1

**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Sherpa: Guiding Students up the Smalltalk Mountain

by

Suzanne Carole Skublics

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science**

**School of Computer Science
Carleton University
Ottawa, Ontario
August 1, 1991**

**©copyright
1991, Suzanne Carole Skublics**

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-70958-8

Canada

**The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,**

**Sherpa: Guiding Students up the
Smalltalk Mountain**

submitted by

Suzanne Carole Skublics, B.A.Sc.

**in partial fulfillment of the requirements for
the degree of Master Of Computer Science**

Thesis Supervisor

Director, School of Computer Science

**Carleton University
August 1, 1991**

Abstract

The goal of this thesis is to make it easier for students with little or no programming experience to learn Smalltalk. We are motivated by the use of Smalltalk as the first programming language within the Bachelor of Computer Science program at Carleton University. From our experience, we have found that Smalltalk is a good vehicle for introducing programming and problem solving concepts. However, the object-oriented paradigm and the Smalltalk system consisting of the language, the class library, and the environment, require innovative pedagogic approaches and support tools. This thesis describes the design and implementation of Sherpa: a prototype development environment to make Smalltalk easier for students to learn, and easier for instructors to teach. Using Sherpa, multiple instructors can design several incremental views of the Smalltalk system to correspond with the presentation of course material. Our system is unique in this respect. Through the use of graphics and a spiral curriculum, students learn object-oriented concepts more gradually in Sherpa. We propose a curriculum which can be used in conjunction with Sherpa for an introduction to programming and problem solving course that uses Smalltalk.

Acknowledgements

I would like to thank Professor John Pugh for his guidance and constructive criticism throughout this thesis, and Professor Dave Thomas for his continuous encouragement and understanding.

This research was supported by Northern Telecom Canada. I am indebted to them for their assistance.

My good friends Janet Jansen and Paul White were particularly supportive during eight difficult months. Thanks for the tea.

I am grateful to my colleagues at Object Technology International (OTI), in particular Kent Johnson and Peter Shipton for sharing their insights into the mysteries of Smalltalk, and Kevin McGuire for his user interface expertise.

Thanks to my mother and my brother Tom for their love and patience.

Finally, and most especially, my thanks to James for his love and support. The experience I gained from watching him complete his thesis cannot be described.

Table of Contents

1	Introduction.....	2
1.1	Motivation.....	2
1.2	Sherpa.....	4
1.3	Thesis Organization.....	6
2	Experiences of Smalltalk Users.....	7
2.1	Our Experience	7
2.2	Other Experiences	11
2.2.1	Borning and O'Shea: Deltatalk	11
2.2.2	Nielsen and Richards: Learning Smalltalk	12
2.2.3	Urlocker: Teaching Object-Oriented Programming	14
2.2.4	O'Shea Panel	16
2.2.5	Rosson and Carroll	19
2.3	Summary.....	21
3	Related Work.....	22
3.1	Program Visualization.....	22
3.1.1	Debugging with Visualization	23
	PROVIDE.....	24
3.1.2	Teaching with Visualization	25
3.1.3	Object-Oriented Visualization Tools	26
	TRACK.....	27
	GraphTrace	29
3.2	Animation	31
3.2.1	Cunningham Diagrams.....	31
3.2.2	Smalltalk Animation	33
3.3	Teaching Systems.....	34
3.3.1	Children	35
	Karel the Robot.....	35
	Playground.....	37
	Discussion.....	39
3.3.2	Adults	40
	IBM T.J. Watson Research Group.....	40
	Discussion.....	47
3.4	Summary.....	48

4	Design	49
4.1	The Student Sherpa Environment	51
4.1.1	The Worlds	51
4.1.2	World Views	54
4.1.3	The GraphicWorld	55
4.1.4	The WorldBrowser	59
4.1.5	The Galaxies	66
4.1.6	The AnimatedTracer	68
4.1.7	The Viewer	81
4.1.8	The ErrorWindow	83
4.2	The Instructor Sherpa Environment	85
4.2.1	The Universe	86
4.2.2	The Universe Window	86
4.2.3	The WorldModifier	90
4.3	Summary	93
5	Teaching Curriculum	94
5.1	The Stages	96
5.2	Summary	100
6	Conclusion	101
6.1	Conclusion	101
6.2	The Sherpa Implementation	103
6.3	Future Research Topics	106
7	References	109

List of Figures

Figure 1: TRACK.....	28
Figure 2: GraphTrace	30
Figure 3: Cunningham Diagrams	32
Figure 4: Karel the Robot's World	36
Figure 5: Playground.....	38
Figure 6: The ViewMatcher.....	42
Figure 7: The overall Sherpa environment versus the base Smalltalk environment	50
Figure 8: PenWorld in Sherpa versus Pens without Sherpa	53
Figure 9: Class Hierarchy Browser in the Smalltalk System	54
Figure 10: A GraphicWorld Window	56
Figure 11: A GraphicWorld open on a RobotWorld.....	57
Figure 12: Sample default pictures of objects not in the picture library	58
Figure 13: The BitEditor open on a Robot picture	59
Figure 14: A CHB with class Pen selected.....	60
Figure 15: A WorldBrowser with class Pen selected.....	61
Figure 16: The CHB method pane displaying a class definition and method code.	62
Figure 17: Modal Class Definition Window.....	63
Figure 18: Class definitions in Sherpa and in the standard CHB	64
Figure 19: A method with source code visible	65
Figure 20: A method with source code hidden	65
Figure 21: Literal Galaxy Window	67
Figure 22: The AnimatedTracer.....	69
Figure 23: Trace of greet: message	70
Figure 24: "traceIt" chosen from any text pane menu.....	70
Figure 25a: Animation of "Quincy greet: Tommy" in the AnimatedTracer	71
Figure 25b: Animation of "Quincy greet: Tommy" in the AnimatedTracer	72
Figure 25c: Animation of "Quincy greet: Tommy" in the AnimatedTracer	73
Figure 25d: Animation of "Quincy greet: Tommy" in the AnimatedTracer	74
Figure 25e: Animation of "Quincy greet: Tommy" in the AnimatedTracer	75
Figure 26: Objects returned from a message send	76
Figure 27: Viewers before greet:.....	79
Figure 28: Viewers after greet:	80
Figure 29: An Inspector and a Viewer open on the same Robot object.....	82

Figure 30: Multiple Viewers	82
Figure 31: A Walkback window	83
Figure 32: The Debugger.....	84
Figure 33: The ErrorWindow	85
Figure 34: A Universe window	87
Figure 35: A Universe window with its help window	88
Figure 36: A Universe with several Galaxies and Worlds	88
Figure 37: Toys Galaxy selected.....	89
Figure 38: A WorldModifier.....	91
Figure 39: RobotWorld containing the basic Robot behaviour	92
Figure 40: Spiral curriculum using Sherpa.....	95
Figure 41: OrderedCollection World	99
Figure 42: UI World.....	100

List of Appendices

Appendix I: Sherpa Classes	113
Appendix II: Base Classes	114

"Sherpa (sher'pa) n. one of Nepal tribe, employed as porter or guide on Himalayan mountaineering expeditions." Webster

1 Introduction

1.1 Motivation

The motivation for this thesis arises from our experience teaching Smalltalk¹ as the introductory programming language within the Bachelor of Computer Science program at Carleton University. Our objectives are to make Smalltalk easier for students to learn, and easier for instructors to teach. We have designed and implemented an enhanced development environment for students, and flexible tools for multiple instructors to tailor this environment. Our work is aimed toward students with little or no previous programming experience.

An object-oriented programming language, such as Smalltalk, is good for teaching the basic concepts of computer science. The built-in class library permits instructors to focus more on real-world problem solving rather than on abstract computer science problems [Pugh 90]. The Computer Science department at Carleton University recently switched from Pascal to Smalltalk in the introductory programming course. Other languages, such as Scheme, Modula-2, and Ada, were also considered. Smalltalk was chosen because it:

1. supports data and procedural abstraction
2. has a modern interactive programming environment
3. permits self-motivated students to explore
4. has a simple and minimal syntax
5. runs on MS-DOS based microcomputers
6. has some simple user interface components such as menus and prompts
7. has industrial relevance
8. can be used as an implementation language in subsequent computer science courses [LaLonde and Pugh 90b].

¹ Specifically, Smalltalk/V286, which is a registered trademark of Digital, Inc. In this paper, Smalltalk means Smalltalk/V286.

Our experiences with teaching the language concur with those of other researchers: there is a cost to learning Smalltalk's rich programming environment. Students progress slowly. David N. Smith coined the phrase, "climbing the Smalltalk mountain"² to describe students learning Smalltalk. In our introductory programming course, we have observed that the first *hill* for students climbing the mountain is learning the basic object-oriented concepts such as objects and message sending. Those with no previous programming experience climb this *hill* easier than those with experience. Most students seem to best understand the introductory concepts when they are explained using diagrams and real world examples. When students are learning how to use the tools and how to navigate the environment, previous experience is not a significant advantage. By the end of the course, the students have learned how to work effectively in the Smalltalk environment; we say they have reached *first base camp*.

In a classroom situation, the instructor of the course controls what is being taught and when it is taught. Both the object-oriented paradigm and the Smalltalk environment are taught incrementally: new concepts build on previous ones. However, when the students start to use Smalltalk on their own in the lab, they are exposed to a rich development environment that tends to overwhelm them. The size of the class library is the most common obstacle - there are over 100 classes and 2000 methods available.

Whereas extensive work³ has been done to simplify the teaching of traditional programming languages such as Pascal, little research has been done to simplify the learning and teaching of an object-oriented programming language. Program visualization tools such as GraphTrace [Kleyn and Gingrich 88], TRACK [Boecker and Herczeg 90], BALSA [Brown and Sedgewick 84] and Cunningham Diagrams [Cunningham and Beck 86] help students learn program analysis and debugging but they

² [Rosson and Carroll 90], page 76.

³ See past issues of ACM's Special Interest Group on Computer Science Education (SIGSCE).

are not aimed at beginners. Program analysis and debugging are only a small part of learning how to program. We need a development environment that corresponds to a learning curriculum and contains program visualization tools. Systems such as Logo [Papert 80] and Playground [Fenton and Beck 89] are successful learning environments but were designed for children, not for university level students.

Researchers at the IBM T.J. Watson Research Center [Rosson and Carroll 90] have recently developed a system that they claim simplifies the learning of Smalltalk by restricting the information to which a learner is exposed. This system relies on the previous programming experience of students and is self taught. From our experience, we believe that students with no previous programming background have a better understanding of object-oriented concepts and Smalltalk when taught by an instructor. Concepts are then reinforced when students perform meaningful tasks in the development environment.

Through questionnaires and informal discussions, we gathered data from students to identify features of the programming environment that either hindered or simplified learning. Coupled with the successes and problems reported by other researchers learning and teaching Smalltalk, the information gathered was used to design a new working environment for students. We call our system Sherpa.

1.2 Sherpa

The Sherpa system is a Smalltalk development environment for students learning how to program, and for an instructor teaching students in an introductory programming course. We have enhanced the Smalltalk/V286 environment to make it easier for students to learn the language. The major contributions of Sherpa are:

1. to teach the basic concepts of problem solving with objects, we developed a graphic world in which students learn with pictures and animation
2. to prevent students from being confused and overwhelmed by the large class library, we developed a modified Browser which can provide a filtered view of system components
3. to support the teaching of encapsulation, we developed a read-only Inspector
4. to notify students of errors, we developed a simplified Debugger containing only necessary and sufficient information for debugging
5. to help students deal with the distributed nature of an object-oriented program, we developed an animated tracer for animating objects and message sends
6. to help instructors tailor the Sherpa system, we developed flexible tools for building, organizing, and simulating student environments

Instead of being exposed to the large and complex Smalltalk system, students learn the system incrementally, just as they learn computer science and object-oriented concepts in the course lectures. Students start learning the basic concepts graphically in Sherpa's graphic world. The Smalltalk class library is then slowly introduced using a modified Browser. An animated tracer is used in the early stages when students need to *see* what happens during a computation.

An instructor designs a set of student environments that incrementally expand the students' view of the system. The environments correspond to the instructor's course presentation material. If there are several instructors, the Sherpa system is flexible in that instructors can tailor the environment to fit their own curriculum. Instructors are provided with a full set of tools to control which classes, methods and variables students can access, which Debugger is available, and which Inspector is available.

1.3 Thesis Organization

In Chapter 2, we discuss our experiences using Smalltalk to teach an introductory programming course as well as the experiences of other teachers and users of Smalltalk and other object-oriented systems. In Chapter 3, we describe some of the existing tools and teaching systems that have influenced the design of Sherpa. In Chapter 4, we describe Sherpa's design and implementation and in Chapter 5 we propose a curriculum for teaching Smalltalk using Sherpa. Finally, we summarize our work and future plans in Chapter 6. We have included two Appendices:

1. the set of classes in the implementation of Sherpa
2. the set of base classes that students might learn in an introductory programming course using Smalltalk/V286.

2 Experiences of Smalltalk Users

A concern when learning Smalltalk is the time and effort required to become productive in the language and proficient in the environment. Most Smalltalk users are self-taught: they read reference manuals, used on-line tutorials, and drew on the experience of colleagues. Some have been taught in a lecture-style course. Unfortunately, there are few satisfactory textbooks and courses to help them: "...a dearth of teaching tools and methodologies are all that keep object-oriented programming from being effectively disseminated."⁴

A few researchers in the object-oriented world are looking into improving the learning environment. They have reported positive and negative aspects of Smalltalk, proposed teaching and learning tools, and experimented with learning strategies. Our Sherpa system design is based on our own experiences teaching Smalltalk and on the improvements suggested by other researchers.

2.1 Our Experience

At Carleton University, we have taught the introductory programming course using Smalltalk/V286 for two years. Smalltalk is also used in the applications course that students take subsequent to the introductory course. The data types course in second year had already used Smalltalk-80 for two years [LaLonde and Pugh 90b]. In these courses, students are exposed to concepts such as data abstraction, information hiding, inheritance, polymorphism and class libraries. Students do not learn case statements, linked lists or pointers that they would learn using a language such as C and Pascal.

⁴ Kent Beck in [O'Shea, Beck, Halbert, Schmucker 86], page 303.

Other courses in the curriculum were reorganized to teach students the traditional notions found in other programming paradigms. Using Smalltalk as the pedagogic language has been successful: the students enjoy using it, and they are learning a leading-edge technology. We believe that they are able to implement more realistic applications in their assignments, and that their skill and productivity levels have improved.

Despite the success, we wanted to improve the introductory programming course. Data was gathered from many sources, namely: discussing the course amongst the instructors, observing students' programming habits while in a lab, recording and analyzing the questions asked in class, and tracking common errors made on assignments and tests. We identified the following positive features of using the Smalltalk environment:

1. Interactive tools and incremental program development

Browsers, Source Code Debuggers and Inspectors help students work productively and efficiently because of incremental compilation and execution.

2. Consistency

A consistent programming environment reduces the overhead for beginners. In Smalltalk, everything is an object⁵ such as a string or a file, and all computation is done with message passing.

3. Syntax

Smalltalk's uniform syntax helps a beginner because it is simple and small. It also permits instructors to concentrate on explaining the semantic issues rather than dwelling on syntactic ones.

We identified the following common difficulties that students experience:

1. Class versus Instance

Students have trouble understanding the differences between classes and instances, and between class and instance methods.

⁵ Variables are not objects in Smalltalk.

2. Class library

Students are exposed to a class library with hundreds of classes and methods. They need only learn a small subset of classes and methods but they mistakenly believe they must learn the entire library.

3. Locating Objects

Due to the large class library and the distributed nature of object-oriented code, students have difficulty locating the classes and methods they need for code use and reuse.

4. Message sending

Students have difficulty understanding the behaviour of a class when the distributed nature of object-oriented code forces them to follow a chain of message sends.

5. Object instantiation and initialization

Students have trouble understanding how to create new objects and initialize them using the `new` message. The problem stems from the concept of metaclasses which we do not teach in an introductory course.

6. Encapsulation

Some students do not understand that they cannot access an object's internal state without a message send. The exception is the `Inspector` which violates encapsulation.

7. Debugging

The Debugger is a complicated tool for beginning students. They do not know how to sort through the vast amount of information contained in the debugger to help them solve a problem.

8. Help

There is a lack of sufficient textbooks and on-line help when students encounter problems.

Most of the comments are about the Smalltalk environment as opposed to the object-oriented paradigm. Students with little or no previous programming background experience difficulties learning the environment similar to those students simply learning their first programming language. Students who have previously programmed in

languages such as Pascal, C, or Basic experience more difficulty learning to use the Smalltalk environment because they must, in effect, unlearn their old ways.

In addition to the difficulties listed above, many students have problems building a user interface for their program. Smalltalk uses a paradigm for constructing user interfaces called Model-View-Controller⁶. The paradigm involves a subsystem of collaborating objects that support a highly interactive software development process. The responsibilities are separated between the model of the application domain, the views for displaying the state of the model, and the control of the model and views. This distribution of responsibilities between objects makes it difficult for students to examine and manipulate a coherent view. Even experienced MVC programmers spend a substantial amount of time creating a user interface [Shan 90].

We believe that the MVC paradigm is too complicated to be taught at the introductory level. Students can use simple user interface components such as prompters and menus without having to learn MVC. With the advent of better event-driven user interfaces⁷, user interface management systems⁸, and user interface builders, students will be able to build WIMP-based⁹ applications more easily.

⁶ The user interface paradigm in Smalltalk-80 is often referred to as Model-View-Controller or simply MVC. The term used in Smalltalk/V systems is Model-Pane-Dispatcher or MPD. MVC and MPD are similar but have subtle differences.

⁷ Eva [J. McAffer and D. Thomas. Eva: An Event Driver Framework for Building User Interfaces in Smalltalk. Proceedings Graphics Interface, 1988. pp 168-175], SmallScript [K. Healand and D. Thomas. SmallScript: A User Programmable Framework Based on Smalltalk and PostScript. Proceedings Graphics Interface, 1989. pp 55-61], Smalltalk/VPM and Windows all simplified the user interface paradigm with an event driven user interface.

⁸ The reader is directed to [Baecker and Buxton 87] for an extensive reference list of user interface management systems.

⁹ WIMP: Window, Icon, Menu, Pointer

2.2 Other Experiences

2.2.1 Borning and O'Shea: Deltatalk

Borning and O'Shea [Borning and O'Shea 87] suggest changes to the Smalltalk-80 language and environment that may make it easier to learn and use. The results and recommendations of an empirical study performed with students on the ease of learning Smalltalk-80 follows:

1. Students have difficulty with the class library because of the large number of classes and methods. They cannot find what they want and cannot distinguish between similar classes. Although Smalltalk-80 has categories to help organize classes, the navigation problem is intensified because Smalltalk-80 is much larger than Smalltalk/V286. The Browser is a helpful tool but it could be restructured to allow a more gradual introduction to the library.
2. Students need something to help them understand inheritance. Inherited state and methods are not immediately visible through the class browser. An improvement would be that while viewing a class in the Browser, its methods and its inherited methods would be visible in one window pane.
3. More support is needed for reorganizing the class hierarchy. For example, when a designer discovers that an abstract class should be introduced into an application, the methods must be factored among classes in the hierarchy. There is little support for simplifying this task.
4. The Model-View-Controller paradigm is an important framework for constructing user interfaces but it is more complex than is necessary.

The problems learning Smalltalk cited by Borning and O'Shea result in part from the class hierarchy library being large and distributed. The Class Hierarchy Browser is only one view of the library code; it can be changed to help students learn and navigate the system.

2.2.2 Nielsen and Richards: Learning Smalltalk

Nielsen and Richards discuss their experiences with learning and using Smalltalk-80, and offer suggestions to improve the environment based on this experience [Nielsen and Richards 89]. With the intention of building small-scale programs for Human-Computer-Interaction experiments, they learned Smalltalk by reading the standard Smalltalk books¹⁰, and exploring the system.

Nielsen and Richards encountered many problems. According to Nielsen and Richards:

1. The system should provide more help:

Without guidance, new students will encounter several problems.

2. Problems are due to the large and diffuse nature of Smalltalk and to the generic design of the classes:

Students typically follow a long chain of message-passing and class-inheritance links to understand how a program works. The Smalltalk Browser has cross-referencing tools to help students, but novices usually need to know the context of a program execution to understand it. Without actually executing the program, users do not know which method will be invoked at a given moment because of dynamic binding and polymorphism inherent in Smalltalk. Static code is the least effective way to understand the dynamics of a computation. "...novices should use a run-time source Debugger so they can see what is executing and inspect the values of instance variables."¹¹

¹⁰ A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.
A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

¹¹ [Nielsen and Richards 89], page 76.

3. The system needs more examples:

Users might learn more about a class if they could see example instances of that class; seeing an example use of a class is more beneficial than reading a general description. More class examples should be added to the system - a practice already prevalent in Smalltalk-80.

4. The uniform syntax does not help a novice

The uniform syntax makes all the code look the same. The simplicity of having everything in the system as an object and all computation being done through message passing is conceptually powerful, but interpreting the code is a demanding task. This difficulty is compounded by the fact that variables in Smalltalk are not typed. However, this does make Smalltalk more flexible for interactive, exploratory programming. Objects are typed and although not required, a variable is often accompanied by a comment explaining the class of object to which it is bound.

5. The system needs more rules:

It is difficult for novices to get a good understanding of the structure of the system because there are only a few general rules about how to construct objects. On the other hand, Smalltalk has many predefined data structures and it is not always clear which one to use. There are also many different ways of performing the same function. Although the large number of possibilities is useful to some users, it is confusing to others.

6. Users need help with code reuse and locating objects:

One of the many advantages of Smalltalk is the code reuse promoted by the class hierarchy. However, it is difficult to learn how to reuse. A user must find what reusable components are available and then learn how to reuse them. These steps are difficult because there are many classes to choose from and their protocol is distributed. On the positive side, the interactive and flexible nature of the environment lets the user modify code fragments and test them quickly to find the

desired behaviour. They suggest the addition of a smart Browser that would help the user find pieces of code according to a specified description; this is a good idea as psychologists have demonstrated that people are better at recognition than generation.

Nielsen and Richards, as well as many others, believe that the best way to learn Smalltalk is from knowledgeable colleagues. If this is not possible, a good tutorial is better than exploring and learning on one's own. Smalltalk is too large for the novice to grasp quickly. Once learned, Smalltalk is an enormously productive tool: A programming environment is a great improvement over traditional isolated languages [Nielsen and Richards 89].

Nielson and Richards had difficulties typical of students who teach themselves Smalltalk. Most of the problems they cited are solved when the basic object-oriented concepts are taught to students by an experienced Smalltalk instructor.

2.2.3 Urlocker: Teaching Object-Oriented Programming

Urlocker has taught object-oriented programming with Actor¹² to many professionals [Urlocker 89]. His students usually had procedural programming experience, but little prior exposure to object-oriented concepts. Urlocker identified difficulties his students had in learning object-oriented programming and based on his experience, he devised a model for teaching it. According to the model, students go through three phases when learning an object-oriented language. The phases are *unfreezing*, *change*, and *refreezing*.

In the *unfreezing* phase, students must let go of previous beliefs. This is the biggest difficulty for students making the transition from procedural to object-oriented

¹² A C-based object-oriented language with a Smalltalk syntax. [Whitewater Group. *Actor User's Manual*, Volume 1. 1990]

programming. They must think and design in an object-oriented way. This phase introduces them to the basic concepts of objects, message sending, classes and inheritance.

In the *change* phase, students learn the new approach. They learn how to use the development tools in the environment, how to create objects, how to send messages, how to browse code and how to create new classes and methods. Because they are using Actor, they then learn the syntax. The predefined classes such as collections, files and streams are subsequently introduced through lectures and exercises.

In the *refreezing* phase, students integrate the new knowledge with their programming experience and then practice applying the new concepts. The students undertake a realistic application to consolidate object-oriented techniques into their programming approach.

Following this model, Urlocker teaches courses which balance theory and practice. Hands-on sessions are needed to ensure that lessons are absorbed. As the students progress, the challenges increase and more realistic exercises are performed. Students must undertake a realistic application during their learning phase so that they will face many of the difficult design decisions of object-oriented programming. Windowing and graphics are slowly and sparingly introduced to maintain a high level of fun and interest among students.

We agree with Urlocker's descriptions of students learning an object-oriented language. Students go through phases, though maybe not as clearly defined as Urlocker contends. We do not believe that students must completely let go of their previous programming beliefs. They may still be able to think in terms of objects if the object-oriented concepts are related to their previous knowledge. The idea of integrating new knowledge with old, used in the refreezing phase, could be incorporated into the unfreezing stage: the old knowledge could be used to explain the new.

Urlocker's teaching model is appropriate for experienced users and can be simplified for students with no previous programming background. The curriculum we propose with Sherpa follows similar phases; the first phase teaches the introductory object-oriented concepts (objects, message sending, bindings), the second phase has the same content as Urlocker's (create classes and methods, browse code, class library), and the third phase solidifies the concepts by using them in the development of a realistic application.

2.2.4 O'Shea Panel

There is a lack of consensus on several key issues regarding the ease of learning object-oriented programming systems. This is evident in the panel discussion between Tim O'Shea, Kent Beck, Dan Halbert and Kurt Schmucker. [O'Shea, Beck, Halbert, Schmucker 86]

Tim O'Shea gathered data from professional programmers, teachers, postgraduate students, self-paced learners and self-taught programmers about learning object-oriented programming. O'Shea reports that the fundamental concepts - objects, classes and message sending - are easy for learners to master. The subtleties of method inheritance are difficult. Facilities for tracing method inheritance would improve the environment. The size of a mature object-oriented programming system is *horrifying*, but effective use of Browsers and restructuring the class library into layers eases navigation problems. With improved curricula and a consistent environment, O'Shea believes that object-oriented programming would be easier to learn.

Kent Beck has taught Smalltalk-80 to Tektronix customers and new engineers. Beck claims that there are five stages to mastering an object-oriented programming system:

1. Understanding the computational mechanism
2. Becoming proficient in the environment
3. Learning the available reusable parts of systems
4. Learning how to decompose problems into objects
5. Turning specific solutions into generic ones

The computational mechanism and the environment can be learned by reading the Goldberg Smalltalk-80 books and by using the environment. In the Smalltalk-80 system, learning the reusable parts is a daunting task because there are over 250 classes in the library. Beck claims that students feel comfortable browsing the system to learn what is available to help them. Learning how to decompose a problem into objects is difficult for procedural programmers because they cannot rely on their previous experience. However, after seeing examples in the system such as the MVC code, Beck claims that students learn how to solve problems using objects. The most difficult task, even for experts, is abstraction: making specific solutions into generic ones.

Beck concludes that teaching object-oriented programming is the same as teaching procedural systems: what is easy to teach in object-oriented programming is easy in procedural.

Dan Halbert's group does not teach formal courses but is introducing object-oriented programming to programmers by giving advice. The programmers have difficulty getting started, but they eventually enjoy it. Most of the programmers have a basic understanding of the ideas of object-oriented programming but they do not know the methodology.

Presently, people can learn object-oriented programming only by example, trial and error, or apprenticeship. Halbert states that although material exists describing various object-oriented systems, there is little material on the design of good object-oriented programs and a lack of examples of well written programs.

Halbert believes that it takes a relatively long time to learn to program in most object-oriented programming systems. The syntax and semantics are simple and do not slow down the user. Most of the time is spent learning the large class library. Libraries are a great advantage because programmers can build on existing code, but it is difficult to filter out what is essential and what is extra. This is a particular problem in Smalltalk/V where no distinction is made between private and public methods. Halbert suggests that the library be structured so that students can learn minimal subsets of classes and behaviour first and then gradually learn more.

Halbert's last point is that reading an object-oriented program on paper is difficult because of the distributed nature of the code: following a thread of control is not easy. This can be solved with environment tools to help read and write code, such as a tool that presents a complete view of a type with all its inherited methods, or a tool to look at code using the environment instead of paper. The tools that Halbert speaks of exist in some versions of Smalltalk.

Kurt Schmucker has taught a course in basic principles of object-oriented programming and a course specifically in Objective-C. He draws the following conclusions:

1. It is better to learn the basic concepts of object-oriented programming independent of a language.
2. Some people just do not understand but these people usually have problems with other high level languages as well.
3. The obstacle in a hybrid language, such as Objective-C, is competence in the underlying language.¹³
4. Interactive interpreters are invaluable for teaching.
5. Metaclasses and class messages are a slight complication for some but should not be removed from the system.

¹³ This problem has also been identified when C programmers move to C++. They tend to program in enhanced C rather than in an object-oriented manner. [W. LaLonde, J. McGugan, D. Thomas. "The Real Advantages of Pure Object-Oriented Systems or Why Object-Oriented Extensions to C are Doomed to Fail". IEEE, 1989. pp344-350]

Halbert's conclusions concur with our experiences using and teaching Smalltalk. Schmucker's discussion of an interactive interpreter and O'Shea's discussion of improving the curricula and making the Smalltalk environment more consistent are ideas we have adopted in Sherpa. Beck's idea that Smalltalk can be learned simply by reading the appropriate books, by using the system and by looking at examples in the system may be true for experienced programmers, but we believe it is inappropriate for beginners.

2.2.5 Rosson and Carroll

Rosson and Carroll have experimented with using Smalltalk and teaching it to experienced programmers [Rosson and Carroll 90]. They argue that object-oriented programming is a natural way of thinking but that this does not necessarily imply that it is easy to use or to learn. The little psychological evidence that exists is encouraging but there are costs to learning object-oriented programming [O'Shea, Beck, Halbert, Schmucker 86].

Rosson and Carroll agree with this cost to learning. For example, the large class library is daunting and a major obstacle for users to overcome, users feel that they need to know everything. Although intrigued by the language and environment, they are overwhelmed. Users either learn slowly, by Rosson and Carroll's measurements, or they give up.

Experienced programmers expect a program to have a beginning and an end, and rely on understanding a program by reading it. But Smalltalk programs consist of communicating objects; understanding a program means knowing which objects are involved and what protocol is being used. Understanding Smalltalk code is similar to reading a hypertext document because the code is distributed, and the behaviour of an

object depends on its state. Additionally, the large number of classes and methods makes it difficult to know where to begin to learn.

One advantage of Smalltalk is that the language and the environment are integrated. Powerful tools such as the Browser, Debugger, Inspector, and Disk Browser are vital but the new user must learn how to use them effectively.

Rosson and Carroll complain of the user interface *tease*: Smalltalk is good for rapid prototyping but the MVC paradigm is the most complex in the system. They question how a user of Smalltalk can be satisfied building an application without a user interface. Users are warned to avoid the MVC paradigm until they are more experienced.

We agree with most of Rosson and Carroll's observations about using and learning Smalltalk. They have proposed solutions which we discuss in Chapter 3.

2.3 Summary

Although each researcher cited particular difficulties in learning Smalltalk, the general consensus among researchers falls into four categories:

1. Class library

Presenting a filtered view of the large class library to learn only the necessary components is a difficult but not impossible task.

2. Interactive tools

The Browser, Inspector, Debugger are helpful and necessary tools but time and effort is required to learn how to use them efficiently.

3. Understanding object-oriented program code

The distributed nature of the code makes it difficult to understand a program. Smalltalk suffers from the Yoyo problem: following a chain of message sends often involves going up and down the hierarchy several times [Taenzer, Ganti and Podar 89].

4. Locating objects

The large class library and the distributed nature of the code makes it difficult for students to find the classes and methods needed for a program. Users must find what reusable components are available and then learn how to use them.

5. MVC paradigm

The MVC paradigm is the most complex code in the system. Most of the researchers include the MVC paradigm in their teaching curricula; this creates some of their difficulties.

With modifications to the teaching curricula and the tools, the task of learning Smalltalk can be simplified. None of the difficulties identified immobilize the students; they merely decrease the rate at which students learn. To make Smalltalk easier to learn, the Sherpa system has addressed most of the problems discussed in this chapter.

3 Related Work

In this chapter, we discuss program visualization, animation and teaching systems.

We are interested in program visualization systems because they use graphics to illustrate a program. "It is commonly acknowledged that the human mind is strongly visually oriented and that people acquire information at a significantly higher rate by discovering graphical relationships in complex pictures than by reading text."¹⁴

We are interested in animation because one of our objectives is for students to have fun while they learn. In our experience, students enjoy working with animated programs. This is evident from the popularity of video games.

Sherpa is a teaching system. We examined several state-of-the-art systems for teaching object-oriented concepts. Our goal was to identify features that have been successful, features that need to be improved, and features that are missing.

3.1 Program Visualization

A program visualization system uses graphics to illustrate some aspect of a program such as its run-time execution. Either the code or the data can be illustrated, dynamically or statically. A dynamic illustration is an animation of a running program; a static illustration contains snapshots of a program at certain points [Myers 88]. The goal of program visualization is to help programmers form clear and correct mental images of a program's structure and function [Brown, Carling, Herot, Kramlich, and Souza 85]. The visual representation of a program's execution is generally effective for helping a programmer understand what a program is doing. Program visualization is effective not

¹⁴ [Reader 85], page 12.

only in program debugging, but also in the design, maintenance, and teaching of software [Isoda, Shimomura and Ono 87].

Raeder [Raeder 85] and Myers [Myers 88] discuss the state of the art in graphical programming techniques. They survey recent systems and attempt to classify them. This is a difficult task as many systems cross the borders between visual programming and program visualization, compiled and interpretive, static and dynamic, and code and data oriented. Many systems also combine goals such as algorithm animation, tracing, debugging, teaching, and program understanding.

3.1.1 Debugging with Visualization

Program debugging tools traditionally combine the activities of process visualization, state selection and process modification. In process visualization, users select specific objects to inspect; the objects' values are displayed, almost always textually. Traditional, non-visual debuggers return character based information such as traces or dumps of values and variables. Users typically have little control over the visual representation of data. A trace of the method execution history is often visualized as well.

In state selection, users either predict where the points of interest will be in the program execution or work in single-step mode which traps the program after each instruction. Predicting points of interest assumes that users are familiar with the program. Single-stepping can be slow and restrictive.

In process modification, the process state can be changed by the user. However, the modified code no longer corresponds to the original source code - the user must maintain the consistency between the program and the computation [Moher 88].

Debuggers that attempt to satisfy students of all skill levels often add so many features that beginning students are confused. An example of this is the visual debugger called VIPS (Visual Interactive Programming Support) which graphically displays up to eight simultaneous views of program execution [Isoda, Shimomura and Ono 87]. The task of finding the correct blend of simplicity and flexibility is difficult. To identify features useful for beginning students using Sherpa, we investigated various debugging tools.

PROVIDE

An experimental tool being developed at the University of Illinois combines the benefits of program visualization and debugging. PROVIDE [Moher 88] is a prototype environment designed for the visualization and debugging of programs written in a subset of C. Moher's goal was to create a system to observe and control program execution at a level of abstraction chosen by the user. Graphics are used to represent process states. The environment is geared toward debugging-in-the-small. Beyond its debugging capabilities, it may be used for teaching, for training new users with existing systems, and for code walkthroughs.

PROVIDE has a useful feature called *deferred-binding picture definition*: the view of program execution can be modified by the user at any time. Pictures are continuous displays of program objects that are constructed by binding objects to picture types predefined in a library. A one-to-one mapping does not exist between objects and pictures: one object may drive several pictures and a picture may be driven by more than one parameter. In PROVIDE, an object's picture only survives while the functions

containing the object are active. This is contrary to BALSA [Brown and Sedgewick 84] which allows the user to control how long an object's picture persists.

We agree that pictures are the desired view of objects. Thus we have adopted the idea of graphically representing objects to animate them. In Sherpa, we compromised the approaches taken in PROVIDE and BALSA by making the picture for an object persist only as long as the object does. We keep a library of predefined pictures so that new objects have a default picture when created; users can subsequently change an object's picture.

3.1.2 Teaching with Visualization

Some people try to use debugging tools to teach what programs do, but "debugging is different from teaching".¹⁵ The BALSA system [Brown and Sedgewick 84] is an algorithm animation system that was designed specifically for teaching the dynamics of data structure manipulation; it was not designed for debugging. Instructors add graphics-generating procedure calls to their programs resulting in a sequence of pictures that can be retained for forward and reverse display. Setting up an animation sequence is similar to preparing a lecture: decisions are made before data is presented.

The elegance of BALSA is the ability to create real-time illustrations. This contrasts with the "Sorting out sorting" film [Baecker 81], that also uses animation to teach data structures and algorithms but was created from a series of static pictures. By comparing BALSA to Baecker's film, we see the flexibility of using real-time animation in teaching. Instructors can easily modify their curriculum as needed and can spontaneously teach and illustrate different concepts in class.

¹⁵ [Moher 88], page 852.

3.1.3 Object-Oriented Visualization Tools

Most program visualization tools combine static and dynamic techniques with code and data graphical techniques to teach any combination of: computer science concepts; program understanding; and complex debugging. Browsing and tracing techniques are typically used to aid with program analysis and debugging.

The object-oriented tracing tools have the same problems as those designed for traditional programming languages: it is difficult to get the correct amount of information at the level of detail desired to understand the problem at hand [Boecker and Herczeg 90]. Tracing typically displays only part of a program execution using some sort of filtering criteria. For example, Brown's PV prototype [Brown, Carling, Herot, Kramlich, and Souza 85] and Cunningham diagrams [Cunningham and Beck 86] trace and illustrate only the message passing structure in object-oriented programming. They lack the detail necessary to fully analyze a program.

Although the Smalltalk environment includes many tools, it lacks a satisfactory tracer. Browsers and Inspectors can be used to obtain a static view of objects and their protocol; Debuggers can be used to inspect and modify objects during computation. However, a tracer or a visual Debugger can uncover the dynamic properties of Smalltalk programs. A tracer is "a core ingredient of sophisticated programming environments".¹⁶ The two tracers discussed here influenced the design of the AnimatedTracer in Sherpa.

¹⁶ [Boecker and Herczeg 90], page 415.

TRACK

Boecker and Herczeg [Boecker and Herczeg 90] have designed a visual trace construction kit called TRACK in which users interactively construct environments to trace the execution of methods and the flow of objects in Smalltalk-80. Boecker and Herczeg's goal is to provide insight into the structure and execution of object-oriented programs. TRACK uses visualization techniques to aid programming, debugging and program understanding.

In TRACK, both trace specifications and output are graphic. A user sets up a program trace by direct manipulation of graphical objects. The result is a visualization of the program's dynamic behaviour. Users control the traces that are running: they can set up several traces to run simultaneously, they can set breakpoints, and they can change the scope to focus on points of interest. Prefabricated programs as well as programs written by the user can be animated.

All message sends are displayed textually in the global trace window. A ball moving from sender to receiver graphically displays a message send. Users can choose to execute the messages as an animation or in single-step mode. See Figure 1 [Boecker and Herczeg 90]: p. 422.

TRACK is a good tracing tool but it is too advanced for the novice programmer. Users require a sound understanding of programming and object-oriented concepts to set up a trace in TRACK. They need to know what information is of interest to trace, what is needed to understand a program, and what is extraneous. A novice programmer does not have this knowledge. As Sherpa is designed for novices, our tracer gives students little flexibility. Students are limited as to what they can do; this avoids confusion when students are confronted with an array of choices.

Figure 1: TRACK

GraphTrace

GraphTrace [Kleyn and Gingrich 88] is a tool to help users understand large-scale object-oriented programs. It is similar to London's animations [London and Duisberg 85] and Cunningham diagrams [Cunningham and Beck 86]. Users control the different aspects of a program to be displayed. GraphTrace displays graph diagrams consisting of **nodes** which represent various system components, and **links** which represent the relationships, such as inheritance, between components. Although GraphTrace claims to animate programs, we consider it to be more of a tracer. To understand a program, users view a combination of textual message sends with graphical displays in a tree structure. The highlighting of nodes and links in the graphs gives the appearance of movement but the system components do not appear lifelike, part of the definition of animation. See Figure 2 [Kleyn and Gingrich 88]: p. 194.

The user has flexibility with GraphTrace as far as controlling the animation speed, including single-step mode. There are also several graphs for viewing an application such as the method invocation graph to show the sequence of methods called during a program's execution. The key feature of GraphTrace that influenced Sherpa was the sequence of computation displayed under the user's control. Showing the sequence of message sends helps students understand the relationships between objects in a program.

Figure 2: GraphTrace

3.2 Animation

"One learns by doing and one does Smalltalk by browsing the work of other programmers".¹⁷ Much of learning object-oriented programming concepts involves browsing other programmers' source code and trying to understand it. Some researchers have concentrated on tools and design methodologies that help users understand object-oriented concepts. Most of these tools such as class diagrams [Wilson 90], class and object diagrams [Booch 90], and Responsibility Driven Design [Wirfs-Brock 90], are static. From Balsa [Brown and Sedgewick 84] and the Animation Kit [London and Duisberg 85], we see the flexibility of using real-time animation and the benefit of dynamically representing the changing state of objects for teaching. Thus we wanted to use animation in Sherpa to teach object-oriented concepts. We investigated tools that create dynamic displays.

3.2.1 Cunningham Diagrams

Cunningham and Beck developed a notation to represent the message sending dialogue that takes place between objects engaging in a computation [Cunningham and Beck 86]. They have used their diagrams to teach beginning and advanced students. Cunningham and Beck conclude that the diagrams can help teach the concepts and practice of object-oriented programming to naive users. A program that emphasizes the dialogue between objects is easier to understand by examining diagrams; it is more difficult to understand by simply examining source code.

¹⁷ [Cunningham and Beck 86], page 361.

Figure 3 shows a typical Cunningham diagram. Each object participating in the computation is represented by a box; an object is identified by its class. A message send is drawn as a directed arc terminating in the receiving object. The selector of the method invoked by the message-send is placed at the receiving end of the arc. State and sequence information is omitted from the diagrams.

Figure 3: Cunningham Diagrams

When an inherited method is invoked, the receiver is split into layers with the superclass below the receiving class. An arc travels deeper through the layers until it finds the class that implements the method; the **Object** class is at the bottom of the hierarchy. Displaying the subclass above the superclass contradicts the elevation metaphor implied by the terms but the authors believe that method look-up should be thought of as searching deeper. They also admit that their diagrams did not look good drawn the other way.

Collecting the information needed to construct the diagram requires at least as much computation as done by the Debugger. Cunningham and Beck added an editor to the Debugger. When constructing a diagram, the editor uses the Debugger's computation

data to display each message send, one at a time, on the same diagram. Once the series of message sends convey the computation in one diagram, the diagram can be saved and redisplayed. A diagram is an accumulation of information throughout a debugging session. It does not show the sequence of computation and does not display the context of the message sends. Cunningham and Beck believe that the intermediate behaviour displayed during the dynamic creation of the Cunningham diagrams is useful information and should be recorded. They believe that playing back the series of diagrams will add further insight into understanding a computation.

Cunningham diagrams had a major influence on Sherpa. Although the diagrams are not animated, the potential of animating them inspired us to investigate further. We also realized the benefit of using the Debugger in Sherpa's tracer because the Debugger already has the information from a computation.

3.2.2 Smalltalk Animation

The Animation kit by London and Duisberg is a toolkit used to create graphical snapshots, animations and movies for explaining programs and algorithms [London and Duisberg 85]. The kit creates the animations in real-time. London and Duisberg implemented the kit in Smalltalk because it is an open system; they could exploit a more general and productive environment that contained graphics and animation tools. The Smalltalk system simplified their exploratory and prototyping research. Although the kit does not demonstrate object-oriented concepts or message protocol as the Cunningham diagrams do, the ideas behind its design influenced the design of Sherpa.

The Animation kit was designed to help users understand what programs do, how they work, and why they work. The kit contains a set of tools, which from the authors' experience, are easily learned and applied by students to study algorithms and to create

animations. Students view a graphical representation of an algorithm: an abstract representation of the data structures' internal state with secondary information omitted to avoid distraction and clutter.

Of particular interest to us is the choice of pictures displayed. The Animation Kit uses the type of diagrams that programmers typically sketch at a desk as well as using the schematic diagrams used in programming and data structure textbooks.

Another feature of the Animation kit that influenced Sherpa is the transition between changing states. The kit shows smooth transitions between states with built-in delays to compensate for the real-time speed of the animation. The kit avoids the two extremes of showing changed states: refreshing the screen completely; or flashing a change on the screen. The former is too time consuming; the latter requires mental effort to deduce the evolution of the pictures.

3.3 Teaching Systems

Program visualization tools are only part of what is needed to help users learn object-oriented programming concepts. In Sherpa, we want a development environment which includes program visualization tools along with a learning curriculum to teach users. Little research has been done to improve the learning environments and to develop curricula. A few systems have been designed for children but they have not been extended for adults.

3.3.1 Children

The first weeks of a programming course are crucial to students' perception of computer science. They are receptive to new ideas and learn to appreciate the discipline. We looked at Karel the Robot [Pattis 81], Playground [Fenton and Beck 89], and Logo [Papert 80], systems that teach children about computers and programming, because they typically have a slow yet exciting introduction, and they do not assume any previous programming experience. We do not discuss Logo in detail as the other systems are similar.

Karel the Robot

Karel the Robot [Pattis 81] is a non-hostile environment to be used at the beginning of an introductory programming course to help students learn how to program more intuitively. The common fear students have of computing and anything mathematical in nature is addressed. Students learn the vocabulary of technical terms and explore a simple yet rich domain of problem solving and programming with little overhead.

Karel is a Robot that lives in a simple world. Students learn how to write programs by giving Karel tasks to perform in certain situations. Karel's programming language is similar to Pascal and thus has positive transfer effects for students eventually learning Pascal. Students learn concepts such as conditions, constructs, procedural programming, problem solving, and stepwise refinement.

The results of executing a program for Karel are visual. Karel is drawn as half moon object in a grid of streets bordered by 2 walls. The grid may contain special objects that Karel can sense and manipulate, as shown in Figure 4 [Pattis 81]: p.3. Karel can move in any direction and can *see* left, straight and right. Students write programs

and procedures with five primitive commands: move, turnleft, pickbeeper, putbeeper, and turnoff. With visual feedback, students see their programs running. They learn that they can make Karel perform complicated tasks with a series of simple statements.

Figure 4: Karel the Robot's World

Playground

Playground is part of the Apple Computer Vivarium Project started in 1986 by Ann Marion and Alan Kay to investigate the phenomena of learning [Fenton and Beck 89]. Playground is an object-oriented programming environment for children; the children build simulations using graphical objects that must obey certain laws. A biology metaphor was adopted because it proved to be useful for explaining complex dynamic systems.

Children write programs by creating graphical objects on the screen and then selecting, manipulating, and inspecting them. Each object is an agent that acts according to specified agent rules. The execution of an agent rule may cause objects to react by moving or changing appearance. Through animation code, an object's display is updated on the screen giving the effect of a real-time presentation.

The children choose objects, such as artificial animals, from a predefined library. More than one object of the same type can be displayed. A set of predefined behaviours that follow naive physics can be assigned to any object by the user. Objects can sense the presence of other objects and can be made to interact with each other, for example, in the design of a game or animation. The motion of an object follows the rules of turtle geometry defined in Logo [Papert 80]. As in ARK [Smith 86], it is possible to introduce objects which add physical laws to the environment that all objects must obey.

Figure 5: Playground

The results have been encouraging so far; children are able to construct simulations of surprising complexity. Fenton and Beck hope to create a general purpose language for personal computer users when they discover how to apply the Playground programming style to desktop programming problems. See Figure 5 [Fenton and Beck 89]: p. 126.

Discussion

Although Karel the Robot, Playground and Logo were designed to teach children, they have many features that are useful for teaching the basic object-oriented concepts to adults. They teach students how to create objects, how to modify both the state and behaviour of objects, how to write small programs by creating animations, how to use the environment, and how to have fun programming. Objects such as turtles, robots and animals are fun and interesting to work with. Some researchers may argue that students should learn by performing meaningful or real world tasks, but students enjoy learning things that are fun.

In our Sherpa system, we have adopted some of the features of the systems discussed above. We believe that animated objects make it easier and more exciting to learn object-oriented concepts and Smalltalk. Sherpa's introductory environment, called the GraphicWorld, is based on animated objects that students can *play* with to *see* results. The GraphicWorld was also inspired by a project done at Carleton University¹⁸.

¹⁸ Kevin McGuire. Thing Browser. Technical project, Carleton University, School of Computer Science, 1990 (unpublished).

3.3.2 Adults

Although adult learners eventually accept the effort involved in learning Smalltalk and begin to appreciate the increased power and flexibility, they still need motivation to continue climbing the mountain. According to Rosson and Carroll, learners need an introduction that is geared toward quickly doing interesting and meaningful work [Rosson and Carroll 90].

IBM T.J. Watson Research Group

A group at the IBM T.J. Watson Research Center is researching the use and teaching of Smalltalk. The group is interested in how people begin programming with Smalltalk. They have targeted experienced procedural programmers, users of languages such as C, Pascal, and Basic, who are familiar with object-oriented concepts: the users know about message-passing and inheritance but they have not designed an interactive Smalltalk application. The group has proposed several techniques to help new users learn the object-oriented paradigm: [Campbell 90], [Campbell, Brown and DiBello 90], [Carroll, Singer, Bellamy and Alpert 90], [Carroll 90], [Gold and Rosson 90], [Rosson and Carroll 90], [Rosson, Carroll and Bellamy 90], and [Singley and Rosson 90].

One of the group's research project applies the Minimalist model of training and instruction to develop a teaching curriculum. "Minimalism emphasizes the streamlining of instructional materials, and organizing the content of manuals, tutorials or other educational artifacts."¹⁹ The goal of this project is for users to accomplish meaningful

¹⁹ [Rosson and Carroll 90], page 77.

tasks quickly; the project exploits the users' previous knowledge to help them learn a new programming environment.

Rosson and Carroll's teaching scenario requires tools. They designed a tool called Bittitalk to help students learn the large class library. Bittitalk is a filtered view of Smalltalk containing only selected classes and methods. The rationale is that it is easier to start by learning a small subset of the library, thus Bittitalk hides what is not initially needed. By conducting experiments, they found that fewer than 10% of the normally visible methods are relevant for initial learning. The Bittitalk library subset is viewed through the standard Class Hierarchy Browser. The existing Debugger and Inspector both reflect the filtered Bittitalk view. An application specific Bittitalk view is discussed but the ease of setting it up is not.

To complement Bittitalk, the group developed the View Matcher [Carroll, Singer, Bellamy and Alpert 90], a tool to help learners follow the messages sent between objects in a program. The View Matcher, as shown in Figure 6 [Carroll, Singer, Bellamy and Alpert 90]: p. 433, is a browser that presents integrated and dynamic views of a running application. The View Matcher window contains an Inspector view, a commentary view, an application view showing the user interface of the application, a Class Hierarchy Browser showing only the classes pertinent to the application, and a stack view of message sends similar to the walkback of the debugger. As the application executes, changes are reflected immediately in the multiple views. The user should be able to apply the experience gained using View Matcher because the View Matcher views are similar to those of tools in the Smalltalk environment.

The View Matcher is designed for self-directed learning. A user learns how a running application corresponds to its implementation by observing the message sends, state changes, commentary, and changing user interface. The user can work in two modes: halt mode and animate mode. In halt mode, the application stops at breakpoints and updates each of the views. These breakpoints are chosen by the designer of the particular View Matcher application; thus the designer decides what is important for the user to learn. In animate mode, the user watches an application execute without interruptions; it is useful for analysis. Only the information in the stack view and the application view is updated. The entire display is not updated because the number of visual changes would be too confusing for users to follow.

We believe that the View Matcher is a useful teaching tool. We found the following features of particular interest:

1. It displays only the pertinent classes; this makes it easier to grasp and navigate the class hierarchy.
2. It displays information about a running application in a message display stack and in an Inspector; this is useful for tracing.
3. It presents views of objects similar to those presented in other Smalltalk tools; this makes it easier to learn how to use the other tools.

However, we have reservations about the View Matcher:

1. The design assumes that a multi-view window of a running application makes it easier for the user to understand the entire application [Carroll, Singer, Bellamy and Alpert 90]. We believe that the multi-view window forces the user to deal with too much information.
2. It is used for self-directed learning. We believe that in the initial stages of learning Smalltalk, students need guidance and practice working in the environment.

In [Rosson and Carroll 90] and [Rosson, Carroll and Bellamy 90], they discuss an introductory Smalltalk curriculum that is based on the minimalist model of instruction and that uses both Bittalk and the View Matcher. They have challenged themselves to develop a curriculum which includes only four hours of instruction. At the end of four hours, a learner should be able to develop the code for an interactive application. The group's audience is experienced; the curriculum exploits prior knowledge of real world tasks and computer concepts such as stacks.

The curriculum is a self-instruction package; the only hardcopy documentation is an interaction tips sheet. Users are encouraged to look for help on-line instead of in manuals. The on-line help provides:

1. textual explanations of what a user should expect after sending a message
2. suggestions for anticipated problems
3. a crude error recovery mechanism.

The instruction incorporates a spiral curriculum²⁰: an introduction to Smalltalk with stages that successively reveal more of the underlying system. The group has experimented with five instruction stages using a Blackjack game as the application; people are familiar with the game and would not have to waste time learning it.

In the Blackjack Game, learners are first introduced to the model of the game. The code is hidden to emphasize the message passing paradigm; there is no need to see how an object performs its function. The curriculum then spirals through the application several more times adding more information. The second stage adds a user interface to the application; it is displayed in the View Matcher. The third stage uses Bittalk to view the source code for only the model and its messages. The fourth stage simply adds the source code for the entire application. The final stage is an open-ended interaction of

²⁰ Reference to [Bruner, Goodnow, Austin 56] in [Rosson and Carroll 90], p78.

multiple applications with all source code available to the learners. In keeping with the idea of providing interesting and meaningful tasks for learners, three other applications, gomoku, tictactoe, and a recipes database, have been developed to follow this curriculum.

To help students reify plans and goals, Singley and Carroll [Singley and Carroll 90] are designing an instructional system for Smalltalk that also applies the minimalist approach. The system is a tutor containing a number of planning tools to help students solve real-world tasks. Students are tutored, under their own control, and are slowly introduced to an untutored environment. The current tutor contains four components: Video Guru, Goal Poster, Commentator, and Adaptive Index.

The Video Guru presents planning information to students. An audio commentary accompanied by an on-line video for visual demonstrations is used to explain coding and underlying design goals.

The Goal Poster maintains a current goal tree for students. The tutor writes to a goal blackboard for two reasons: to record a textual trace of the audio commentary, and to record its interpretation of the students' actions such as writing code, or opening a browser. For example, if the tutor interprets that a student is making an error, it would indicate this on the goal blackboard.

The Commentator helps students understand programs and code reuse; it provides students with context sensitive explanations of chunks of code. The tutor can also automatically add its own comments to code written by students. Students are encouraged to modify these comments as necessary.

The Adaptive Index helps experienced procedural programmers find methods in the large class hierarchy. It maps Smalltalk functions to procedural ones. Students enter a familiar procedural or computer term into a query system. The result is a set of pointers to appropriate methods in the hierarchy. This tool is of particular interest as it is also

useful in Sherpa for students without experience; they could simply guess the name of a desired method by relating it to their real-world knowledge. Given that the cross-referencing in the adaptive index is comprehensive, the struggle of searching through the library for a method is reduced.

To understand how users learn to use applications and how they become experts, Campbell believes a long-term developmental approach is needed [Campbell 90], [Campbell, Brown and DiBello 90]. Campbell's approach studies users as they develop skills and conceptual understanding over time. The learning stages between novice and expert are analyzed qualitatively rather than quantitatively. For example, the skill level of a user is not measured by how quickly the user can perform the skill.

Data gathered about learning Smalltalk forms the basis of the group's proposal for a model of developmental levels for Smalltalk programming. The order of learning levels is preliminary; they are based on limited data. The seven levels are:

1. Interacting with the visual interface
2. Syntax rules and order of precedence
3. Locating classes and methods in the hierarchy
4. Class versus instance distinction
5. Model-View-Controller
6. Object-Oriented design
7. Grandmaster level

It is important to create a wide range of tasks for all user skill levels. Two distinct streams are used in the model: issue-based and proficiency-based. The issue-based stream forms a *soft sequence* in which only the beginning of each level must be followed in order. The proficiency-based stream forms a *hard sequence* in which learners must complete one level before moving to the next. In both streams, learners can work on tasks at different skill levels; an array of tasks to accommodate the range of user expertise is to be performed at each level. Tasks aimed at *typical* users are not created because users attempt tasks with different strategies depending on their experience.

The research group at IBM has found that the developmental approach to learning how to program is feasible. There is a tremendous difference between the tasks undertaken at the different levels. The mythical average programming task either flattens these differences or favours one level of expertise.

Discussion

Many ideas from the work done by the T.J. Watson research group influenced the design of Sherpa. We found the idea of a filtered view most useful, particularly that the filtered class library is reflected in other views such as the Inspector and Debugger. In Sherpa, we adopted the idea of a range of tasks to accommodate the different learning rates of students. The environments are simple enough for the beginner and are not so restrictive as to slow down the advanced learner. We do not follow Campbell's teaching order. Based on our teaching experience, we propose an alternative order in Chapter 5.

In their research, Rosson and Carroll discuss several applications other than the BlackJack game. However, they do not discuss the time and effort required to incorporate a new application into the curriculum. If students are to be taught using many examples, we believe an important feature is to be able to integrate new applications into the curriculum quickly and easily. In Sherpa, we provide tools to easily add new applications to a curriculum.

In [Rosson and Carroll 90], they claim that after four to five hours, users have a good initial grasp of the Smalltalk user interface paradigm; users are able to suggest enhancements and ways to implement them. From our experience both using and teaching Smalltalk, we find the pace of this five-stage curriculum too fast for beginners.

The first of the five instruction stages is good: creating objects, sending messages and observing results; the rest of the stages are too fast.

3.4 Summary

The research done with program visualization, animation and teaching systems has inspired many ideas for the design of the Sherpa system for both students and instructors.

In particular, we adopted the following strategies for the Sherpa system:

1. Make the system interesting and fun for beginners.
2. Use graphics.
3. Make tools simple yet flexible for both students and instructors.
4. Make it simple for instructors to tailor the system to suit their curriculum.
5. Handle multiple instructors with different styles.
6. Use dynamic representation of computation with contextual information.
7. Use a gradual, incremental approach for introducing new concepts.

4 Design

In this chapter, we discuss the design and implementation of Sherpa: a prototype development environment to make Smalltalk easier for students to learn, and easier for instructors to teach. The experiences of Smalltalk users discussed in Chapter 2 form the basis of our improvements to create a new working environment. We added two new tools, the GraphicWorld and the AnimatedTracer, to help students learn the basic concepts of objects, instantiation, and message sending. We simplified some of the existing tools, namely, the Browser, the Inspector and the Debugger. To help students and instructors work in Sherpa, we added a primitive on-line help system. Figure 7, Sherpa versus the base Smalltalk environment, shows the new objects introduced, namely the Universe, the Galaxy and the World, and the new tools developed to view and modify these objects.

Students are expected to start learning Smalltalk in the Sherpa environment and are then gradually exposed to the Smalltalk environment that exists today. To ease the transition, the user interface for all Sherpa tools was designed to be consistent with the existing Smalltalk interface. Students who learn with Sherpa and switch to Smalltalk should not be overwhelmed.

There are two working environments in Sherpa: one for students and one for instructors. The student Sherpa is a filtered view of Smalltalk supplemented with our new tools for learning. The instructor Sherpa is the complete Smalltalk environment supplemented with tools for modifying the student Sherpa. We have designed flexible tools for instructors to tailor the student environment to correspond with the presentation of course material.

Normal Smalltalk

Sherpa

Figure 7: The Overall Sherpa System versus The Normal Smalltalk

4.1 The Student Sherpa Environment

4.1.1 The Worlds

As discussed in Chapter 2, in the early stages of learning Smalltalk, most students find the class library too large to learn. When exposed to over 100 classes and 2000 methods in Smalltalk/V286, students do not know where to begin or what is important to learn. Researchers such as Borning and O'Shea have proposed that the hierarchy be restructured so that students can gradually learn subsets that build on each other [Borning and O'Shea 87]. We agree that gradually learned subsets are beneficial to new students but we do not want to restructure the library. Rosson and Carroll discuss their idea and usage of Bittitalk: a filtered view of Smalltalk showing only a subset of classes and methods in the Class Hierarchy Browser [Rosson and Carroll 90], [Rosson, Carroll and Bellamy 90]. We agree that a filtered view of Smalltalk is beneficial but it is not enough; the view must be flexible and available through a simplified Browser. A combination of these two ideas has led us to develop the concept of a World.

A World is an object developed to encapsulate related concepts and class protocol. Information outside a World is hidden from students. Browsers, Debuggers, and Inspectors can all be set up to reflect only the information in a World.

A World contains a subset of the Smalltalk class library. It can be built from any number and combination of classes that are in the library, including those defined by the instructor. By using a set of Worlds that build on each other, an instructor teaches the class library to students incrementally. The Worlds are introduced as slowly or as quickly as desired.

When a class is added to a World, the default is to hide all of its instance and class methods. Once a class is in a World, any number of its methods can be added with one of the following states: display source code or display comment only. This flexibility allows instructors to teach students the protocol of a class without showing the source code implementation. This is an important feature because many students believe that they must see the source code of a message in order to understand it. Our experience indicates that students are slowed down by trying to learn all of the detail. We are teaching them about information hiding: knowing the messages that an object recognizes and knowing what each method does by reading the comment is sufficient information to use an object successfully.

The same mechanism for hiding methods is used for instance and class variables of a class; they are hidden by default. An instructor simply adds a variable to a World to make it visible. When students browse a class, the class definition includes only the variables the instructor intended the students to know about. This feature is important because some variables are simply a source of confusion for students. An instructor can also make inherited variables visible without making the superclass visible; students may need to know about a variable without knowing the class in which it was defined.

In Figure 8, we illustrate a possible Pen World for teaching students how to use Pens with and without Sherpa. The Pen World contains 2 out of 18 instance variables, 0 out of 1 class variables, 14 out of 71 instance methods, and 0 out of 4 class methods - this includes inherited variables and methods. Instructors may initially choose to hide the source code for all of the methods until students are comfortable using pens. Once comfortable, the students could be introduced to FullPenWorld, a World in which source code is shown.

Pen World in Sherpa

```
class Pen
```

```
instance variables
```

```
direction
```

```
downState
```

```
class variables (None)
```

```
instance methods
```

```
black
```

```
defaultNib:
```

```
direction:
```

```
direction:
```

```
down
```

```
go:
```

```
goto:
```

```
home
```

```
location
```

```
north
```

```
place:
```

```
turn:
```

```
up
```

```
white
```

```
class methods (None)
```

Pens without Sherpa

```
class BitBit
```

```
instance variables
```

```
clipHeight
```

```
clipWidth
```

```
clipX
```

```
clipY
```

```
destForm
```

```
destX
```

```
destY
```

```
halfTone
```

```
height
```

```
rule
```

```
sourceX
```

```
sourceY
```

```
sourceForm
```

```
width
```

```
class variables (None)
```

```
instance methods
```

```
clipRect
```

```
clipRect:
```

```
clipX
```

```
clipY
```

```
combinationRule:
```

```
copyBits
```

```
destForm:sourceForm:h...
```

```
destForm
```

```
destForm:
```

```
destForm:sourceForm:
```

```
destOrigin:
```

```
destRect:
```

```
destX
```

```
destX:
```

```
destY
```

```
destY:
```

```
drawFrom:to:
```

```
drawLoopX:Y:
```

```
drawTo:
```

```
extent
```

```
extent:
```

```
height
```

```
height:
```

```
hideCursor
```

```
mask
```

```
mask:
```

```
sourceX
```

```
sourceX:
```

```
sourceY
```

```
sourceY:
```

```
sourceForm:
```

```
sourceRect:
```

```
sourceOrigin:
```

```
sourceForm
```

```
width
```

```
width:
```

```
class methods
```

```
bitBltError:
```

```
destForm:sourceForm:
```

```
class Pen
```

```
instance variables
```

```
direction
```

```
downState
```

```
fractionX
```

```
fractionY
```

```
class variables
```

```
DoubleCenter
```

```
instance methods
```

```
black
```

```
bounce:
```

```
centerText:font:
```

```
changeNib:
```

```
curveFrom:to:towards:
```

```
defaultNib:
```

```
direction
```

```
direction:
```

```
down
```

```
dragon:
```

```
drawRectangle
```

```
ellipse:aspect:
```

```
ellipsePrim:aspectX:Y:
```

```
fillAt:
```

```
fillAtX:andY:
```

```
frame
```

```
frame:
```

```
go:
```

```
goto:
```

```
gray
```

```
grid:
```

```
home
```

```
initPen:
```

```
location
```

```
mandala:diameter:
```

```
north
```

```
place:
```

```
polygon:sides:
```

```
rectangleAt:length:width:
```

```
reflectDrawX:Y:
```

```
solidEllipse:aspect:
```

```
spiral:angle:
```

```
turn:
```

```
up
```

```
white
```

```
class methods
```

```
new
```

```
new:
```

Figure 8: PenWorld in Sherpa versus Pens without Sherpa

4.1.2 World Views

A World is simply an object in Smalltalk. Students need simple tools to view the contents of a World. We have designed two views: graphic views and source code views. The graphic view is called a GraphicWorld and is used to display classes that are meaningful when visually represented; classes that are abstract, such as the HomeContext class, are difficult to visualize and thus inappropriate for the GraphicWorld. There is no tool similar to the GraphicWorld in the existing Smalltalk system. The source code view is called a WorldBrowser and can be used to display all classes. It is based on the Class Hierarchy Browser shown in Figure 9.

Figure 9: Class Hierarchy Browser in the Smalltalk System

4.1.3 The GraphicWorld

The GraphicWorld was designed to be the first view of the Smalltalk system that students have. For students to get a good first impression of the system, we believe that the introduction should be fun and reduce fear. We base our design on Logo-type environments such as Karel the Robot [Pattis 81] which have helped students learn how to program using pictures.

In the graphic view of a World, students do not see source code. They see a list of the classes in the World, the protocol for each class, a list of existing instances of each class, and a graphic representation of each instance. They are graphically introduced to instantiation, message protocol, inheritance, global variables, assignment, and data encapsulation. When students do something in the GraphicWorld, they *see* the results. For example, by sending a message to an object, students observe the changes in the object pictorially. An empty GraphicWorld window is shown in Figure 10. A GraphicWorld open on a RobotWorld containing two Robot instances, Quincy and Tommy, is shown in Figure 11.

Figure 10: A GraphicWorld Window

Figure 11: A GraphicWorld open on a RobotWorld

In this World view, students create instances, assign them to global variables, and send them messages. For each instance created, a graphic object is drawn on the screen. This object can be inspected using the Viewer, a tool described in Section 4.1.7. As no source code is shown, students learn message protocol by watching the graphic objects change as a result of a message send, and by reading the comment available in the messages pane. An example for the square message is shown in Figure 11.

In our example World, the superclass of Robot is included to teach students about inheritance. By experimenting, students learn that messages recognized by Robots are defined in the Robot class, or are inherited from the Player class. Another concept learned in the GraphicWorld is that classes can have multiple instances distinguished here by their name.

Note in Figure 11 that *Tommy's* picture is different from *Quincy's*. All objects in the system know how to draw themselves. We have defined a PictureLibrary to store the pictures of specific objects, both specific instances and classes. Students can define a picture for any object using a BitEditor as shown in Figure 13. When asked to display itself, an object first checks for a picture in the picture library. If it does not exist, the object checks if its class has a default picture. If not, a picture is automatically created using the object's printOn method. Figure 12 shows several example pictures. We base our decisions about pictures on the work done in PROVIDE and BALSA, as discussed in Chapter 3. Our example in Figure 11 shows *Quincy* displaying a predefined picture and *Tommy* displaying the default picture for the Robot class.

Figure 12: Sample default pictures of objects not in the picture library

Figure 13: The BitEditor open on a Robot picture

The first GraphicWorld used by students does not show source code. Once students have exhausted the capabilities of the GraphicWorld, it can be extended to allow students to add new behaviour to objects. Students can add new messages using a WorldBrowser and then immediately test them in the GraphicWorld. This adds a new dimension to the GraphicWorld; students can create miniature applications such as video games while knowing very little about the system.

4.1.4 The WorldBrowser

The WorldBrowser is a view of a World in which students see the classes, methods and variables contained in a World. The WorldBrowser is based on the existing Class Hierarchy Browser (CHB). The visible differences between the two Browsers are shown

in Figures 14 and 15. Students can perform the same operations in a WorldBrowser as in a CHB; they are simply restricted as to what they see and how they see it.

Figure 14: A CHB with class Pen selected

Figure 15: A WorldBrowser with class Pen selected

Some aspects of the standard CHB were changed because they confuse students. The instance/class method button was changed from two buttons to one toggle switch. As a toggle switch, the existence of class methods can be hidden from students when first learning about methods - the switch can be deactivated so that class methods cannot be displayed. The toggle was also moved to the top of the method pane. Being above the list of methods instead of below, it serves as a title describing what is below.

Figure 16: The CHB method pane displaying a class definition and method code

Another source of confusion for students is the method code pane. In the CHB, this pane can display both a class definition and the source code for a method, as shown in Figure 16. If method source is displayed, students may write a new method simply by typing over the old source. But if the class definition is displayed, students can type a new method in the pane only to discover that it will not be accepted when they try to save. This is inconsistent and can be frustrating for new students. The WorldBrowser solves this problem by adding a *class definition* modal window²¹ as shown in Figure 17. The definition no longer appears in the method code pane.

Figure 17: Modal Class Definition Window

21 A modal window does not allow the user to perform any operation outside the window. It must be closed before any other window can be accessed.

The format of the class definition has also been changed; the CHB shows more information than students need at an introductory level. In the CHB, the definition is a message send showing the class and immediate superclass names, the instance and class variables, and the pool dictionaries. The new class definition is not a message send but more of a template as shown in Figure 18. The template shows the class' hierarchy of superclasses contained in the World and does not show the pool dictionaries. By changing from a message send to a template, we have lost the ability to edit the message send to add instance or class variables. These variables must now be added in the modal window.

Class: Player Super Classes: () Instance Variables: (pen picture name) Class Variables: ()	Object subclass: #Player instanceVariableNames: 'pen picture name' classVariableNames: " poolDictionaries: "
---	---

Figure 18: Class definitions in Sherpa and in the standard CHB

Apart from: hiding classes, variables and methods, the WorldBrowser can also hide method source code. Figure 19 shows the method `square` selected and its source code visible in the method code pane. Figure 20 shows the method `goto:` selected and the new format displayed if the source code is not to be shown. The method comment is displayed to tell students what the method does.

Figure 19: A method with source code visible

Figure 20: A method with source code hidden

Students can develop Smalltalk code in the WorldBrowser just as in the CHB; the main difference is that the entire class library is not visible. This may cause a problem if students want to create a class that is already defined in the library, or extend a class with a new method or variable that is already defined. Our solution for this problem is to verify everything that students try to add and not let them overwrite anything. Students are simply advised to choose another name.

Another problem is the visibility of method source code. If a method in a World is set to display its source code, what if the method uses another method that is not contained in any of the Worlds available to the student? It is possible to automatically hide the source code for a method if it contains a message that is not contained in any of the visible Worlds but this may not be the best solution. It may be better to use our idea of a base class list (discussed in section 4.2.1), which currently contains a set of all of the classes that students will need to learn, and extend it to contain methods along the same line. The base list could be used as the criterion for displaying the source code or not.

4.1.5 The Galaxies

When first learning in Sherpa, students work with only one or two Worlds viewed through the GraphicWorld and possibly through the WorldBrowser. They may also use a simplified Inspector and a simplified Debugger as discussed in sections 4.1.7 and 4.1.8 respectively. As instructors gradually teach students more of the class library and assign more complex tasks, students will need to work with several Worlds, possibly grouped in some meaningful way. They also need the existing Smalltalk Inspector and Debugger. To encapsulate the various combinations of Worlds, World views, Inspectors, and Debuggers, we created the concept of a Galaxy. A Galaxy is the students' working environment which contains the following information:

1. a collection of Worlds visible by a WorldBrowser
2. a collection of Worlds visible by a GraphicWorld
3. the available Inspector view
4. the available Debugger view

Students do not control the contents of a Galaxy; they only control the Galaxy in which they work. Students enter a Galaxy from the screen menu by selecting "enter Galaxy" as shown in Figure 21. A Galaxy window opens for the Galaxy selected. This window is similar to the system transcript in that it cannot be closed; this is to prevent students from losing control of the Galaxy by accidentally closing the window. In our current implementation, students cannot have more than one Galaxy open at a time.

Figure 21: Literal Galaxy Window

Figure 21 shows an example Galaxy containing several related Worlds. In this example, each World contains only the class for which it is named. The Galaxy is called the LiteralGalaxy because all of the Worlds contain classes which have literal representations. When students enter a Galaxy, the tools such as the Debuggers and Inspectors, reflect the contents of the Galaxy. Students do not see any classes, methods or variables that are outside the Galaxy.

4.1.6 The AnimatedTracer

As discussed in Chapter 2, during the early stages of learning object-oriented programming, students have difficulty visualizing what is happening during a message send. A tracing tool that simply shows a textual traceback of message sends with a data dump of values is too abstract for students to understand. "Computer programs ... are often best understood by observing graphical simulations of their behaviour. Graphical displays of a program in action expose some properties that we might otherwise find difficult to understand or even notice. Moreover, this communications medium becomes even more powerful if we interact with such animations..."²²

The AnimatedTracer is a tracing tool designed to graphically display all of the objects and actions involved in a message send. From the animation, students learn who the sender and the receiver of a message are, what objects are created, used, and deleted, how parameters are passed, and what results are returned. As a side benefit, they learn the method of stepping through a program, the value of Inspectors, and the precedence rules of evaluation.

The AnimatedTracer is a window as shown in Figure 22. In the tracer, students see the graphical representation of the objects involved in the message send together with the

²² M.H. Brown, Exploring Algorithms Using Balsa-II, Computer 21(5), May 1988, p.14.

code that is being traced. The AnimatedTracer can be invoked in two ways. The first way is to place a breakpoint in a method to indicate where to start the trace. In Smalltalk, breakpoints are typically set by placing the message *self halt* in a method. To be consistent, we created the message *self trace*, shown in Figure 23, to be used exactly as *self halt*. The second way to start a trace is to select the code to be traced and choose "traceIt" from a menu as in Figure 24. This method is consistent with the "doIt" and "showIt" commands used to execute code.

Figure 22: The AnimatedTracer

```

greet: aRobot
  "Move to aRobot's position on the screen, say
  hello, then return home"

  self trace.
  self goto: aRobot location.
  aRobot hello.
  self home.

"Test case: Quincy greet: Tommy"

```

Figure 23: Trace of greet: message

Figure 24: "traceIt" chosen from any text pane menu

The windows that students see when tracing the code in Figure 23 are shown in Figures 25a through 25e. The face in the top left corner represents the *user* and the computer represents *Smalltalk*. These icons were designed to help students understand how the chain of message sends begins. Using an arc labelled "doIt" from the *user* icon to the *Smalltalk* icon, we attempt to convey that executing code with "doIt" or "showIt" sends the "doIt" message with the selected code to Smalltalk. Smalltalk then executes the messages in the selected code one by one - the compiler actually executes the code but we avoid discussing the compiler with students at this early stage.

Figure 25a: Animation of "Quincy greet: Tommy" in the AnimatedTracer

Figure 25b: Animation of "Quincy greet: Tommy" in the AnimatedTracer

Figure 25c: Animation of "Quincy greet: Tommy" in the AnimatedTracer

Figure 25d: Animation of "Quincy greet: Tommy" in the AnimatedTracer

Figure 25e: Animation of "Quincy greet: Tommy" in the AnimatedTracer

A message send is conveyed by drawing an arc, labelled with the message, from the sender to the receiver. If the message has a parameter, it is included in the message label. Graphical objects are drawn for each object involved in the message send, including the parameter. If a message returns a result other than the receiver, a thick arrow is drawn from the base of the receiver's icon to the base of the sender's icon with the resulting object drawn along the arc. Although not in the current implementation of Sherpa, this concept is illustrated in Figure 26.

Figure 26: Objects returned from a message send

In our example, the message `greet:` is sent to *Quincy*, a Robot instance, with the parameter *Tommy*, another Robot instance. A message arc labelled `greet: aRobot` is drawn to the *Quincy* icon. The parameter `aRobot` is the object *Tommy*, drawn immediately below the arc label. The parameter object is drawn to teach students that the actual object, not simply a copy, is passed into a method.

Many ideas for the AnimatedTracer were inspired by the Cunningham diagrams discussed in Chapter 3. We implemented a prototype version of the diagrams to investigate which features might be useful in Sherpa. From the prototype, we noted that the sequential information displayed during the dynamic creation of a diagram is useful.

In the AnimatedTracer, we display sequential information by animating all message sends one at a time.

Another feature which proved to be a good visual aid was drawing a message arc from sender to receiver. However, contextual information was missing because the sender and receiver were drawn as classes instead of instances. In the AnimatedTracer, we decided to display the instances involved in a message send to give students more visual information. Instead of a box with its name, each instance has a picture to make the tracer more fun to use. To give students more contextual information without cluttering up the screen, the state of each instance can be inspected at any time during a message send.

Students can execute the AnimatedTracer in one of two modes: step mode and animate mode. In step mode, execution suspends after the first message send is displayed. Students are by default in step mode and must control the trace. A minimal amount of control is available through four buttons: **begin**, **next level**, **animate**, and **go to end**. We decided to limit students' choices to avoid confusion for them. Our decision was based on the complexity of the tracers and debuggers discussed in Chapter 3. In particular, from a demonstration of TRACK given at the OOPSLA 90 Conference, we realized that a flexible tracer is a beneficial tool for experienced students but is too complex for beginners. Students learning how to program do not know how to set up a useful trace, namely: what data to display, where to set breakpoints or which messages to filter in or out. Students set the initial breakpoint to indicate where to start the trace but the trace continues in step mode after that. They do not control the data that is displayed or the messages that are filtered. However, at any step in the trace, students can see a more detailed trace of the current message selected by selecting **next level**. This button is based on the **hop** button in the Smalltalk Debugger. Students can also end a trace at

any point with the **go to end** button. This button is similar to the jump button in the Smalltalk Debugger.

Messages that have completed execution are erased to avoid cluttering the screen. Objects created during a message send are lost unless bound to non-local variables. Thus, only objects that persist remain on the screen. This could be used to teach students about variables and scope.

Students have the option of watching a trace without interrupts. Instead of step mode, students execute in animate mode. Message sends are displayed with a delay of approximately 2 seconds between each send. Although not implemented in Sherpa, a useful feature for the animate mode would be to be able to control the speed of the animation. There is no animation feature in the Smalltalk Debugger.

In both the step mode and the animate mode, the trace demonstrates to students the flow of message sends. Data information is not displayed to avoid clutter. We teach students that the behaviour of an instance when sent a message often depends on its internal state. The internal state of any object in the display can be inspected at any time during a trace. For example, Figure 27 shows the *Quincy* Robot before it moves to greet the *Tommy* Robot. Figure 28 shows the objects after the **goto:** message contained in the **greet:** method is sent.

Figure 27: Viewers before greet:

Figure 28: Viewers after greet:

The AnimatedTracer is based on the existing Smalltalk Debugger. When the Debugger is invoked by *self halt*, it contains information about the current context of the objects involved in a message send. By adding a *self trace* facility that invokes a Debugger process, this information is available to the tracer and can be used to calculate the pictures to display. For maximum code reuse, the AnimatedTracer is a subclass of Debugger, thus inheriting the behaviour needed in the tracer.

4.1.7 The Viewer

The Viewer is a read-only Inspector. An Inspector displays all of the instance variables of an object and their values. Using an Inspector, students can change the state of an object. Although useful for debugging and exploratory learning, this flexibility violates data encapsulation.

The Viewer has two purposes. The first purpose is to prevent students from violating encapsulation. In a Viewer, the state values are read-only. The second purpose is to hide the variables that students do not need to see. A Viewer displays only those variables visible in the current World. Figure 29 shows an Inspector and a Viewer open on the same Robot instance. The visible difference between these two tools is that the Inspector displays the value of a variable only when that variable is selected. The Viewer shows all of the values of the variables.

When working in a Galaxy that contains a Viewer, selecting "inspect it" from any menu opens a Viewer instead of an Inspector. When in a Viewer, if students choose to further inspect an object, another Viewer is opened as shown in Figure 30.

Viewing: Robot		Inspecting: Robot	
self	Robot('Quincy')	self	Robot('Quincy')
pen	a Pen	animator	
picture	a Form	name	
name	'Quincy'	pen	
location	101 @ 10	picture	
		savePicture	
		varList	

Figure 29: An Inspector and a Viewer open on the same Robot object.

Viewing: Robot		Viewing: Point	
self	Robot('Quincy')	self	101 @ 10
pen	a Pen	x	101
picture	a Form	y	10
name	'Quincy'		
location	101 @ 10		

Figure 30: Multiple Viewers

The Viewer is also available in both the GraphicWorld and the AnimatedTracer. In both of these tools, if an object's picture is displayed in the window, double clicking on the object will open a Viewer. Students use the Viewer just as they would an Inspector, to inspect the internal state of an object.

4.1.8 The ErrorWindow

The ErrorWindow is a simplified version of the Debugger. The Debugger is a complex tool in the Smalltalk environment and most beginning students do not learn how to use it effectively. They can however, use some of the information from the Debugger to help them debug their code.

When a non-syntactic error occurs, a walkback window opens as shown in Figure 31. This window has the error message in the label bar and a traceback of the message sends in a list pane below. If the error cannot be determined at this point, students can choose to "debug", opening the Debugger as shown in Figure 32. For most beginning students, the Debugger contains information that they do not understand so they simply close the window.

Figure 31: A Walkback window

Figure 32: The Debugger

Many typical errors made at the beginner level can be debugged by reading the error message from the label bar in the walkback window. We often tell our beginner students to read the label bar when an error occurs and then close the window. This is the basis for the ErrorWindow. When an error occurs, a window displaying only the error message opens as shown in Figure 33. The traceback is not shown and further debugging is not allowed.

Figure 33: The ErrorWindow

The ErrorWindow is only a temporary solution to the problem. It helps students in the initial learning stages. However, they must eventually use the full Debugger as their programs become more complex. We need a Debugger that is easier for students to use and one that they can learn gradually; we do not address this problem in this thesis.

4.2 The Instructor Sherpa Environment

We designed the instructor Sherpa environment to give instructors complete control over the student Sherpa environment. As instructors typically have their own style of teaching and their own order for teaching concepts, we needed to make Sherpa flexible enough to accommodate the different preferences of instructors. In Sherpa, instructors design, build, and organize several Worlds and Galaxies to correspond with the course material being taught. The instructor Sherpa contains the tools necessary to easily perform these tasks. To help instructors test the student environments that they build, we designed a simulation mode in which instructors can pretend they are the students. To encapsulate Galaxies, Worlds, and tools specifically designed for the instructor, we developed the concept of a Universe.

4.2.1 The Universe

Instructors develop Galaxies and Worlds from within the Universe. The entire Smalltalk class library and all of the Smalltalk tools are available for use. When instructors enter the Universe for the first time, it is empty. Instructors typically build a few Worlds and then some Galaxies to organize the Worlds. Each Galaxy is marked as available or unavailable to students. Although several Galaxies may exist in the Universe, students can only be in one Galaxy at a time, thus only one can be marked as available.

Once a Galaxy is created and its attributes are set, instructors can simulate what students would see when working in that Galaxy. It is a simple task to switch back and forth between the simulation and the Universe. We designed the simulation mode to encourage instructors to perform a user test of their student environments.

To save the Galaxies and Worlds, instructors simply save the Smalltalk image. To make a Galaxy part of the student environment, it must be filed out and subsequently filed into the students' image. Filing in a Galaxy automatically sets up the student Sherpa environment described in section 4.1.

Instructors enter the Universe by selecting "Enter Universe" from the system menu as shown in Figure 34. There are two main tools in the Universe: the Universe window and the WorldModifier. The Universe window is a summary of all of the Sherpa related objects in the Universe. The WorldModifier is a tool to manipulate Worlds.

4.2.2 The Universe Window

The Universe window helps instructors organize the Worlds and Galaxies in the Universe. Figure 34 shows a Universe window in which instructors see a list of all of the

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Galaxies, Worlds, WorldBrowsers and GraphicWorlds in the Universe. Help text exists for the Universe as shown in Figure 35. An example Universe containing several Galaxies and Worlds is shown in Figure 36. If a Galaxy is selected, its contents are revealed as in Figure 37; the Worlds, the available World views, and the Galaxy's attributes are automatically displayed.

Figure 34: A Universe window

Figure 35: A Universe window with its help window

Universe		In Galaxy	
all			
Galaxies	Worlds	Browser : Graphic	
Toys Drawing Literal Fighters	Collections Literals Machinery Pens Robot UserInterface UserInterfaceExt		

Figure 36: A Universe with several Galaxies and Worlds

Universe		In Galaxy	
Galaxies	Worlds	Browser : Graphic	
Toys	Collections		
Drawing	Literals		
Literals	Machinery	**	**
Fighters	Pens		
	Robot	**	**
	UserInterface		
	UserInterfaceExt		

Figure 37: Toys Galaxy selected

In the Universe window, instructors have many functions available:

1. create, delete, modify, simulate, or file out a Galaxy
2. create, delete, modify or file out a World
3. add a World to a Galaxy; delete a World from a Galaxy
4. set the view for a World: WorldBrowser or GraphicWorld or both
5. set the Galaxy available to students in the student Sherpa
6. set the Galaxy attributes: Viewer or Inspector; ErrorWindow or Debugger.

As we did not want students to use the instructors' Universe tools, we designed a procedure to extract the student Sherpa from the instructor Sherpa. Instructors simply file out their changes. The filed-out file contains all of the information needed to create the Galaxy exactly as it was when simulated in the instructor Sherpa. The procedure is based on the laboratory configuration used for the introductory programming course at Carleton University. Students work with their own Smalltalk image in which they save all of their term work. They access the Smalltalk source and execution files from a

network file server. To avoid having to get a new image each time the instructor changes the student Sherpa, students file the instructor's changes from the network into their image. With this procedure, students do not lose the work they have done, and instructors do not have to develop the entire curriculum and all of the student Sherpa environments before a course begins. Both parties benefit from the incremental nature of Sherpa.

4.2.3 The WorldModifier

The WorldModifier is a Universe tool for manipulating Worlds. An example WorldModifier is open on the RobotWorld in Figure 38; it displays the classes, methods and variables contained in the World. The WorldModifier is used to:

1. hide or show classes in a World
2. hide or show class and instance variables in a World
3. hide or show class and instance methods in a World
4. hide or show method source code
5. extend the base class list
6. create new Worlds
7. modify existing Worlds
8. simulate World views, both WorldBrowsers and GraphicWorlds.

Figure 38: A WorldModifier

In our original design of the WorldModifier, all of the classes in the Smalltalk library were displayed. However, the size of the library can sometimes be a daunting experience for instructors, especially those unfamiliar with Smalltalk. By consulting with colleagues, we identified a base set of 42 classes that we believe beginning students need to know (See Appendix I). The instructors' work is simplified because they have to work with only 42 classes instead of over 100. The base class list is not fixed; instructors can extend it as needed. This feature is necessary for instructor defined classes such as our Robot class. Although it may prove to be a useful feature, we did not identify a base set of methods or variables.

Figure 39 shows an example RobotWorld; the complementary WorldModifier is shown in Figure 38. The classes and variables in the World are marked by an asterisk(*). Methods in the World are marked with either an asterisk or a percent sign(%). A percent sign indicates that the source code for that method is hidden; only the method name and comment are visible to students.

```
Robot World
class Robot

instance methods
chase:
greet:
house
jump
square
```

Figure 39: RobotWorld containing the basic Robot behaviour

The WorldModifier was designed to anticipate the needs of instructors building a set of Worlds. Through informal user testing, we identified features that instructors deemed important and implemented them in the WorldModifier. As shown in Figure 38, the menus give instructors many options.

The WorldModifier does not currently handle the display of inherited methods and variables. If an instructor wants to add an inherited method or variable to a World without showing the class of that method or variable, it must be done manually in the Smalltalk code of Sherpa. If this feature were added to the WorldModifier, a natural extension of Sherpa would be to have the WorldBrowser show inherited methods and variables. Students could then be given the option of hiding and showing them.

4.3 Summary

The student Sherpa environment was designed to encourage beginning students learning Smalltalk. To ease the struggle of learning a first programming language, we believe that the environment must be simple and fun. The concept of a World containing only a subset of the class library helps to simplify the environment. The GraphicWorld and the AnimatedTracer were designed to teach students the basic programming concepts while they have fun with graphics.

The student Sherpa is not fool-proof. Students who really want to view more than the instructors wish can bypass the system. The intent of the student Sherpa is not to prevent students from exploring and learning more about Smalltalk. It is designed so that the class library and the environment can be introduced gradually so as not to confuse beginning students.

The instructor Sherpa was developed to help instructors design the student Sherpa to correspond to their presentation material. With the diverse styles of instructors, a simple working environment with flexible tools was necessary. Our system is unique in this respect.

5 Teaching Curriculum

To date, no research has been done to determine a proper order for teaching students computer science concepts when an object-oriented pedagogic language is used²³. In teaching Smalltalk as the first programming language at Carleton University, we have experimented with various teaching orders [Skublics and White 91], [LaLonde and Pugh 90b].

In this chapter, we propose a teaching curriculum for an introduction to programming and problem solving course using Smalltalk. The Sherpa system is incorporated into the curriculum as a teaching aid.

Concepts are introduced to students in several stages in the classroom and in Sherpa. Although the stages are presented as distinct units, it is difficult to teach them as such because the concepts overlap. The stages serve as guidelines as to where each concept is formally introduced in a curriculum. Figure 40 shows how the specific stages of our curriculum and the Sherpa tools follow a spiral curriculum²⁴. Table 1 summarizes the stages, the time allocated for each stage, and the Sherpa tools introduced to students.

The stages described here are a proposal to teach computer science concepts in an introductory programming course using Smalltalk. The order has been used to teach object-oriented programming and Smalltalk to a group of seven students, four of whom had no previous programming experience. The complete Sherpa environment was not used. Some examples of GraphicWorlds such as the RobotWorld, and the AnimatedTracer were used in class to demonstrate the basic object-oriented concepts; the response was positive.

²³ Issues of ACM SIGCSE Bulletin contain proposals for non object-oriented teaching curricula.

²⁴ Reference to [Bruner, Goodnow, Austin 56] in [Rosson and Carroll 90], p78.

Figure 40: Spiral curriculum using Sherpa

Time (%)	Stage	Tools used
15	<i>Problem solving with objects</i>	ErrorWindow, Viewer, AnimatedTracer, GraphicWorld
15	<i>Computation with message sends and Encapsulation</i>	"
10	<i>Class vs Instance</i>	"
5	<i>Bindings</i>	"
5	<i>Literals</i>	" and WorldBrowser
10	<i>Extending classes with new methods</i>	" and full Debugger or ErrorWindow
10	<i>Inheritance</i>	"
10	<i>Introducing new classes into the library</i>	" and full Debugger, Inspector
10	<i>Collection</i>	"
10	<i>User interface component</i>	"

Table 1 : Summary of Curriculum Stages

5.1 The Stages

In the **Problem solving stage**, students learn how to solve problems using objects and object-oriented concepts. They learn how to create new objects and inspect them. They learn about state and behaviour with little exposure to the differences between classes and instances. This is an important stage because students are exposed to the environment for the first time and must get accustomed to it. Without formal introduction, students also learn about assignment statements, global variables, instantiation, encapsulation and computation using messages.

In the student Sherpa, no Browsers are available and no method code is visible. Students do not need to know about classes and methods in the library yet. In all GraphicWorlds, the Player class is included in the class hierarchy to slowly expose students to inheritance. The student Sherpa contains the ObjectGalaxy which includes:

1. Robot GraphicWorld with Robots moving and changing
2. Airplane GraphicWorld with planes taking off and landing
3. PenWorld WorldBrowser to use Pens for drawing on the screen
4. Any objects that can be animated, such as Animals, in a GraphicWorld

In the **Computation stage**, students learn how to use messages for computation. We use the GraphicWorlds from the Problem solving stage to reinforce the notion that when message sends occur, objects react. Students use the AnimatedTracer to learn about senders and receivers of message sends. In the student Sherpa, no Browsers are available and no method code is visible because students are not adding behaviour yet. In this stage, students are exposed to literal objects and precedence rules to help them learn how to *read* code.

The distinction between class and instance is a difficult concept for students to grasp. Students have already been exposed to classes and instances by working in GraphicWorlds but in the **Class vs Instance stage**, we introduce the concepts formally. More time is spent teaching this in the classroom than working in the environment. Students learn that classes can have multiple instances and that these instances have the same behaviour but may have different states. The Viewer is used to compare objects.

In the **Bindings stage**, students receive a formal introduction to bindings, variables and scope. They have already learned about global variables and assignment statements in the GraphicWorlds, and about instance variables by inspecting objects. The emphasis is placed on two concepts: variables are bound to real objects and not simply copies; and state variables are not accessed directly but rather with accessor messages. Students are still working in the ObjectGalaxy although instructors can introduce new Worlds to add variety.

In the **Literals stage**, students learn about literal objects: Numbers, Strings, Arrays, Characters, and Symbols. Polymorphism is taught at this stage because literals have many good examples of this concept. Students are exposed to the WorldBrowser to see how classes and message protocol can be organized. However, the code for the message protocol is not yet visible in the browser. The AnimatedTracer is used to learn more

about message protocol. Students work in a LiteralGalaxy, as in Figure 21, which contains WorldBrowsers for:

1. NumberWorld (integers, fractions, floats)
2. StringWorld
3. ArrayWorld
4. CharacterWorld
5. SymbolWorld

In the **Extending Classes** stage, students learn how to add simple behaviour to existing classes. A Pen World with simple behaviour is provided in a WorldBrowser. Students learn to add new instance methods and learn to immediately test them. The Robot GraphicWorld is provided for students who wish to add new behaviour to Robots and *see* the results during a test. Students are given freedom in this stage to experiment. They have learned enough to develop a video game using Robots in the GraphicWorld.

Students have already been exposed to inheritance before the formal **Inheritance** stage; they learned about Robots inheriting behaviour from Player in the Robot GraphicWorld. In this stage, a Player World containing Robots, Airplanes, and Animals is created to show that all of these classes inherit from the Player class; both the GraphicWorld and WorldBrowsers are used. The LiteralGalaxy is modified to include a Magnitude World showing the inheritance from the Magnitude class.

In the **New Classes** stage, students learn how to add new classes to the existing library. A GraphicWorld view of Player is used for students to add new subclasses of Player. Students add a class and its protocol in the Player WorldBrowser and then test their code in the Player GraphicWorld. To move students away from using the GraphicWorld, they start to use only the WorldBrowsers to develop their classes. As adding new classes is more difficult, the full Debugger is introduced at this stage to help

students debug their code. We also introduce class methods here because we teach students that they can override the new class method to initialize an object.

In the **Collection stage**, we formally introduce Collections. The students have already used simple examples of Collections up to this point; various Collection Worlds are presented with sufficient message protocol to use them. An OrderedCollection World is shown in Figure 41. Method code is not visible because it is not necessary for the students to see the implementation of most of the Collection protocol.

Figure 41: OrderedCollection World

In a Collection Galaxy, we include Worlds for Arrays, Strings, OrderedCollections, SortedCollections, Dictionaries, and Sets. We teach the differences between the different kinds of Collections. As little as possible of the Collection hierarchy is shown, only enough to teach students the protocol that is inherited.

In the **User Interface component stage**, students learn how to use prompters and menus to make their assignments interactive. A UI World including prompters, menus and possibly text editor windows is contained in the students' Galaxy, as shown in Figure 42. By the time students have reached this stage, the Galaxy contains most of the classes that students will need to see; A Real World exists that contains all of these classes.


```

class Menu

instance methods
class methods
labels:lines.selectors:
.....

class Prompter

instance methods
class methods
prompt:default:
prompt:defaultExpression:
.....

class TextEditor

instance methods
class methods

```

Figure 42: UI World

5.2 Summary

The proposed curriculum describes an order for teaching object-oriented concepts following a spiral model. The proposed stages demonstrate how Sherpa can be used exclusively by instructors to set up various Worlds and Galaxies that aid in teaching the curriculum. Sherpa plays a major role for student, in the first half of the course. The GraphicWorlds and the AnimatedTracer help students gain a solid understanding of concepts without having to contend with the entire Smalltalk environment. The students learn concepts visually instead of textually. The Viewer implicitly teaches them to obey encapsulation. The ErrorWindow helps them find simple errors: they do not need the detailed information of the full Debugger.

In the second half of the course, when students have a better background, they are introduced to the WorldBrowser, the Inspector and the full Debugger. In these stages of the curriculum, Sherpa is mainly used as a way of limiting the class library visible to the students.

6 Conclusion

6.1 Conclusion

This thesis is motivated by the use of Smalltalk as the pedagogic language within the B.C.S. program at Carleton University. Our objectives are to make Smalltalk easier for students to learn, and easier for more than one instructor to teach. To meet these objectives, we designed and implemented Sherpa: an enhanced development environment for both students and instructors. Our work is aimed toward students with little or no previous programming experience.

We have found that many programmers who use an object-oriented language such as Smalltalk taught themselves using reference manuals, on-line tutorials and the experience of colleagues. Unfortunately, only some have been taught by an instructor in a lecture-style format. We believe that an experienced instructor is an asset to students learning object-oriented programming, and in particular, to beginning students. For this reason, Sherpa was not designed to stand alone but rather to assist instructors in teaching Smalltalk.

Sherpa was aimed to solve the following difficulties for students:

1. Understanding the difference between classes and instances
2. Coping with a large class library from which they need only learn a small subset of classes and methods
3. Following a chain of message sends to understand distributed code
4. Understanding and respecting encapsulation
5. Debugging problems without being overwhelmed with information

We did not tackle all of the problems identified in Chapter 2, in particular:

1. Implementing an extensive on-line help system
2. Simplifying the explanation of object instantiation (the new message)
3. Simplifying the search for classes and methods within the library

We developed Sherpa following two main themes:

1. Using pictures to explain and demonstrate concepts
2. Reducing the amount of information presented to the beginning student

Although Sherpa incorporates several ideas from other tools and systems, Sherpa is unique in the following ways:

1. In the Cunningham diagrams, arcs are sent to classes to represent message sends; we send message arcs to graphically represented instances
2. The research done in the IBM group assumes that students have a background in programming; we assume no previous background
3. The teaching curriculum proposed in the IBM group teaches Smalltalk quickly; our curriculum proposes a slower, more reasonable pace
4. Existing systems and tools do not provide a flexible environment to handle multiple instructors with different styles; Sherpa has the instructor environment

There are two working environments in Sherpa: one for the student and one for the instructor. The student Sherpa system is a Smalltalk development environment with the following enhancements:

1. The **World**, an object that encapsulates specific library classes, methods and variables to help students cope with the large class library.
2. The **GraphicWorld**, a view of a World using pictures and animation to teach the basic concepts of problem solving with objects.
3. The **WorldBrowser**, a view of a World consistent with the Class Hierarchy Browser to help students browse a World.
4. The **Viewer**, a read-only Inspector to assist the teaching of encapsulation.
5. The **ErrorWindow**, a simplified Debugger containing sufficient information to help students debug a program.
6. The **AnimatedTracer**, a tool that animates objects and message sends to help students understand the distributed nature of an object-oriented program.

The instructor system is the student Sherpa augmented with flexible tools to help instructors build, organize, and simulate the student environments. Using Worlds and Galaxies, instructors control the classes, methods, and variables that students can access.

Instructors incrementally expand the students' view of the system to correspond to their course presentation material. Just as students gradually learn computer science concepts and object-oriented concepts in the classroom, Sherpa allows gradual learning in the programming environment.

We believe that using the Sherpa system makes it easier for students to learn how to program using Smalltalk, and easier for instructors to follow their own style when teaching students how to program.

Informal demonstrations of Sherpa have been given to various students and instructors. The feedback has been positive. Various components of the Sherpa system will be integrated into the Fall 1991 Introduction to Programming course in the School of Computer Science at Carleton University.

An early prototype of the GraphicWorld component of Sherpa was given to high school students as an experiment. The results were encouraging. The students had fun while learning how to add new classes and methods. After only a few hours, some of the students were already designing video games.

6.2 The Sherpa Implementation

The Sherpa system was implemented in Smalltalk/V286. The design and implementation took approximately 8 months and includes 33 defined classes and 26 extended classes totalling 250 methods. Although the system could have been implemented in any language, the task was simplified because of Smalltalk's open environment and its extensive class library. Access to the system level code makes it easier to extend the existing library to handle the features of Sherpa.

There were several problems encountered in the development of the Sherpa system. The design of the AnimatedTracer for example, required access to the source code for the Compiler. To determine the correct pictures to draw, we wanted to be able to parse a message send into the sender, receiver, message and arguments before the message was executed. Unfortunately, the V286 Smalltalk library does not make this code available. We had several approaches to solving this problem. First we attempted to reverse engineer the Compiler by using the Debugger to step through code. Then we looked at the source code for the Compiler in the Smalltalk-80 library, which is available, and tried to modify it for the V286 library. To simply prove the concept of the AnimatedTracer, we finally decided to implement the AnimatedTracer using hooks into the Debugger. The hooks interrupt a debug process at points where the information needed to animate a message send are available.

Another problem encountered was with the animated objects in the GraphicWorld. The objects use the Animator, an instance of the Animation class provided in the V286 class library. Unfortunately, the Animation class inherits operations from Pen and BitBlt when it should simply use them. This made the class difficult to use and modify to meet our needs. Its limited capabilities were discovered quite late in the development of the GraphicWorld. Although we attempted to redesign the Animation class, we believed that subsequently rewriting the GraphicWorld classes was beyond the objectives of this thesis. By extending the Animation behaviour and modifying the Player class to maintain information similar to the Animation class, the GraphicWorld behaves as we intended in the design.

The current prototype of Sherpa was implemented to analyze the concepts proposed in this thesis. Some of the features proposed were not implemented because they would not add further proof of the concepts. Although they could easily be implemented as

extensions of the existing features, the following items are not included in the current Sherpa prototype:

1. the animation of return values in the AnimatedTracer
2. the view of Galaxy attributes in a browser
3. the ability to execute an animated trace using traceIt from a text menu
4. the ability to optionally display inherited methods and variables in a Browser
5. the ability to hide class variables.

6.3 Future Research Topics

In Chapter 5, we proposed a teaching curriculum for an introduction to programming and problem solving course using Smalltalk. A major extension of this thesis would be to determine a proper order for teaching computer science concepts when an object-oriented pedagogic language is used. One challenge is to separate computer science concepts, language concepts, and development environment concepts; the latter two concepts are typically taught intertwined. Once one or more teaching orders are developed, the challenge lies in setting up an experiment to gather data. The experiment needs many people with similar previous programming backgrounds and people with varying levels of expertise; finding inexperienced people is a challenge with Computer Science being introduced in high school. The experiment also needs to be conducted over several years, as [Campbell 90] concurs, to gather significant data.

As noted earlier, we did not implement an extensive on-line help system. Some way of documenting each class and each method could be designed as on-line help for students. A hypermedia type of environment may be useful. Students would be able to get help from anywhere in the system simply by double clicking on text such as method, variable and class names. Due to its complexity, teaching students a hypermedia environment would also require an incremental introduction such as a spiral curriculum.

Documentation could also help students with code reuse. Keywords extracted from the documentation could be linked to the respective classes or applications and serve as a keyword library; there must also be a mechanism for the user to link keywords manually. Users could then give a description of the library components they are looking for. Keywords would be extracted from the description and matched up with the keyword library resulting in a list of possible reusable components.

Unfortunately, documentation and keywords are not always accurate. Some mechanism is needed to automatically generate documentation, perhaps a graphic representation of how a class or application functions. This representation would always be up to date as its components would be linked directly into the code. If either the graphics or the code were changed, it would be reflected in the other. The problem then becomes how to search and describe text and graphic patterns.

The difficulties students have debugging their programs was only partially investigated in this thesis. The ErrorWindow is only the start of the solution. The existing Smalltalk Debugger should be redesigned to hide information from users unless needed. An investigation of the typical use of the Debugger during different stages of learning may identify the necessary data and tools at each stage. The model of the current Debugger has most of the information needed to debug a program. A different view of this information needs to be designed. Implementing a redesigned Debugger is made more difficult by the lack of access to the Compiler source code in Smalltalk/V286. A combination of reverse engineering the compiler and looking at the source code in Smalltalk-80 may help the task but would require extensive work.

The AnimatedTracer is an example of a different view of the debugger model but it is simply a prototype. A feature that could be added to the AnimatedTracer would give users the freedom to filter the traces. The user does not control the messages and objects that are traced or the depth of the trace in the current AnimatedTracer. Another

enhancement could be to show method lookup and inheritance. This feature may help students better understand polymorphism and inheritance. The current AnimatedTracer displays the message being sent but does not indicate the class that implemented the message. Access to the source code for the Compiler would simplify the task of enhancing the AnimatedTracer.

This thesis addresses problems that beginning computer science students have with programming in the introductory programming course. In the latter stages of the course, students are typically working with two or more classes in an assignment. Students have difficulty understanding how classes collaborate. The teaching issues to address here involve design more than programming techniques. Courses taken subsequent to the introductory programming course address design issues. A major extension of this thesis would be to investigate problems with designing, particularly object-oriented programs, and introduce new tools and techniques into these latter courses, a tool such as an extension of CRC [Beck and Cunningham 89]. Typical problems that students have in the latter courses include abstract classes, factoring class hierarchies, and deciding what should be an object. The idea of using graphics and gradual learning as discussed in this thesis may help with learning and teaching design concepts as well.

7 References

Baecker 81

Ronald M. Baecker. *Sorting out sorting*(film), Dynamic Graphics Project, Computer Science Research Group, University of Toronto, Toronto, Canada, 1981.

Baecker and Buxton 87

Ronald M. Baecker and William A.S. Buxton. "Programming Techniques and Tools". *Readings in Human-Computer Interaction: A Multidisciplinary Approach*. California: Morgan Kaufmann Publishers, Inc. 1987. pp 555-560.

Beck and Cunningham 89

Kent Beck and Ward Cunningham. "A Laboratory For Teaching Object-Oriented Thinking". *OOPSLA '89 Conference Proceedings* 24(10), October 1989. pp 1-6.

Boecker and Herczeg 90

Heinz-Dieter Boecker and Jurgen Herczeg. "TRACK - A Trace Construction Kit". *SIGCHI '90 Conference Proceedings*, Seattle, Washington. April 1990. pp 415-422.

Booch 90

Grady Booch. *Object Oriented Design with Applications*. Redwood City, California: Benjamin/Cummings Publishing Company, Inc. 1990.

Borning and O'Shea 87

Alan Borning and Tim O'Shea. "Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk -80 Language". *European Conference on Object-Oriented Programming '87, Special Issue of BIGRE, #54*, Paris, France. June 1987. pp 3-12.

Brown and Sedgewick 84

Marc H. Brown and Robert Sedgewick. "A System for Algorithm Animation". *Computer Graphics: SIGGRAPH '84 Conference Proceedings*. Minneapolis, Minnesota. 18(3). July 1984. pp 177-186.

Brown, Carling, Herot, Kramlich, and Souza 85

Gretchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich, and Paul Souza. "Program Visualization: Graphical Support for Software Development". *I.E.E.E. Computer* 18(8), August 1985. pp 27-35.

Bruner, Goodnow, Austin 56

J.S. Bruner, J.J. Goodnow, and G.A. Austin. *A study of thinking*. New York: Wiley. 1956.

Campbell 90

Robert L. Campbell. "Developmental Scenario Analysis of Smalltalk Programming". *SIGCHI '90 Conference Proceedings*, Seattle, Washington. April 1990. pp 269-276.

Campbell, Brown and DiBello 90

Robert L. Campbell, Norman R. Brown and Lia A. DiBello. "The Programmer's Burden: Developing Expertise in Programming". *IBM Research Report RC15382*, 1990. IBM T.J. Watson Research Center, Yorktown Heights, NY. 8 pages.

Carroll, Singer, Bellamy and Alpert 90

John M. Carroll, Janice A. Singer, Rachel K.E. Bellamy, and Sherman R. Alpert. "A View Matcher for Learning Smalltalk". *SIGCHI '90 Conference Proceedings*, Seattle, Washington. April 1990. pp 431-437.

Carroll 90

John M. Carroll. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. Cambridge, MA: MIT Press. 1990.

Cunningham and Beck 86

Ward Cunningham and Kent Beck. "A Diagram for Object-Oriented Programs". *OOPSLA '86 Conference Proceedings* 21(11), November 1986. pp 361-367.

Ezell 90

C. L. Ezell. "Creating Pedagogical Programming Environments". *SIGCSE Bulletin* 22(2). June 1990. pp 42-46.

Fenton and Beck 89

Jay Fenton and Kent Beck. "Playground: An Object-Oriented Simulation System with Agent Rules for Children of All Ages". *OOPSLA '89 Conference Proceedings* 24(10), October 1989. pp 123-137.

Gold and Rosson 90

Eric Gold and Mary Beth Rosson. "Object-Centered Environments for Smalltalk". *IBM Research Report RC15548*, 1990. IBM T.J. Watson Research Center, Yorktown Heights, NY. 21 pages.

Goldberg and Ross 81

Adele Goldberg and J. Ross. "Is the Smalltalk-80 System for Children?" *BYTE*, August 1981. pp 349-369.

Isoda, Shimomura and Ono 87

S. Isoda, T. Shimomura and Y. Ono. "VIPS: A Visual Debugger". *IEEE Software* 4 (5), May 1987. pp 8-19.

Kleyn and Gingrich 88

Michael F. Kleyn and Paul C. Gingrich. "GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views". *OOPSLA '88 Conference Proceedings* 23(11), November 1988. pp 191-205.

Knudsen and Madsen 88

J. L. Knudsen and Ole L. Madsen. "Teaching object-oriented programming is more than teaching object-oriented programming languages". In S. Gjessing and K. Nygaard, editors. *European Conference on Object-Oriented Programming*. pp 21-40, Oslo, Norway: Springer-Verlag. 1988.

LaLonde, Pugh and Thomas

Wilf R. LaLonde, John Pugh, and Dave Thomas. "The Smalltalk Experience". Technical Report, Carleton University.

LaLonde and Pugh 90a

Wilf R. LaLonde and John Pugh. *Inside Smalltalk, Volume I*. New Jersey: Prentice Hall. 1990.

LaLonde and Pugh 90b

Wilf R. LaLonde and John Pugh. "Smalltalk as the first programming language: the Carleton experience". *Journal of Object-Oriented Programming* 3(4). November/December 1990. pp 60-65.

London and Duisberg 85

Ralph L. London and Robert A. Duisberg. "Animating Programs using Smalltalk". *IEEE Computer* 18(8), August 1985. pp 61-71.

Madsen and Moller-Pederson 88

Ole L. Madsen and Birger Moller-Pederson. "What object-oriented programming may be - and what it does not have to be". In S. Gjessing and K. Nygaard, editors. *European Conference on Object-Oriented Programming*. Springer-Verlag, Oslo, Norway, 1988. pp 1-20.

Moher 88

Thomas G. Moher. "PROVIDE: A Process Visualization and Debugging Environment". *IEEE Transactions on Software Engineering* 14(6). June 1988. pp 849-857.

Myers 83

Brad Myers. "Incense: a system for displaying data structures". *Computer Graphics* 17(3). July 1983. pp 115-125.

Myers 88

Brad Myers. "The State of the Art in Visual Programming and Program Visualization". *Technical Report CMU-CS-88-114*, Carnegie Mellon University. 25 pages. February 1988.

Nielsen and Richards 89

Jakob Nielsen and John T. Richards. "The Experience of Learning and Using Smalltalk". *IEEE Software* 6(5) May 1989. pp 73-77.

O'Shea, Beck, Halbert, Schmucker 86

Tim O'Shea, Kent Beck, Dan Halbert and Kurt Schmucker. "Panel: The Learnability of Object-Oriented Programming Systems". *OOPSLA '86 Conference Proceedings* 21(11), November 1986. pp 502-504.

Papert 80

Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. New York: Basic Books, Inc. 1980.

Pattis 81

Richard E. Pattis. *KAREL the ROBOT: A Gentle Introduction to the Art of Programming*. New York: Wiley. 1981.

Pugh 89

John Pugh. "Teaching OOP and Smalltalk: A View from the Trenches". *HOOPLA* 2(2), 1989. pp 3-5.

Pugh 90

John Pugh. "Carleton University uses Smalltalk/V286 as first programming language". In B. Noparstak (editor), *SCOOP III*(3), 1990.

Raeder 85

Georg Raeder. "A Survey of Current Graphical Programming Techniques". *I.E.E.E. Computer* 18(8), August 1985. pp 11-25.

Rosson and Carroll 90

Mary Beth Rosson and John M. Carroll. "Climbing the Smalltalk Mountain". *SIGCHI Bulletin* 21(3). January 1990. pp 76-79.

Rosson, Carroll and Bellamy 90

Mary Beth Rosson, John M. Carroll, and Rachel K.E. Bellamy. "Smalltalk Scaffolding: A Case Study of Minimalist Instruction". *SIGCHI '90 Conference Proceedings*, Seattle, Washington. April 1990. pp 423-429.

Shafer 88

Dan Shafer, David A. Wilson, Jeff McKenna, John Pugh, and Adele Goldberg. "Panel: Teaching OOP". *OOPSLA '88 Conference Proceedings* 23(11), November 1988. pp 386-387.

Shan 90

Yen-Ping Shan. "MoDE: A UIMS for Smalltalk". *OOPSLA/ECOOP '90 Conference Proceedings* 25(10), October 1990. pp 258-268.

Singley and Carroll 90

Mark K. Singley and John M. Carroll. "Minimalist Planning Tools in an Instructional System for Smalltalk". *IBM Research Report RC15860*, 1990. IBM T.J. Watson Research Center, Yorktown Heights, NY. 8 pages.

Skublics and White 91

Suzanne C. Skublics and Paul J. White. "Teaching Smalltalk as a First Programming Language". *SIGCSE Conference Proceedings*, San Antonio, Texas. March 1991. pp 231-234.

Smith 86

Randall B. Smith. "The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations". *Proceedings of the 1986 I.E.E.E. Computer Society Workshop on Visual Languages*, Dallas, Texas. June 1986. pp 99-106.

Taenzer, Ganti and Podar 89

David Taenzer, Murthy Ganti, and Sunil Podar. "Object-Oriented Software Reuse: The Yoyo Problem". *Journal of Object-Oriented Programming* 2(4). November/December 1989. pp 25-36.

Urlocker 89

Zack Urlocker. "Teaching Object-Oriented Programming". *Journal of Object-Oriented Programming* 2(2). July/August 1989. pp 45-47.

White 90

Paul J. White. Private communication. Carleton University, 1990.

Wilson 90

David A. Wilson. "Class Diagrams: A Tool For Design, Documentation, and Teaching". *Journal of Object-Oriented Programming* 2(5). January/February 1990. pp 38-44.

Wirfs-Brock, Wilkerson and Wiener 90

Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ. 1990.

Appendix I: Sherpa Classes

This appendix includes the class hierarchy for the Sherpa system.

Legend

Class	Class defined by Sherpa
<i>Class</i>	Class extended by Sherpa
Class	Present to show inheritance

Object

AnimatedObject
 AnimObjWithPict
AnimatedTracer
Application
Behavior
 Class
BitBlit
 CharacterScanner
 Pen
 Animation
 EnvAnimation
BitEditor
 LargeBitEditor
ClassHierarchyBrowser
 ApplicationBrowser
 WorldBrowser
Collection
 IndexedCollection
 FixedSizeCollection
 Array
 CompiledMethod
 String
 OrderedCollection
 HierarchyList
 Process
Dispatcher
 ScreenDispatcher
 ScrollDispatcher
 GraphDispatcher
 ListSelector
 TextEditor
 TopDispatcher
 AnswerTopDispatcher
 ModalTopDispatcher
DispatchManager
DisplayObject
 DisplayMedium
 Form

File

Galaxy
GraphicWorld
HierarchyListElement
InputEvent
Inspector
 Debugger
 AnimatedDebugger
 DictionaryInspector
 Viewer
Magnitude
Menu
 HierarchyMenu
 Notifier
 PagedMenu
Message
 MessageSend
ModalWindow
 ClassDefinition
 HelpWindow
Pane
 SubPane
 GraphPane
 ListPane
 ConstrainedListPane
 KeepSelectListPane
 LabelPane
 MultiSelectPane
 TextPane
 TopPane
PictureLibrary
Player
 Machine
 Robot
Point
Rectangle
Universe
World
WorldBuilder

Appendix II: Base Classes

This appendix includes the list of 42 base classes available in the WorldModifier tool contained in the Instructor Sherpa Environment.

- Object
 - BitBlt
 - Pen
 - Boolean
 - False
 - True
 - Collection
 - Bag
 - IndexedCollection
 - FixedSizeCollection
 - Array
 - Interval
 - String
 - Symbol
 - OrderedCollection
 - Set
 - Dictionary
- Context
 - HomeContext
- DisplayObject
 - DisplayMedium
 - Form
 - DisplayScreen
- Magnitude
 - Character
 - Date
 - Number
 - Float
 - Fraction
 - Integer
 - LargeNegativeInteger
 - LargePositiveInteger
 - SmallInteger
 - Time
- Menu
- Point
- Prompter
- Rectangle
- Stream
 - ReadStream
 - WriteStream
- UndefinedObject

END

05·06·92

FIN