decide ↪ new Monsters appear	
What will happen to player1 when player touches a Monster? (player1's power is 0) ↪ player1 will be removed (correct) ↪ player1 will move down ↪ player1 will not move ↪ player1 will move up ↪ player1 will move left	

Table A.6: Questions asked in the second simulation: the mine simulation.

Questions in the third simulation

Question	Simulation Layout
What will happen to lostPerson when the left key is pressed? ↪ lostPerson will move left (correct) ↪ lostPerson will move right ↪ lostPerson will not move ↪ lostPerson will move up ↪ lostPerson will be removed	
What will happen to lostPerson when the up key is pressed? ↪ lostPerson will not move (correct) ↪ lostPerson will move right ↪ lostPerson will move left ↪ lostPerson will move up ↪ lostPerson will be removed	
What will happen to lostPerson when lostPerson touches a map? ↪ lostPerson will change from blue to green (correct) ↪ lostPerson will move up ↪ lostPerson will move down ↪ lostPerson will not move ↪ lostPerson will be removed	

continued on next page...

. . . continued from previous page

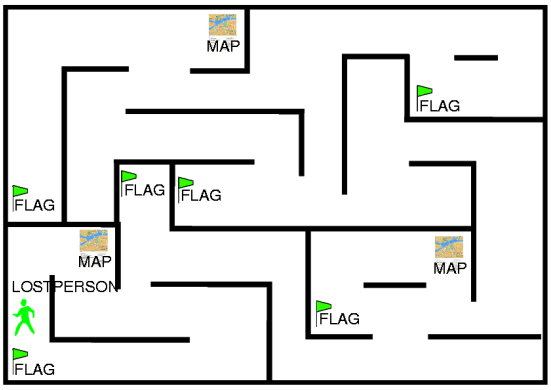
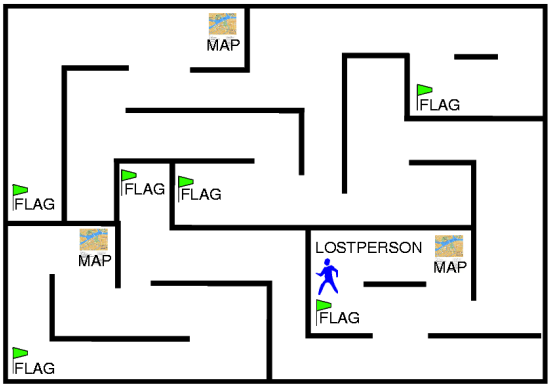
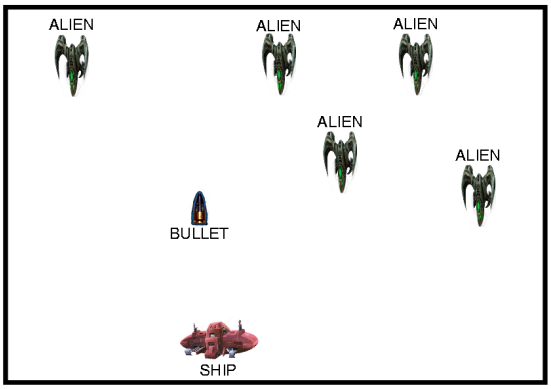
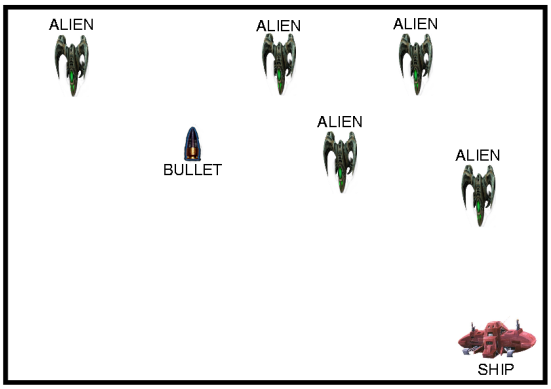
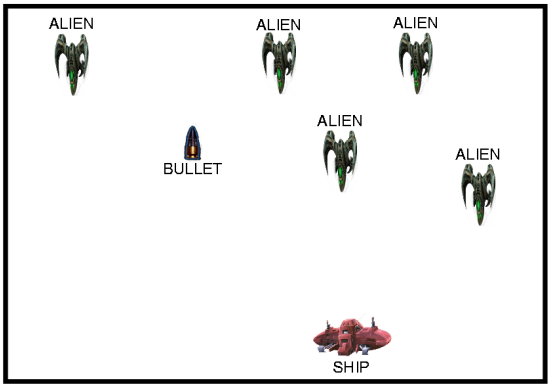
Question	**Simulation Layout**
What will happen to a flag when lostPerson touches a flag? (lostPerson's happiness is 3) ↪ the flag will change from green to red (correct) ↪ the flag will be removed ↪ the flag will move up ↪ the flag will move down ↪ nothing will happen	
What will happen to a flag when lostPerson touches it? (lostPerson's happiness is 0) ↪ nothing will happen (correct) ↪ the flag will change colour ↪ the flag will move down ↪ the flag will move up ↪ the flag will be removed	

Table A.7: Questions asked in the third simulation: the maze simulation.

Questions in the fourth simulation

Question	Simulation Layout
What will happen to the space ship when the left key is pressed? ↪ Space ship will move left (correct) ↪ Space ship will move right ↪ Space ship will not move ↪ Space ship will fire a bullet ↪ Space ship will be removed	
What will happen to the space ship when the right key is pressed? ↪ Space ship will move to left side of screen (correct) ↪ Space ship will not move ↪ Space ship will move right ↪ Space ship will fire a bullet ↪ Space ship will be removed	
What will happen to a bullet when a bullet touches an alien? ↪ Bullet will be removed (correct) ↪ Bullet will move up ↪ Bullet will not move ↪ Bullet will move down ↪ Bullet will move left	

continued on next page...

…continued from previous page

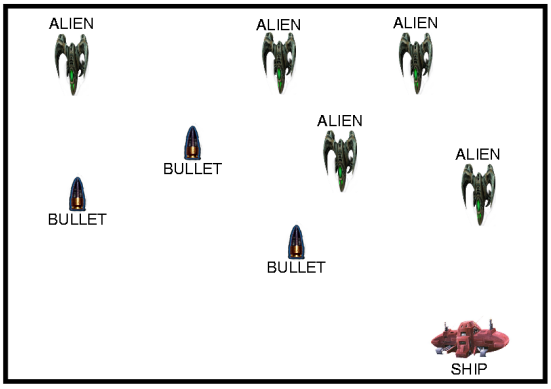
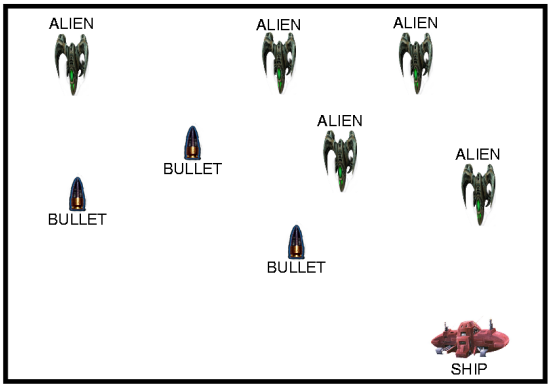
Question	**Simulation Layout**
What will happen to an Alien when a Bullet touches an Alien? ↪ Alien will be removed (correct) ↪ Alien will move left ↪ A new Alien will be created ↪ Alien will not move ↪ Alien will move down	
How many points does a player get for shooting an Alien? ↪ 100 (correct) ↪ 300 ↪ 50 ↪ 250 ↪ 3	

Table A.8: Questions asked in the fourth simulation: the space invaders simulation.