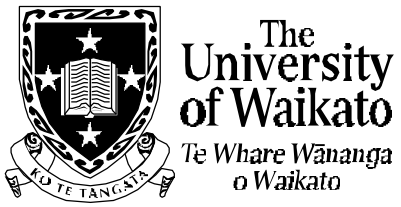
Department of Computer Science



Hamilton, New Zealand

Automating iterative tasks with programming by demonstration

Gordon W. Paynter

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at The University of Waikato.

February 26, 2000

Abstract

Programming by demonstration is an end-user programming technique that allows people to create programs by showing the computer examples of what they want to do. Users do not need specialised programming skills. Instead, they instruct the computer by demonstrating examples, much as they might show another person how to do the task. Programming by demonstration empowers users to create programs that perform tedious and time-consuming computer chores. However, it is not in widespread use, and is instead confined to research applications that end users never see. This makes it difficult to evaluate programming by demonstration tools and techniques.

This thesis claims that domain-independent programming by demonstration can be made available in existing applications and used to automate iterative tasks by end users. It is supported by Familiar, a domain-independent, AppleScript-based programming-by-demonstration tool embodying standard machine learning algorithms. Familiar is designed for end users, so works in the existing applications that they regularly use.

The assertion that programming by demonstration can be made available in existing applications is validated by identifying the relevant platform requirements and a range of platforms that meet them. A detailed scrutiny of AppleScript highlights problems with the architecture and with many implementations, and yields a set of guidelines for designing applications that support programming-by-demonstration systems and other agents.

An evaluation shows that end users are capable of using programming by demonstration to automate iterative tasks. However, the subjects tended to prefer other tools, choosing Familiar only when the alternatives were unsuitable or unavailable. Familiar's inferencing is evaluated on an extensive set of examples, highlighting the tasks it can perform and the functionality it requires.

Acknowledgments

I sometimes wonder what I would be doing now if Ian Witten had not talked me into writing a thesis. (I imagine I would have cashed in my stock options and retired by now.) I also wonder what might have become of me had I attempted a thesis under a supervisor less generous with his time and experience than Ian, without whose encouragement and example I surely would never have finished. Thanks Ian.

My two other supervisors, Geoff Holmes and Mark Apperley, have been very supportive, and patient. I'd like to thank them, and those others who have read this thesis and offered advice: Sally Jo Cunningham, Steve Jones, Simon Christiansen, Stuart Yeates, and Grant Floyd; and also my mother (who made one helpful suggestion before falling asleep at Section 1.3). I'm grateful to the many people who have commented on, and contributed to, my research over the last four years, including Allen Cypher, Eibe Frank, Craig Nevill-Manning, Carl Gutwin, Henry Lieberman, the staff of my department, and numerous others. I have borrowed code and ideas from any number of people, including Tom Bonura, Roman Zeiliger, Marty Geier (and his colleagues at Mitre), and the Weka programmers. I am sure I have omitted numerous others; for this I apologise.

I would surely not have lasted four years without the diversions afforded me by my flatmates: Simon, Sarah, Dale, Teri, Donna, Cat, Nick, Jan, Elliot, and others too numerous to mention. Our department is blessed with a fantastic soccer team, comprising many of our students and staff. Thanks Alvin (who started it all), Annette, Aziz, Carl, Chithiran, Conrad, David B., David M., Dong, Eibe, Geoff, Jamie, Len, Max, Kim, Richard, Rob, Steve, Stuart, Talia, Tony, Yong…

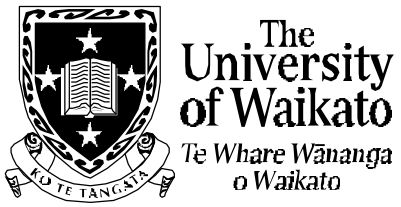
Finally, I'd like to thank my parents, my brother Craig, my sister Sharon, and the rest of my family, particularly the Gordons, for their support and encouragement. I dedicate this thesis, and the years of work it represents, to my uncle Doug Gordon, without whose support it could never have been attempted, let alone finished.

Table of contents

List of figures

List of tables

Department of Computer Science



Hamilton, New Zealand

Automating iterative tasks with programming by demonstration

Gordon W. Paynter

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at The University of Waikato.

February 26, 2000

Abstract

Programming by demonstration is an end-user programming technique that allows people to create programs by showing the computer examples of what they want to do. Users do not need specialised programming skills. Instead, they instruct the computer by demonstrating examples, much as they might show another person how to do the task. Programming by demonstration empowers users to create programs that perform tedious and time-consuming computer chores. However, it is not in widespread use, and is instead confined to research applications that end users never see. This makes it difficult to evaluate programming by demonstration tools and techniques.

This thesis claims that domain-independent programming by demonstration can be made available in existing applications and used to automate iterative tasks by end users. It is supported by Familiar, a domain-independent, AppleScript-based programming-by-demonstration tool embodying standard machine learning algorithms. Familiar is designed for end users, so works in the existing applications that they regularly use.

The assertion that programming by demonstration can be made available in existing applications is validated by identifying the relevant platform requirements and a range of platforms that meet them. A detailed scrutiny of AppleScript highlights problems with the architecture and with many implementations, and yields a set of guidelines for designing applications that support programming-by-demonstration systems and other agents.

An evaluation shows that end users are capable of using programming by demonstration to automate iterative tasks. However, the subjects tended to prefer other tools, choosing Familiar only when the alternatives were unsuitable or unavailable. Familiar's inferencing is evaluated on an extensive set of examples, highlighting the tasks it can perform and the functionality it requires.

Acknowledgments

I sometimes wonder what I would be doing now if Ian Witten had not talked me into writing a thesis. (I imagine I would have cashed in my stock options and retired by now.) I also wonder what might have become of me had I attempted a thesis under a supervisor less generous with his time and experience than Ian, without whose encouragement and example I surely would never have finished. Thanks Ian.

My two other supervisors, Geoff Holmes and Mark Apperley, have been very supportive, and patient. I'd like to thank them, and those others who have read this thesis and offered advice: Sally Jo Cunningham, Steve Jones, Simon Christiansen, Stuart Yeates, and Grant Floyd; and also my mother (who made one helpful suggestion before falling asleep at Section 1.3). I'm grateful to the many people who have commented on, and contributed to, my research over the last four years, including Allen Cypher, Eibe Frank, Craig Nevill-Manning, Carl Gutwin, Henry Lieberman, the staff of my department, and numerous others. I have borrowed code and ideas from any number of people, including Tom Bonura, Roman Zeiliger, Marty Geier (and his colleagues at Mitre), and the Weka programmers. I am sure I have omitted numerous others; for this I apologise.

I would surely not have lasted four years without the diversions afforded me by my flatmates: Simon, Sarah, Dale, Teri, Donna, Cat, Nick, Jan, Elliot, and others too numerous to mention. Our department is blessed with a fantastic soccer team, comprising many of our students and staff. Thanks Alvin (who started it all), Annette, Aziz, Carl, Chithiran, Conrad, David B., David M., Dong, Eibe, Geoff, Jamie, Len, Max, Kim, Richard, Rob, Steve, Stuart, Talia, Tony, Yong…

Finally, I'd like to thank my parents, my brother Craig, my sister Sharon, and the rest of my family, particularly the Gordons, for their support and encouragement. I dedicate this thesis, and the years of work it represents, to my uncle Doug Gordon, without whose support it could never have been attempted, let alone finished.

Table of contents

List of figures

List of tables

1 Introduction

Computers reputedly excel at repetitive tasks, yet users are frequently forced to perform the same actions over and over again. Repetition can be automated with computer programs, but this is no solution for the majority of users, who have neither the time nor the inclination to learn computer programming. They have little choice but to perform repetition by hand.

Programming by demonstration is an end-user programming technique that lets users create programs by showing the computer examples of what they want to do. The computer observes the user's demonstration and attempts to learn a solution to their problem. When the task is mastered, the user can delegate its completion to the computer. An underlying assumption is that a user who knows enough to perform a particular task knows enough to teach the task to another person by showing them how it is performed. The goal of programming by demonstration is to learn from such a performance, just as another human would, so that the user can instruct the machine even if they do not have the requisite skills to write a computer program. The advantage of this approach is that the user assumes the role of a teacher, not a programmer, and relies on everyday teaching skills, not rarefied programming expertise.

In an ideal world, non-programmers would use programming by demonstration to automate tasks they would otherwise perform by hand. Sadly, the real world is less accommodating, and programming by demonstration is unavailable to the majority of users. This thesis shows how it could flourish on every desktop.

1.1 Programming by demonstration (PBD)

The macro recorder is the simplest and most widely used form of PBD. A macro is a simple program. A macro recorder uses a "tape recorder" metaphor to create a program: the user instructs the system to start recording, performs the actions

they wish to record, and instructs the system to stop recording. The user is asked to assign their macro a name and method of invocation, typically a menu item or combination of keystrokes. Later, the user can invoke the macro, and the actions they "recorded" will be "played back" in the user interface. Consider, for example, a businessman who, at the end of every day, copies the file containing the day's transactions onto a floppy disk, which he later stores off-site. Noticing that his actions have become routine, he decides to automate this task with a macro: he chooses *Start recording* from the macro menu; he drags the transaction file icon onto the disk icon; he gives the *Eject disk* command; he chooses *Stop recording.* He titles this macro "Daily backup" and assigns it a place in a menu. The next day, instead of performing the task by hand, he simply invokes the new *Daily backup* macro. The macro player mimics his actions, copying the file and ejecting the disk.

A macro executes identical commands every time it is invoked. It cannot automate a task that is slightly different in each repetition. What would happen if our hypothetical businessman stored his transactions in a new file each day? When he invokes the *Daily backup* command, the macro player repeats the actions it was shown, making a copy of the file used to demonstrate the task, then ejecting the floppy disk. The macro recorder did not learn to copy the *newest* file, it learned to copy a *particular* file. In fact, the true situation lacks even this sophistication: the macro recorder knows nothing of files and floppy disks, it knows only the mouse actions the businessman made, and blindly copies them each time the macro is invoked. If the transaction file were moved, the macro would move the mouse pointer to its original position, simulate pressing the mouse button, and drag the mouse pointer to the original position of the floppy disk. The effect of these actions is unguessable.

Let us imagine a much smarter macro recorder. This sophisticated tool writes programs that handle new examples, exceptions, variation, and whatever else the job requires. It works at a high level, and has some knowledge of application objects and the meaning of user actions. Such a system observes the businessman's demonstration, learns a program that copies a file, and solicits feedback about this program. A single demonstration, on its own, is not enough to teach it the task—it is not enough to teach another *person* the task without additional explanation—but the user can give new lessons. When the new system copies the wrong file, the businessman tells it so, causing it to undo its actions and ask the businessman for a new demonstration. After examining the second

demonstration, it notices that the newest file was copied both times, predicts that it should always copy the newest file, and asks the businessman to confirm this inference. When it encounters problems, the new system handles them gracefully. If the newest file is moved, it is copied regardless; if the file is removed completely, the system alerts the user and asks what it should do.

The simple macro recorder and this imaginary extension both learn the user's task from his demonstration. The businessman is not a programmer, and could not have written a program to perform backups himself, but was able to program by demonstration. Macro recorders are an existing technology, and widely used in some domains, but are limited to tasks that are identical at each repetition. More sophisticated PBD systems, like our imaginary example, are almost unheard of in the real world. There are numerous impediments to their use, most based on interaction with computer programs and communication with the user. Many are purely technical problems: how can the system monitor the user's actions in another program, or perform actions in another program? How does it know what commands these programs support, and which objects to apply them to, and what their effects are, and how to reverse them? Then there are communication problems: how does the system tell the user what it has learned, or what it will do in unexpected situations, or that it needs new input? Finally, there are problems of risk: how can the user trust the system, knowing that it will perform actions autonomously, and how can they know its capabilities? Does the potential benefit of automating the task outweigh the risk of executing a poorly understood program?

The uncertainties of PBD are many, and its users ill-equipped to cope with them, but the potential benefits—if we can surmount these problems—are great.

1.2 Thesis statement

> *This thesis claims that domain-independent programming by demonstration can be made available in existing applications and used to automate iterative tasks by end users.*

The claim is argued in two parts. The first explains how "domain-independent PBD can be made available in existing applications" on a range of platforms and architectures, and how these can be improved to support demonstrational interfaces. The second establishes that PBD can then be "used to automate

iterative tasks by end users" through a user evaluation of a PBD system and a Gedanken experiment that considers the automation of specific iterative tasks.

Both aspects of the thesis are supported by the design and implementation of the Familiar PBD system. Familiar is designed for end users, but limited by the platform on which it is implemented. These limitations do not prevent end users from automating iterative tasks in the evaluation.

1.2.1 Automating iterative tasks

This thesis considers tasks that are iterative in nature: the user can only complete them by repeatedly performing a command or a series of commands. Such a user, we might say, is facing an iteration problem, which they must solve by performing an iterative task. Iteration problems are encountered in every computing environment, but we are primarily concerned with the needs and abilities of end users, and focuses on iterative tasks in direct-manipulation graphical user interfaces.

A distinction can be drawn between iterative tasks and repetitive tasks. Repetitive tasks are those where the user repeats an action, or a series of actions, at a future time or times. Iterative tasks add the constraint that repetition occurs continuously—they are entirely completed before the user moves on to other tasks—so they are a subset of repetitive tasks. Some tasks, like checking for mail every day at a specific time, are repetitive but not iterative.

An iterative task is automated, and an iterative problem solved, when the user finds a tool that performs or obviates the iterative aspects of the task. The user performs the task once, then delegates the remaining performances to the computer. If a word-processor user wanted to globally replace the word *disc*, for example, with the word *disk*, they could select every occurrence of the former and type the latter in its place. If *disc* occurred 100 times, they would perform 100 iterations, typing 100 words. The *find and replace* tool, however, can circumvent this iteration: the user need merely invoke it, enter the search term *disc* and the replacement term *disk*, and press *replace all*. The tool will automate the task, performing it on their behalf.

1.2.2 End users and applications

The vast majority of computer users—99% by some estimates—are end users.[1]

End users are people who use computers to get their work done. Teachers, architects, secretaries, students, accountants, salespeople—these are end users. They see the computer as a tool for solving problems, and have little interest in the machine itself. Few end users know how to write programs, and the remainder have little motivation to learn, but all are skilled in the computer applications and environments relevant to their interests and expertise.

End-user programming lets end users write programs, usually in the context of some application. Spreadsheets, with their high-level formula languages, are a rare example of a successful end-user programming environment. Programming languages designed for novice programmers (BASIC, AppleScript, HyperTalk, and others) have not been adopted on a large scale because it is necessary to learn to program before they are useful.

PBD is an end-user programming technique, but seldom caters directly to the needs and abilities of end users. The end user's influence on design is pronounced: end users are not programmers, but are skilled with specific programs—generally commercial applications—and are unlikely to sacrifice this expertise and invest time learning new, possibly inferior, substitutes. Consequently, PBD must work with existing applications, rather than attempting to replace them, if it is to be useful to the majority of end users. It must be easy to learn and use, and benefit the user without requiring lengthy training.

1.2.3 Domain independence

Domain-independent PBD systems are those that work in every application. Simple macro recorders are domain independent. They can record and synthesise low-level actions in any application.

Sophisticated PBD systems tend to be domain specific; that is, they work only in a particular application. This allows them to exploit detailed knowledge of the application to help the user, but prevents their use in other applications and in tasks that involve more than one application.

---

[1] Smith *et al.* (1994), p. 56

PBD systems face a trade-off between domain independence and sophistication. The macro recorder is completely domain-independent, but its abilities are very limited. Systems capable of learning more complex behaviours require additional knowledge of their environment and the applications in which they are used. When this information is added, the PBD system becomes dependent upon the application for access to its data and user interface. As the system's requirements increase, the number of applications that meet them decrease. In many cases the requirements are so extreme that the system cannot be used outside the target application.

Simple macro recorders are a rare example of completely domain-independent PBD. More sophisticated systems have been designed and implemented, but few are domain independent in practice (though some are in principle). This thesis will show that domain-independent systems are not only possible, as macro recorders demonstrate, but that sophisticated PBD systems—capable of reasoning about the user's actions, accommodating error and variation, learning from several examples, soliciting and accepting feedback—can be made available in a domain-independent manner.

1.3 Examples of iterative tasks

The motivation for developing PBD systems is to help end users perform iterative tasks. The most common iterative tasks—those that many users encounter regularly—can often be automated by special-purpose tools built into the application, or by using aggregation tools like multiple selection. The rarer a problem is, the less demand there is for its solution, and the less likely it is that application developers will incur the expense of a solution. Consequently, many of the most vexing iterative tasks may never be encountered by other users, and may not even be repeated by the users who report them. This thesis considers a mixture of common problems, many of which can be solved with application tools, such as writing form letters, calculating subtotals, and entering regular sequences; and others which are specialised tasks with no convenient solution.

This section describes three specialised tasks described by three end users: Harvey and Sue, university staff members, and Elisa, a secretary. Each task has, at its heart, an iteration problem, though each uses different applications and has different requirements. Sue's task takes place entirely in a spreadsheet

application, but the others span applications and require domain-independent solutions. Elisa's task cannot be fully automated—it relies on her judgement and intuition—but might be partially automated. These tasks and many others are described in Appendix A; their solution is discussed in Chapter 7.

1.3.1 Image transfer and conversion

Harvey is in the process of relocating to another city and another computer system. He has stored a set of Macintosh PICT image files on a Unix system in Hamilton, which he now needs, in JPEG format, on a Microsoft Windows computer in Calgary. To retrieve the files, he must perform an iterative task on the Macintosh computer: he must download each file, convert it to JPEG format, upload the JPEG file to the local Unix system, and delete the local JPEG and PICT files. Later, he will download the JPEG files to the Windows machine.

Harvey uses a Macintosh computer to convert the PICT files because the new Windows system cannot read them. He uses two intermediary Unix systems because the desktop machines can operate only as FTP clients, not servers, and the files are too big to transfer by floppy disk. Harvey found this problem tedious and time-consuming, but being a resourceful fellow he solved it easily, by delegating it to an assistant.

This task involves iteration over data elements; specifically, iteration over the set of image files. If Harvey's hapless assistant had no tools available for automating iteration, he would perform the task one file at a time, applying each command in turn, as described by the simple algorithm in Figure 1.1a.

In practice, the assistant is unlikely to use these actions. There are many different ways in which this task can be performed, and it is likely that the assistant will use multiple selection to make their performance more efficient. Figure 1.1b shows how the task can be solved by aggregating files with multiple selection, and applying commands to all the files. Only the third step, which involves saving the image files, cannot be automated like this, because there is no way to select a group of image files and apply the *save as* command to them all.

The absolute number of actions the user has to perform is greatly reduced by multiple selection, though it cannot completely automate the iteration. If the assistant was dealing with 100 files, then every opportunity to select them and apply a single command to all would reduce the total number of necessary actions by approximately 100. The task would still be tedious, however, as

(a)
1. Open FTP connection to lucy.cs.waikato.ac.nz and change to PICT directory
2. Open FTP connection to janu.cpsc.ucalgary.ca and create JPEG directory
3. For each PICT file on lucy.cs.waikato.ac.nz
   - Download the file to the local machine using an FTP client
   - Open the file using an image manipulation program
   - Save the image as a JPEG file
   - Upload the JPEG file using the FTP client
   - Select and delete the local PICT and JPEG file

(b)
1. Open FTP connection to lucy.cs.waikato.ac.nz and change to PICT directory
2. Select all the PICT files and download them in an FTP client
3. For each PICT file on lucy.cs.waikato.ac.nz
   - Open the file using an image manipulation program
   - Save the image as a JPEG file
4. Open FTP connection to janu.cpsc.ucalgary.ca and create JPEG directory
5. Select all the JPEG files and upload them using the FTP client
6. Select all the local JPEG and PICT files and delete them

Figure 1.1 Performing Harvey's file transfer task (a) one file at a time, and (b) applying each action in turn to every file.

several mouse actions and keystrokes are necessary to open each image and save it in a new format. The assistant could still expect to perform thousands of individual actions.

This problem, and others like it, can be solved by a PBD system. Harvey's assistant could demonstrate how to convert the first few images, and ask the system to finish the remainder. Even if he used the first, inefficient algorithm in his demonstration, the bulk of the task would be performed by the PBD system, and his workload would reduce to a manageable level.

Using PBD, Harvey's assistant can delegate tedious work to the computer, just as Harvey delegated it to him.

1.3.2 Averaging column data

Sue, a staff member in a university Management Economics department, encountered a problem that appears uncomplicated, but which cannot be automated with conventional spreadsheet tools (reported in Hendry and Green, 1994, p. 1041). She loaded two lists of 500 numbers, data collected from a computer simulation, into the first two columns of a spreadsheet (Figure 1.2, columns A and B). These represent ten blocks of 50 records each. Sue wants to calculate the average of column B for each of the blocks, and store the results in the first ten cells of column C. She knows the formula for finding the average of a block of text; the correct formulae are shown in Figure 1.2, column C.

A	B	C
12	3	=AVG(B1:B50)
32	17	=AVG(B51:B100)
45	22	=AVG(B101:B150)
67	13	=AVG(B151:B200)
7	9	=AVG(B201:B250)
44	8	=AVG(B251:B300)
79	13	=AVG(B301:B350)
64	11	=AVG(B351:B400)
42	19	=AVG(B401:B450)
34	22	=AVG(B451:B500)
90	26	
6	14	
…	…	

Figure 1.2 Sue's finished spreadsheet, showing sample data (columns A and B) and the required formulas in Microsoft Excel format (column C).

Entering the formulae is a simple task, but difficult to automate in a conventional spreadsheet. Sue enters the first formula in the topmost cell of column C, but cannot copy and paste it to the next row because the formulae are offset by one cell and the blocks by 50 cells. If she copies the first formula into the second cell, it will be appear as *=AVG(B2:B51)*, not *=AVG(B51:B100)*.

Since Sue cannot use the spreadsheet's automation tools, she has to perform the iteration manually, either entering each formula afresh, or copying the first formula into the nine remaining cells and editing each by hand.

1.3.3 Indexing document files

Elisa, a secretary, has to manage a large collections of documents that are poorly described by their names and appearance in the operating system. (This is a common problem in operating systems with limited filenames.) Most were created by other people, and she has little idea of their contents. Searching for specific documents can be a time-consuming and frustrating task.

Elisa has been asked to build a simple database associating a document's name and path with information describing it, so that anyone can search and browse the database to find obscure files. The database is to have fields for each document's reference number, name, path, author, date, application, type (letter, minutes, etc) and subject. The documents are in a range of formats, though most were created in Microsoft Word, Microsoft Excel, or Microsoft Publisher. They are in located in many directories in a haphazard fashion, and some are interspersed with files she is not interested in (such as executable program files).

This task asks Elisa to iterate over the document files, examine each, create a database record for it, and fill in the database fields with the appropriate information. It contains irregularities and exceptions: many files are in a consistent format, but others are not, and some should be ignored altogether. In many cases, Elisa will have to use her judgement as she describes the document's subject, or guesses who might have been the author. These inferences are all but impossible for a computer to make—even Elisa may not know the answers—but still the task has a strong iterative element, and contains subtasks that Elisa might like automated, like finding and opening documents, creating records, and entering the name, path, date, and application.

1.4 Thesis structure

The thesis is arranged in eight chapters. The first two describe its purpose, and the body of work upon which it builds. They introduce PBD, survey its history, and summarise the arguments for and against its use. Demonstrational interfaces have previously been applied to a variety of problems, including iterative and repetitive tasks.

Chapters 3, 4, and 5 describe the design and implementation of Familiar, a PBD system for automating iterative tasks. The interface design considers issues arising from the end user's motivations, skills, and attitudes. The implementation is domain-independent and works with existing Macintosh applications using the AppleScript language. Its major components are the event recorder which monitors the user and models applications; the sequence recognition manager, which detects iterative patterns in the user's demonstration; and the pattern analysis manger, which extrapolates these patterns and makes the predictions displayed in the interface. Familiar makes novel use of existing machine learning algorithms to guide prediction and to infer conditional rules from the demonstration.

The next two chapters establish the claims of the thesis. Chapter 6 explains how "domain-independent PBD can be made available in existing applications" by documenting the requirements of PBD systems and showing that they are met by a range of architectures. The strengths and weaknesses of AppleScript are explored, and yield a set of guidelines for writing applications that cooperate with PBD systems and other agents. Chapter 7 establishes that PBD can be

"used to automate iterative tasks by end users". It studies end users completing tasks with Familiar, examines their successes and failures, and concludes that end users are capable of using PBD, but that the circumstances in which they choose to do so are limited. A task evaluation considers Familiar's applicability to a range of example tasks. The current implementation can automate many of these tasks, and most of the remainder would be possible if suitable applications were available and modest additions were made to the inferencing engine.

Chapter 8 summarises the thesis, lists its important contributions, outlines future work, and revisits the claims with reference to the complete document. Five appendices provide supplemental information, describing the example tasks used in the evaluation (Appendix A), training and internal data used by the machine learning algorithms (Appendix B, C), the instructions given to the user in the evaluation (Appendix D), and an extension to Familiar for generating stand-alone AppleScript programs (Appendix E).

Additional works

Parts of the interface, inferencing, and evaluation described in Chapters 3, 4, 5, and 7 were previously reported by Paynter and Witten (1999a,b). These publications describe an earlier version of the interface which has been modified in response to the user evaluation in Chapter 7.

The work reported here was conducted concurrently with research into digital library interfaces and text mining algorithms. These topics are linked to this research by the application of standard sequence detection and machine learning techniques to real-world problems. The Phind interface detects repetitive sequences in free text and creates hierarchical phrase indexes for interactive browsing (Nevill-Manning *et al.*, 1997, 1999; Paynter *et al.*, 2000). The Kea keyphrase extraction algorithm uses machine learning to identify the words and phrases within a document that best summarise it (Witten *et al*, 1999; Frank *et al.*, 1999). Keyphind, Phrasier, and Kniles are digital library interfaces that combine the phrase-based browsing techniques pioneered in Phind with the semantically meaningful metadata extracted by Kea to provide topic-based browsing in previously unstructured document collections (Gutwin *et al.*, 1999; Jones and Paynter, 1999). This research is ongoing.

2 Background

This thesis explores the application of programming by demonstration to iterative tasks. This chapter provides background on both topics.

Iterative tasks are those where the user repeats a series of actions. Computer users encounter them regularly, and are armed with a range of strategies for automating them. Tools like multiple selection and simple spreadsheet formulas are invoked so regularly that they are second nature to the user. They are used frequently, and in combination, with little conscious planning. When these are inadequate, more complicated solutions are brought to bear: end-user programming, macro recording, and increasingly complex formulae result.

Programming by demonstration lets users "program" the computer by showing it examples of what they want to do. The process is analogous to teaching: the user presents specific examples of a task, and the computer learns a general strategy that it can apply to new cases. Its power is that it eliminates abstraction and shields the user from the minutiae of programming. To create a conventional program, the user must describe the entire task in advance, in flawless detail, in an abstract form, in a foreign environment. To create a program by demonstration, users do not need programming skills at all; they simply perform the task within the conventional user interface.

Several PBD systems have examined iterative tasks in limited domains. Iteration problems are particularly amenable to solutions by demonstration because all the examples are readily available: the user can demonstrate on the first few, and the system can give feedback as it performs the remainder. The cost of failure is low: each demonstration contributes to the completion of the task manually, so even if the system fails part of the task is completed at little cost to the user.

Other systems address a broader class of repetition problems. These include repetition at regular intervals (at the same time each day, for example), responses to specific events (including dialog boxes and prompts), and routine tasks (such

as writing a letterhead). These problems are more difficult to automate than iteration. The task is performed at different times, so the user must specify and later recall an invocation technique. Only one training example is available, so it is impossible for the user to provide multiple examples, and for the system to offer immediate feedback on real examples. Each repetition must be performed in the correct order and at the correct time, so the system must infer the boundaries and preconditions of each repetition. These difficulties are overcome with a more complex interface and additional direction from the user. Programs for automating general repetitive tasks are applicable to iterative cases when a suitable invocation technique is available: a program that automates a single iteration is useful if it can be invoked 1000 times without intervention, but not if it must be selected 1000 times from a menu.

This chapter begins by discussing the terminology used throughout this thesis. Section 2.2 describes a range of techniques currently used to automate iteration. Subsequent sections discuss the application of PBD to iteration problems (Section 2.3) and repetition problems (Section 2.4). PBD has been applied to other tasks¾like teaching, building interfaces, and generating code¾that are less relevant to this work, but contribute interaction styles, learning techniques, and user analyses that are generally useful. These are described in Section 2.5. PBD has been explored in research environments, but is not widely used. There are a number of reasons this is so, ranging from the user's reluctance to trust the agent and surrender control, to the difficulty of integration with applications, to interface and inferencing design problems. The final discussion addresses the problems with PBD systems and the difficulties they must overcome (Section 2.6).

2.1 Definitions and usage

This section introduces a range of terms that are widely used in the PBD literature, and explains how they are used—or why they are not used—in this thesis. It defines programming by demonstration, demonstrational interfaces, inferencing, domain and application knowledge, domain and application independence, intelligent and adaptive interfaces, agents, commands, noise, and end users.

Programming by demonstration and demonstrational interfaces

In the glossary of *Watch What I do* (Cypher, 1993a), the first—and thus far, the only—book dedicated to PBD and PBD systems, programming by demonstration is a synonym of programming by example.

> "Programming by example (PBE) — When programs are created through the use of examples. The examples serve as placeholders for abstractions. Myers would like to restrict this term to only systems with inferencing and that allow the end user to create real programs, but this would eliminate Pygmalion and SmallStar, which are often classified as PBE (including by their authors), so others prefer the more general definition." (Myers and Maulsby, 1993, p. 602)

This definition is muddied by two factors: the definition of a program, and the role of abstraction. In its simplest form, a program is "a sequence of instructions executed by a computer" (Myers, 1992). Macro recorders exhibit a rudimentary kind of PBD: a recorded macro is a "sequence of instructions" and is later "executed by the computer". They represent an extreme case that perform very little explicit abstraction.

Myers (1992) prefers a stricter definition: he describes systems that handle variables, iteration, and conditionals as "programmable" and argues that a system must be programmable and use inferencing (defined below) to be described as PBE. Macro recorders are not programmable and perform no inferencing, so are not PBE by this definition. Using Myers' terminology, PBE systems, macro recorders, and other tools that use demonstrations are demonstrational interfaces.

> "Demonstrational Interfaces — Demonstrational interfaces allow the user to perform actions on concrete example objects (often, by direct manipulation), while constructing an abstract program. As defined by Myers, this includes Programming by Example and Programming with Example, as well as interfaces that do not support programming." (Myers and Maulsby, 1993, p. 596)

This thesis uses an inclusive definition of programming by demonstration: it occurs when programs are created from demonstrations. This is a tautological definition, but the term is hard to define otherwise, as the above definition of programming by example attests. Programming by example is not used because

demonstrations may not consist exclusively of examples—feedback, hints, and nudges are valuable input. Macro recorders are included in this definition because they learn programs from demonstrations, have similar goals and interfaces to many inferencing PBD systems, and are particularly relevant to iterative tasks. Systems that learn from demonstrations but create no programs will be called demonstrational interfaces.

Inferencing

Inferencing is the process whereby a system generalises from specific examples (an extensional description) to the abstract concept illustrated by the examples (an intensional description); then instantiates the concept it has learned to generate new examples. The hypothetical system in Section 1.1, for example, performs generalisation when it observes the user copying specific files (the extensional description), and infers they are chosen because they are the newest (the intensional description). When subsequently executed, the program will instantiate the intensional description—*newest*—to a particular instance of the concept: the file that is currently the newest.

In this thesis *inferencing* refers to the process of generalisation and instantiation, and *inferencing PBD* refers to PBD systems that use these techniques.

Domains and applications

Applications are computer programs that are employed by users to perform tasks. In this thesis, we examine end users, and define an application as any direct manipulation program with a graphical user interface. This definition includes graphical operating systems like Microsoft Windows and the Macintosh OS, and more traditional applications like the Microsoft Word word processor, the Microsoft Excel spreadsheet, the Eudora email client, the Netscape web browser, and the GIFConverter image manipulation program.

The term “domain” is a common shorthand for “application domain”, or “domain of application”. It describes a set of applications with the same purpose. Most domains are comprised of several (usually competing) programs. The word processing domain, for example, includes such applications as Microsoft Word, Word Perfect, and MacWrite.

Domain knowledge and application knowledge

An agent requires an accurate model of an application to examine, control, describe, and reason about it. This is typically referred to as "domain knowledge" or an agent's "mental model of an application". The term domain knowledge is used throughout the literature to describe information about applications, but few authors attempt to define it. *Watch What I Do* uses it frequently but does not include it in the glossary (Myers and Maulsby, 1993).

The phrase "domain knowledge" is ambiguous. The use of "domain" suggests knowledge about a class of programs, but in practice it typically refers to a single application. Domain knowledge about a particular drawing program will ideally apply to other drawing programs, but two applications that appear very similar and provide identical functionality can be implemented quite differently, so knowledge of one cannot necessarily be applied to the other. The "knowledge" referred to is similarly vague. It can mean explicit information about an implementation, or denote the action of application-specific algorithms that exploit inside knowledge of the domain.

This thesis refers to the information a PBD system maintains about a specific application as *application knowledge*. The term *domain knowledge* will be avoided because of its ambiguity, but may appear in descriptions of supporting research that use the term.

Domain independence and application independence

Domain-independent systems work with more than one application, and ideally work with every application supported by a platform. Macro recorders are domain independent because they can be used to control any program that accepts mouse or keyboard input, regardless of the application domain. Domain-independent systems are independent in the sense that they are not biased towards a particular application or domain, not in the sense that they are unconnected with it, so they can exploit application knowledge and remain independent. However, application knowledge can easily introduce bias.

Domain independence lets the user solve problems that span applications and transfer skills learned in one application to others. Many systems are domain-independent in a theoretical sense: their techniques can be reused in other domains. These are unsatisfactory: they are untested and do not in fact address tasks involving more than one application. A practical but powerful test of

domain independence is whether previously unseen applications are compatible with the PBD system without developer intervention; it is this level of independence that this thesis aspires to.

Domain independence and *application independence* are used synonymously to describe systems that work with many applications from different domains, and allow the user to automate tasks that span domains.

Intelligent and adaptive interfaces

Current research into PBD often refers to "intelligent" user interfaces and "intelligent" agents. Intelligent interfaces are those that apply techniques from artificial intelligence, including adaptive interfaces, user modelling, natural language processing, dialog modelling, and explanation generation (Waern, 1997). Intelligent interfaces are independent of intelligent functionality: a program with an intelligent interface is not necessarily based on an intelligent system, and an intelligent system—such as a PBD system—does not necessarily have an intelligent interface.

A primary goal of intelligent user interface research is to learn about individual users and adapt to their behaviour (Waern, 1997; Crow and Smith, 1992). This adaptation is not necessary—the majority of applications are not adaptive, yet remain useful tools. PBD interfaces can be called "intelligent" because they learn tasks from the user, but it is important to preserve the distinction between learning a task for a user, and adapting to that user. Consider two people who, by coincidence, independently teach a PBD system a task by giving identical demonstrations. Most PBD systems would infer identical programs from the two demonstrations—they would learn from each user without adapting their behaviour to that user. An adaptive PBD system observing the two identical demonstrations *might* infer different programs for each user, based on its prior knowledge of how those users behave.

The vast majority of demonstrational interfaces are not adaptive, though some, particularly learning apprentices (Mitchell *et al.*, 1994), can and do adapt their behaviour to specific users. Adaptive inferencing is not applied more widely because it is not obvious that it is a benefit. PBD systems are not used enough in practice to make useful distinctions between users, and it is not obvious what information adaptation requires nor how it should be used. The effect of adaptation may be beneficial over the long term, but difficult to detect on specific tasks.

This thesis will describe interfaces that adapt their behaviour to specific users as *adaptive*. It will not use the adjective "intelligent" to describe software.

Agents

The agent metaphor supposes software entities, often with human characteristics, that perform tedious tasks on behalf of the user. Many PBD systems are described as "agents" or "intelligent agents".

Erickson (1997) distinguishes between the agent interface metaphor (the program as an autonomous entity) and agent functionality (personalised actions on behalf of the user), and observes that these facets can be decoupled. A system that exploits the agent metaphor may be programmed as a traditional application, and agent-like functionality need not be expressed through the agent metaphor. The definition of an agent is further complicated by programs that have neither the function nor the form of agents, but use the name out of ignorance or expediency (Foner, 1993).

This thesis uses the term *agent* to refer to software that uses the agent metaphor and software that operates other application programs on behalf of the user.

Commands, actions, and events

Three terms are used throughout this thesis to describe user and agent instructions: commands, actions, and events. They are drawn from several sources and are often interchangeable, but can imply a contextual distinction: a user's input is a *command* or *action*; their internal effect and representation is an *event*; an instruction from one program to another is usually a *command*, and occasionally an *event*. User actions are said to be *recorded, traced, logged*, or *monitored*; and stored in a *log*, *command history* or *event trace*. Terms in each of these sets are interchangeable.

Noise

PBD systems learn from the user's examples. They learn best from good examples, and learn more slowly, or even incorrectly, from bad examples. The elements of a demonstration that impede learning are often called *noise*, and a demonstration that contains noise is called *noisy*.

Noise manifests itself in many ways, usually as unnecessary variation between demonstrations. Some of the most common types include initialisation commands, extraneous "noise events", approximate values, reordering

interchangeable actions, omitting actions, and redundant equivalent navigation commands. Noise enters the event trace from several sources. People habitually indulge in exploratory behaviour, use approximate measurements, and vary the way they perform tasks. Occasionally users make mistakes, and these too are classed as noise. Often, noise is introduced when the user discovers a more efficient way to perform a task during a demonstration. Many PBD systems advocate constraining the user's behaviour to minimise the noise entering the event trace (Maulsby *et al.*, 1989a).

This thesis does not use a strict definition of *noise*. In discussion of iteration problems the term is used to refer to a demonstrated command that does not form part of an iteration.

End users

End users are the ultimate users of computer applications. Generally, they view the computer as a tool that they use to accomplish other tasks, but have no interest in the machine for its own sake.

Before the 1980s there were fewer computers, and consequently fewer users, than there are now, but a greater proportion of these were programmers. Non-programmers, or *casual users*, were characterised by their infrequent computer use, need for a natural-feeling system, and limited mathematical and programming skills (Cuff, 1980). With the advent of personal computers, and their widespread uptake, the number of users has increased enormously but the proportion who are programmers has decreased. The nature of the user's interaction with the computer has changed as generic "shrink-wrapped" software has replaced in-house databases and computers have become more powerful and easier to use. Modern computer use is seldom characterised as infrequent, but the majority of users still rely on "natural" interfaces and lack mathematical and programming skills.

Nardi (1993) points out that non-programmers are neither *novice*, *casual*, nor *naïve*; they have specific computational requirements that are largely met by application software. These users are not programmers because they have not been taught to write programs and have little inclination to learn. They lack skills that programmers acquire through training and experience (and often take for granted) like abstraction, task decomposition, and boolean logic. Most users could learn these skills, but they have little motivation to do so: programming is a difficult and time-consuming discipline, and most people have other interests to

pursue. The goal of end-user programming is to help these users program their computers.

End users can be defined by their ability to use computers and their inability to write programs. This is not a satisfactory definition: an extremely inexperienced user who can use a word processor with no understanding of how it works by treating it as a typewriter is an end user, but a programmer learning a new application on a new platform is not. Ideally, we would like to help both these users, so this thesis adopts an inclusive definition, and defines an end user as any person using a computer application for anything other than writing programs.

This usage is consistent with practices outside the field of computer science, where, according to Webster's Dictionary, an end user is "the ultimate consumer of a finished product." In this broader context, an end user is defined by what they do with a product, not their qualifications as a user.

2.2 Automating iteration

Existing approaches to automating iterative tasks include manual execution, task-specific tools, object aggregation, command aggregation, and end-user programming.

Manual execution

Complex solutions to iterative problems are often unnecessary. Consider a database user who works through a series of records, typing a value in the same field of each. The user is completing an iterative task manually. This is trivial if there are only a few records in the database, but becomes more onerous as the database grows. How large the database must be before it is more efficient to automate the task than to perform it by hand is a complex question, and ultimately answered by each user's intuition, based on their assessment of the costs and benefits of automating the task. A number of factors influence their decision, including the quantity of data and the difficulty of the task; the skill, experience, and confidence of the user; the capabilities of the application program and software environment; the cost and risk of failure; and the expectation of similar tasks in the future.

Task-specific tools

Applications provide tools for solving routine problems. A task that affects a large sub-population of an application's users could be automated in later (or competing) versions of the software. Potter, for example, writes of his inability to print all the odd (or even) pages of a document, but even as his plaint was published this feature appeared in a new version of his word processor (Potter and Maulsby, 1993, p. 562). Users cannot rely on task-specific tools. Many iterative tasks are user-specific and will never be performed by another user; others are one-time tasks that the users themselves will never repeat. These are unlikely to be solved by applications because a new feature must benefit many users before it is cost-effective to add it to a commercial product. Further, tools can only be incorporated into future versions of a program, and—as Potter found—it may be a long time before a new feature becomes available to the user.

Element and command aggregation

Bhavnani and John (1998) observe that the iterative elements of a task take two forms: iteration over commands and iteration over data elements. Any iterative task can incorporate either or both of these dimensions. Users perform iterative tasks efficiently by aggregating elements (so that one command can be applied to many elements) and aggregating commands (so that many commands can be applied at once).

The most widely used element aggregation technique is *multiple selection*. The user selects a group of objects and applies a single command to them all: the effect is the same as applying the command to each object individually. Multiple selection is a simple, effective, consistent, and established tool for aggregating objects. Inexperienced users can use it because it is simple; they are motivated to do so because it is effective; they are able to transfer their ability to new applications because it is consistent; and new applications incorporate multiple selection as a matter of course because it is established.

Multiple selection is powerful, but inappropriate for many iterative problems. Some tasks involve iteration over actions, and element aggregation offers little help. Others involve iteration over data that does not afford selection, such as Potter's odd pages, open windows, or non-continuous text. In many iterative tasks a different variation of an action—or a completely different action—is applied to each element. Finally, multiple selection does not scale well in direct

manipulation interfaces: it rapidly becomes tiresome to select hundreds or thousands of elements.

Command aggregation techniques automate iteration by reducing the number of actions the user must make. Macro recorders let the user identify a sequence of commands, aggregate them into a single command, and invoke them with a single action. In an ideal case, a long sequence of commands is invoked with a keystroke. Command aggregation is seldom useful in isolation: few tasks are solved by aggregating commands without also applying those commands to new data at each invocation. PBD systems that perform inferencing ostensibly aggregate commands, as macro recorders do, but also generalise objects, thus incorporating element aggregation.

Just-in-time programming

Just-in-time programming occurs when the user, in the course of pursuing some larger goal, encounters a computable subtask and creates a program to solve it (Potter, 1993b). The program will be created, executed, then discarded; and the user will continue their original task. Many tools and languages are designed to let end user's write programs just in time.

Unlike traditional programming languages, which professional programmers use to build applications, end-user programming languages let typical users solve domain-specific tasks. Scripting and visual programming tools are available on many computer platforms, and can be used to write programs to automate repetition. These are more accessible than their traditional counterparts and make programs easier to visualise and understand, but still require basic programming skills. Most end users are unlikely to adopt them because they have not learned—and are not inclined to learn—to program (Smith *et al.*, 1994).

Nardi (1993) argues that end users are better served by high-level task-specific programming languages, as are found in spreadsheets and computer aided design, than by general-purpose end-user programming. She observes that

> "...conventional programming languages, no matter how well supported, are not appropriate for the large population of users who lack intrinsic interest in computers and have very specific jobs to accomplish. These users should be supported at their level of interest, which is to perform specific computational tasks, not to become computer programmers." (Nardi, 1993, p. 70)

Users are motivated (and able) to learn a language in a familiar and interesting domain if it minimises the abstractions of conventional programming languages. Nardi identifies several factors that contribute to the success of spreadsheet programming languages: the domain is of interest to users; the languages are accessible and immediately useful, but can be learned very slowly; they are interpreted, and thus interactive and incremental; they have a limited number of high-level, task-specific functions (low-level computational functionality is hidden); control constraints are inherent in a spreadsheet's layout; and they often combine two languages, one (the formula language) for the inexperienced, the other (the macro language) for more experienced users.

There are problems with task-specific languages:

> "First, it is expensive to build the many different task-specific languages that are needed for the myriad uses to which computers are put … A second possible problem with a plethora of task-specific programs is that users will be forced to switch between many different systems, learning a new user interface every time. The third, and most serious, problem is that it is difficult to know just how specific a task-specific system should be." (Nardi, 1993, p. 50)

The latter two problems are exacerbated by the fact that many users learn very slowly, if at all. These problems illustrate the trade-off between consistency and specialisation. Nardi advocates specialisation because end users are interested in specialised domains; the advantage of consistency is that users can easily transfer skills. These issues are revisited in Chapter 3 because they are pertinent to the design of PBD systems. Task-specific languages have other problems: they assume that tasks are contained within applications, and risk becoming focussed on "typical" tasks at the expense of those that do not fit the task model like the spreadsheet formula example described in the introduction (Section 1.3.2).

2.3 PBD and iteration problems

PBD can be used to solve iterative problems that are intractable to other tools. In one example a user works through their email messages, copying the subject line of each into a numbered list (Appendix A.5). Multiple selection cannot automate this task because several complex commands are applied to each message. A macro recorder is unsuitable because the task changes from one iteration to the

next: different text is selected, line numbers change, and text is pasted into different locations. A script would handle these complexities, but requires programming abilities that most users do not possess. The Eager PBD system avoids all these pitfalls, and completes the task for the user (Cypher, 1993b).

PBD systems for automating repetition share many features. We can imagine combining their common elements to form a hypothetical generic PBD system. This generic system is embedded in an application program, giving it direct access to the data it requires, but preventing its use in other applications. The user initiates PBD by signalling that they are about to begin a new task, and performing the first iterations. The system, now activated, monitors the user's actions and eventually infers their intent. It begins showing the user what it has learned by predicting future actions, using real data to give examples. When the user has seen enough feedback to be confident that they have taught the task correctly, they can request that the system perform a single step, a full iteration, or the remainder of the task. The remainder of this section will describe several real PBD systems for automating iteration, and how they differ from this hypothetical example.

The predictive calculator is a tool for solving iterative calculations (Witten, 1993). Its interface resembles a hand-held calculator, and lets users automate tasks with no knowledge of programming—they need only know how to use a calculator. It is ideal for iterative calculations, such as evaluating the formula $1 + (\log x) / (8 \log 2)$ for increasing values of x, that might otherwise be performed with a computer program or by hand. The calculator differs from the generic system in that it is always active and performs tasks automatically, relying on typical calculator usage habits to ensure that its results are visible. Mistakes can be rectified with an *undo* button.

Eager is an agent for automating iteration in Apple's HyperCard multimedia environment, where it successfully automates the email example above (Cypher, 1993b). Eager's interaction style has several unique features. It does not need to be explicitly activated because it continually watches what the user does. The user interface appears when Eager detects repetition, and uses *anticipation highlighting* (the colour of the next interface component to be used is changed) to communicate the prediction. When the predictions are routinely correct and the user is confident that Eager has learned the task, they can invoke the Eager

interface and instruct it to take over. Eager will write and execute a HyperTalk script to complete the next step, the next iteration, or the remainder of the task.

Eager is the most polished of the iterative PBD systems, and is theoretically application-independent. However, it is not in general use because it does not tolerate mistakes in demonstrations, cannot explain the predictions it makes (users were forced to trust the system and did so reluctantly), cannot display entire programs, requires a detailed model of the application, requires cooperation from the application, and requires changes to the Macintosh operating system. The last three requirements in particular are too difficult, and too expensive, for widespread uptake.

Metamouse is a teachable agent for performing simple tasks in a drawing program (Witten and Maulsby, 1993). The system is intrusive: it continually asks users to confirm its inference. After each action it asks the user to identify important geometric relationships, and when it recognises a repeated action it asks the user if it should take over. It then confers with the user as it performs the next iteration, and finally completes the task. By taking over as soon as possible, the variation between demonstrations is reduced. Metamouse relied on “touch” relationships (such as the intersection of a line and the corner of a rectangle) and “guide” objects (objects created to illustrate geometrical relationships like distance, then deleted). The interface was ultimately unsuccessful because users found these constructions unintuitive.

TELS is a text editor that learns iterative tasks, such as reformatting bibliographies and address lists (Witten and Mo, 1993). Unlike the generic system, the user gives a single example, identifying its beginning and end, which the system generalises into a program and executes. As the program runs, the user interactively debugs the system’s inferencing.

The Turvy experiment asked users to teach data descriptions to a simulated agent while performing iterative text editing tasks (Maulsby, 1994). Turvy was a human masquerading as an agent, and easily able to infer rules, recognise spoken hints, generate speech, and interact with applications. It was successful because it used natural teaching techniques, learned concepts incrementally, made suggestions rather than requiring specifications, and kept users informed through spoken feedback. In a typical transcript, the user gave an example, then the agent interactively completed the task. The CIMA concept learner infers data

descriptions from the instructions identified in Turvy, but has not been used in an interactive system.

Other interfaces are quite different from the hypothetical generic example introduced above. Tatlin seeks similarities between data in a spreadsheet and a calendar by examining each with AppleScript; the user's actions are ignored (Gaxiola, 1995; Lieberman, 1998). The dynamic macro interface strives for simplicity (Masui and Nakayama, 1994). When the *repeat* key is pressed it detects iterative cycles in the user's recent keystrokes, and immediately executes the next iteration. A second key, called *predict*, is used to reject the prediction and request another. Like a macro recorder, the interface uses low-level events, there is no variation between repetitions, and each repetition must be separately invoked. The interface is simple and expressive, but relies on a powerful undo mechanism. Other systems tackle specialised iterative tasks like code generation and chart building; these are discussed in Section 2.5.

2.4 PBD and repetition problems

Many repetitive problems are not iterative problems: the user does not always perform the remaining repetitions as soon as they have finished the first. Iteration is simplified by the ready availability of data: tasks can be automated completely, immediately, and interactively. In other repetitive situations fewer examples are available and the program is not invoked immediately.

When task data is scarce the user cannot provide multiple examples and feedback cannot exploit real examples. Ultimately, these obstacles are surmounted by increasing the role of the user. The PBD system is told the boundaries of each repetition explicitly, usually through the "tape recorder" metaphor: the user signals the start of the task, performs an example, and signals its end. The user gives more instructions than in the iterative case (where only the start of the task is signalled), and still must specify and later recall an invocation technique. A single example is seldom sufficient input to an inferencing system, so the user has to explicitly edit the generalisation, or find more examples (indicating the beginning and end of each).

Solutions to repetitive problems can be applied to iterative tasks when appropriate invocation techniques are available. Iterative tasks require the ability to execute a program many times in succession, or to edit the program and insert

a top-level loop. There are a range of other invocation techniques (Kosbie and Myers, 1993a). Routine tasks can be invoked explicitly (macro recorders use hot-keys and menus), performed in response to a predetermined user action (e.g. Myers, 1998), or by anticipating repetition and providing automation tools (e.g. Ruvini and Dony, 1999). Tasks can be performed at regular intervals following the user's explicit instruction (e.g. Myers, 1998) or by extrapolating from their past actions (e.g. Yvon and Piernot, 1995). Responses to system events can be automatic (e.g. Potter, 1993a) or recommend defaults (e.g. Mitchell *et al.*, 1994).

2.4.1 Macro recorders

Macro recorders based on low-level events are available for most graphical user interfaces. The recorder operates by intercepting and storing low-level events like keystrokes, mouse movements, and button actions. The player operates by pushing simulated keystrokes into the keyboard buffer and mouse actions into the mouse buffer. The operating system reads these buffers and passes the actions to the active application; neither can distinguish between the macro recorder's instructions and those generated when the user physically presses keys. Sophisticated macro systems use high-level commands, not low-level events, and allow the user to edit the recording; some integrate macros with conventional scripting languages and programming environments. These adaptations are built into specific applications, and not system-wide solutions.

Macro recorders are widely used in some domains, but novice users encounter many difficulties. They are hard to learn and use, and users have difficulty transferring skills between macro recorders with different interfaces and languages. Some recorders are tied to particular applications, and can only be used on tasks that take place entirely within the application. It can be difficult to record a macro correctly the first time: the user must plan the demonstration in advance, then execute it with a low (usually zero) error rate. Finally, macros are inappropriate for many iterative tasks because each playback is identical and there can be no variation between iterations.

2.4.2 Extending macro recorders

Some PBD systems might be described as extensions to the macro recorder interface that use inferencing techniques to automate a broader range of tasks. By combining their common features, we can build another hypothetical generic PBD

system. This composite system retains the “record and play” metaphor but uses high-level events. The user gives a single demonstration, which the system turns into a program that performs the same sequence of high-level commands but generalises from the objects in the demonstration to appropriate abstractions. The user can inspect the program, which is displayed in a visual language that explains how the objects have been generalised, and can then use a set of provided tools to override the system’s generalisation. The user is finally asked to specify an invocation technique: either a menu, button, or hot-key assignment; invocation at regular intervals; or invocation in response to a system event.

SmallStar is an early example: a visual shell (an operating system using a graphical user interface) modelled on the Xerox Star with a PBD system built in (Halbert, 1993). Scripts are recorded, then displayed in a text-based format, using icons to reduce the conceptual gap between objects and their representations. The user can add new steps and control structures, or edit object generalisations with “data description sheets”. Pursuit is another visual shell (Modugno and Myers, 1997). Functionally, it is equivalent to SmallStar, but programs are represented in a visual language that illustrates the effect of actions on data by colouring data consistently, even when it is transformed. A comparative study suggests that it is easier for users to comprehend and generate than SamllStar’s language (Modugno *et al.*, 1996).

The Application Independent Demonstrational Environment (AIDE) is a Smalltalk library for the Macintosh that allows programmers to add inferencing PBD functionality to their programs (Piernot and Yvon, 1993). The user can record and play scripts, review the system’s inferencing, and correct its generalisations through additional examples and dialog boxes. AIDE is domain-independent, but is not widely used because few developers use the SmallTalk platform, and fewer still implement the methods AIDE requires.

A frequent extension to the generic interface is the ability to pass an object or objects to a macro as a parameter.

Mondrian is a graphical editor that can be extended through PBD (Lieberman, 1993b). The user can create new parameterised commands by selecting an object (or objects) and demonstrating the new command upon it. A button that depicts the example before and after the demonstration is created and added to the user interface. It can be used to apply the command to new objects or expanded into a visual representation of the program. The AgentScript interface applies a

similar style of interaction in the Macintosh Finder using the AppleScript language, moving PBD closer to mainstream platforms (Lieberman, 1998). These parameterised programs might be initiated in response to patterns the machine has been taught to recognise through examples (Lieberman *et al.*, 1999).

Other interfaces diverge significantly from the tape-recorder metaphor.

DemoOffice automatically generates parameterised macros for copying data from an email client to a database application (Sugiura and Koseki, 1996). It relies on domain knowledge to initiate macro creation and perform generalisations. If its macros remain unused they are eventually forgotten.

Triggers is a demonstrational tool for automating tasks in the Macintosh interface (Potter, 1993a). To create a program, the user records trigger conditions and actions. Later, when the conditions are met, the system will perform the actions. Triggers interacts with the user interface at a low level: conditions are based on patterns of pixels on the screen, and actions are simulated mouse and keyboard actions. This low-level access allows Triggers to work on a system-wide basis.

2.4.3 Editable histories

Editable histories are an alternative to the tape-recorder metaphor. Instead of signalling a demonstration in advance, the user performs a demonstration, then examines the event trace and indicates which of the commands they have issued are relevant to the demonstration. The operations are then edited and invoked using the techniques that have been described above.

Chimera is a graphical editor with demonstrational features (Kurlander, 1993). The event trace is displayed using the *comic strip metaphor*: each user action is represented by a picture of the affected objects as they appear after the command. Comic strip readers habitually assimilate temporal information and abstraction (McCloud, 1994), and the metaphor leverages these skills. In Chimera, the comic strip serves as a history of the user's actions, and can also be edited to create parameterised macros. Mondrian and Pursuit (described above) also use the comic strip metaphor to represent programs.

Topaz adds "scripting by demonstration" to any application built with the Amulet toolkit (Myers, 1998). Amulet provides a hierarchical command architecture and event trace designed to support undo operations and PBD (Myers and Kosbie, 1996). The user can select actions from the command history,

generalise them in an editing environment, and specify a wide range of invocation techniques.

2.5 Other demonstrational interfaces

Demonstrational techniques have been used for a variety of purposes that have little to do with iteration or repetition, some of which are described in this section. Maulsby (1994) discusses a wider set of instructible systems, analysing their ability to infer the user's task, method of instruction, and feedback techniques.

Teaching

Lieberman (1993a) observed that people and computers learn in fundamentally different ways. Computers learn strictly from an abstract set of instructions, while people need examples to illustrate theory. As a result, people are used to teaching other people with examples, not in the purely theoretical terms required by a computer. Instructing computers would consequently be easier if programmers could give concrete examples of what they wanted to achieve. His Tinker system creates Lisp functions from examples of their input and output. It starts out by remembering which outputs go with which inputs, builds a better understanding as more examples are given, until it finally learns complete programs.

The first macro recorder was possibly the Instant Turtle tool, created by a teacher to learn Logo procedures (cited in Lieberman, 1993a, p. 53). More recently, Cockburn and Bryant (1997) describe a teaching environment that displays and issues Logo commands through turtle movements, a history list, and an iconic programming environment. Users can demonstrate actions on the turtle, and convert them to procedures with the other interfaces. Other educational programs include demonstrational features, like Opsis, that allows students to demonstrate algorithms like tree insertion on abstract examples (Michail, 1998), and Rehearsal world (described below).

Algorithm description

Pygmalion is a tool for creating and illustrating algorithms (Smith, 1975). It was designed for computer scientists and requires background knowledge of programming, but allows the user to create algorithms within a visual, animated,

interactive interface. Icons represent variables and other objects, and no abstraction is performed: example data is always present on the screen. Smith has since identified two key features of this approach to programming:

> "It relies on editing an artifact rather than typing statements in a programming language. Editing has proven to be easy for people. Everyone who uses computers—over 100 million people today—can use text and graphics editors, but hardly anyone can program.
>
> "The screen images always contain concrete examples of the programs data. This eliminates the entire class of errors due to abstraction." (Smith, 1993, p. 25)

Interface construction

The Rehearsal World is an end-user programming environment that lets teachers create interactive lessons for students (Fizner and Gould, 1993). The program uses a *theatre metaphor*: the screen is a stage, each object is a performer, messages are cues. Performers are taught actions by demonstration and there are no abstractions. Data values are always visible.

KidSim is an environment that lets children create simulations by drawing objects, placing them on a game board, and demonstrating simple "graphical rewrite rules" to move and transform them (Smith *et al.*, 1994; Section 3.1.2). Programming in KidSim is necessarily very simple, so that even children can do it. No programming language is used, and abstraction is eliminated by keeping every object visible. Behaviour is taught to individual objects, but applies to object classes, so that things that look the same will behave the same. KidSim was later known as Cocoa and as StageCast Creator. This thesis will use its original name, which is more evocative of its purpose.

Pavlov is a tool for creating interactive animated interfaces by "stimulus-response demonstration" (Wolber, 1996). The user demonstrates animated behaviours for drawing objects, and the event that triggers them, such as user actions or the passing of time. This extends conventional animation tools, which allow no interaction with the user. Gamut is an environment for building interactive interfaces like simulations and video games (McDaniel and Myers, 1999). The user sets up the game board, then teaches the system how the objects react to user actions and other events. If the user presses the right-arrow key, for example, the spaceship object might move to the right. Generally, Gamut can be

shown the rules of a game, but not how to play it. A range of tools are provided to extend the behaviours that Gamut can be taught.

Source code generation

Peridot and Garnet are systems for generating user interface source code. They use PBD because it is a productive tool for generating user interface code, not as an end-user programming technique. To generate a procedure with Peridot the user enters a procedure name, gives examples of input parameters and variables, then draws the user interface they desire (Myers, 1993a). When an input parameter or variable appears in the drawing, Peridot recognises and generalises it, inferring iteration over input parameters and changes to variables. Peridot always gives an English description of its inference and asks for confirmation before making changes. An evaluation found that even experienced programmers generated code faster in Peridot than in their favourite environments. Garnet extends Peridot (Myers, 1993b). It has several parts, including tools for designing new interface components, specifying constraints, creating dialog boxes, coordinating styles, and fetching user actions. Inferencing is unsophisticated, domain specific, and usually successful because of the restricted domain. Feedback is provided by displaying the generated code.

Specialised tasks

Gold is an application for creating bar charts, pie charts, and customised graphs by demonstration (Myers *et al.*, 1994). The interface consists of a spreadsheet and a drawing window. The user creates a graph by sketching an approximation in the drawing window, and associating data from the spreadsheet with the objects they have drawn. Gold infers relationships between data and drawing (e.g. cell value is proportional to rectangle height) and generalises the relationship over the spreadsheet data (e.g. create a rectangle for each cell). The user provides feedback about inferencing by editing the drawing. Another system, DemoOffice, infers similar relationships between an email client and a database (Sugiura and Koseki, 1996).

Tourmaline is a demonstrational text formatting system (Myers, 1993c). As in a conventional word processor, the user can create, store, and reuse text styles explicitly. Styles are also applied automatically: Tourmaline uses its knowledge of formatting and library of previous styles to analyse text and infer the appropriate style. Inferencing is based on built-in (but editable) knowledge of text formatting.

Schlimmer and Hermens' (1993) interactive note-taking system allows users to record semi-structured information. It generalises example text patterns, such as computer models or dress patterns, to provide two tools: an anticipation feature that can be used to enter the next string, and a customised button-box interface.

The calendar apprentice learns to schedule meetings from experience (Mitchell *et al.*, 1994). Each meeting has several attributes, including its date, time, location, and attendees. The system observes the user entering meeting details, and attempts to model their decisions with machine learning. Later, when the user enters a meeting, it suggests default values for the start and end times, location, and other parameters. The calendar apprentice is an example of an adaptive interface: as the learning algorithm observes more of the user's interaction with the system, it adapts its behaviour to that user's preferences.

2.6 Discussion and criticism of PBD

PBD is used in many systems, for a variety of purposes, but has not matured to the point that it is an established commercial technology. This section discusses the reasons it has not been widely adopted, considering in turn problems with agents, problems particular to PBD, and the challenges facing PBD systems.

2.6.1 Problems with agents

Most PBD systems have agent functionality, and many exploit the agent metaphor. Agents are currently popular, as noted above, but are not without critics. The most common criticisms of agent interfaces are that they reduce or remove the user's feeling of control, they can autonomously perform destructive actions, they raise unduly high expectations, and they are practically untested in scientific evaluations (Norman, 1994; Shneiderman, 1997).

These are all justified criticisms of PBD. The first two—perceived and real loss of control—were reported by Cypher (1991) and Maulsby (1994). They are inevitable when the user delegates authority to other entities, be they virtual or human, but can be minimised (see Chapter 3). The user's expectations are easily raised: people attribute human qualities to interfaces that affect human characteristics (Erickson, 1997). Anthropomorphic interfaces raise their expectations further, suggesting the agent is as intelligent and flexible as a person. Most PBD systems are limited to specific domains and problems, but users will

not recognise these limitations if they are oversold by the interface. The final criticism, that agents are still an untested technology, can only be addressed by future usability studies.

2.6.2 Problems with PBD

Nardi (1993) discusses many of the weaknesses of PBD systems: they cannot express conditionals and terminating conditions clearly, they lack error correction, they do not accommodate exploratory behaviour, they inappropriately distribute processing and programming effort, they take control away from users (discussed above), and they are expensive and difficult to create. She later suggests that approaches like PBD assume that "end users remain rather deficient" and that this view will "ultimately mean inhibiting their growth and the kinds of applications they can create" (Nardi, 1993, p. 115).

Nardi's criticisms are discussed in more detail below; most are acknowledged elsewhere in the PBD literature. Nardi concludes her discussion by advocating task-specific demonstrational techniques:

> "A challenge for the future is to determine how to fit specific programming by example techniques into larger programs that offer power and flexibility to end users." (Nardi, 1993, p. 78)

Termination, conditionals, and error correction

Criticism of terminating conditions, conditionals, and error-correction is founded on the inconsistency with which they are addressed. Several projects handle specialised cases well, but no inferencing and representation system has proved universally superior, and most have obvious deficiencies.

Little research has specifically explored the inference and representation of terminating conditions. Termination is often trivial or obvious, particularly when the user iterates over data (e.g. every word in a document, every file in a folder). Eager offers to "finish the task" for a user without explaining how; it achieves this by iterating until it is unable to predict future actions or causes an error (Cypher, 1993b). Inferring termination conditions is difficult when the solution is not inherent in the structure of the task data.

Conditionals are more widely addressed. Programmable systems can be divided into two groups: those that infer conditionals, and those that let the user explicitly add branches to the program code. The latter have similar problems to

end-user programming systems; in either case, visually representing conditional code is more difficult than a straight-line program. Error correction techniques are available in even greater variety; different systems allow the user to give new examples, offer feedback, or directly alter the program.

Exploratory behaviour

People habitually indulge in exploratory behaviour, make mistakes, and vary the way they perform tasks, but PBD systems rely on simple, error-free demonstrations. Inferencing systems assume every action the user demonstrates is important, and are easily confused by noise in the event trace. Maulsby *et al.* (1989a) encourage the user to constrain their actions to give good examples; Nardi asks how much should users be asked to change in order to program in a supposedly natural manner. PBD researchers acknowledge this problem when they observe that demonstrations are noisy; instead of accommodating the user's propensity for variation, however, they design techniques to restrict it. Some systems discourage variation very subtly; Maulsby (1994) for example advocates collaborating with the user as soon as repetition is detected, minimising free variation between repetitions. Another solution, proposed in Chapter 3, is to create noise-tolerant systems that expect irregularities.

Distribution of effort

Many task-specific languages solve iteration without abstraction, but PBD places greater burden on the computer. Nardi suggests that this is unnecessary, and that PBD systems "incorrectly distribute processing and programming effort" (Nardi, 1993, p. 72). Advocates of PBD argue that this is in fact a strength because, by increasing the effort made by the machine (and machines are increasingly able to handle heavy computational loads), the effort required of the user can be reduced.

Expense and difficulty

PBD is expensive and difficult to implement compared to alternative end-user programming techniques (though these are also expensive). However, the cost is incurred only once and the benefits continually accrue; and many useful tools are in widespread use despite being difficult to create. In practice, the difficulty of implementing PBD will affect the system's usability—Chapter 7 shows that the Familiar system would be improved if every application was well designed—but this is true of any end-user programming technique, and there is no evidence that

a PBD system would be any more expensive or any less useable than a task-specific language on the same platform.

2.6.3 Challenges for PBD research

Criticism of PBD tends to come from outside the field—active researchers view the same issues as challenges, and anticipate solutions. This section summarises the main challenges facing PBD researchers. It owes much to Myers' (1992) summary of research issues, Potter's (1993b) discussion of just-in-time programming, and Yvon and Piernot's (1995) list of challenges.

User and developer attitudes

Perhaps the most significant challenge facing PBD tools is convincing developers and users that it is a useful technique. Myers (1992) identifies several issues that should be addressed: convincing users that PBD is useful, identifying when and how it is best used, showing how to build effective systems, and easing implementation difficulties. Unfortunately, a vicious circle is in effect: consumers will not adopt PBD until developers provide it, and developers await consumer demand. This impasse cannot be quickly resolved because of the difficulty of adding PBD to existing applications; incremental changes are more likely as limited PBD tools inspire limited demand, which in turn inspire more changes. The catalysts for this change are the standardisation of user interfaces and their libraries, increasing architectural support for end-user programming (Apple Computer, 1992–1999; Microsoft Corporation, 1996), and consumer demand for so-called "intelligent agent technology".

Inferring the user's task

The ability to generalise the event trace is a significant component of a PBD system. The first obstacles are gaining access to the event trace, the user's data, and application functionality. Several approaches to these problems are proposed in Chapter 6, which describes the requirements of PBD systems and architectures that meet them.

Having captured the event trace and gained access to any necessary data, the next challenge is to infer the user's intent. This problem takes different forms in different PBD systems, but usually involves identifying examples and learning data descriptions. Inference does not stop with the user's demonstration: other issues arise as the program is invoked, such as identifying generalisation errors,

ensuring long-term correctness, incorporating feedback, and handling exceptions. These are easiest to address preventatively—by ensuring that the program is correct and complete before it is run—but this approach sacrifices one of the advantages of demonstrational interfaces: the ability to create a program incrementally, handling new cases as they arise.

User interaction

Potter (1993b) identified obstacles to interaction with just-in-time programming systems that apply to PBD, including the effort of entering a program and the effort of invoking it.

The effort of entering an algorithm with PBD system is ostensibly low, as the user need only demonstrate the commands they would normally perform, but is complicated by the user's subsequent visualisation of, and interaction with, the inferred program. The first difficulty is to describe the program to the user; almost all the systems above do so with simple programming languages. The next is to let the user interact with the visualisation to detect and correct errors. Some systems simply accept binary feedback about their inference; others go further, allowing the user to edit constructed programs, or even to perform generalisation explicitly (without system inferencing). The effort of invoking the algorithm is not limited to initiating it, but includes controlling the speed and extent of its execution, and undoing its actions.

2.7 Summary

This chapter has surveyed research into PBD and automating iteration problems. PBD systems for automating iteration exhibit many common features, including application dependence, direct data access, invocation techniques, pattern detection, feedback on real data, and the ability to execute single steps, whole iterations, or the remainder of the task. Many other systems solve repetitive (not iterative) tasks by extending the macro recorder interface to incorporate inferencing and better interaction and invocation techniques.

There are many criticisms of PBD and barriers to its acceptance. Some are intractable problems, others are addressed by the design principles of Chapter 3.

3 Design considerations

This thesis claims that PBD can be made available in existing applications and used to automate iterative tasks by end users. This chapter considers the design of such a system, and formulates guidelines for end-user PBD. Familiar, a PBD system designed in accordance with these guidelines, is then introduced.

Chapter 2 described a range of PBD techniques for automating iteration, but only one, the simple macro recorder, is widely available. This situation arises because most PBD researchers focus on a single aspect of PBD—typically an inferencing or interface technique—and examine its effect on the user's experience. Modern principles of user interface design suggest that this is the wrong approach to designing software. Instead we should start with the user's needs and abilities, and design interfaces to support them.

The end user's motivation, skills, and attitudes suggest a design unlike the research prototypes in Chapter 2. The user's motivation is assumed to be to automate iterative tasks like the examples in Appendix A. Most of these involve simple iteration in a single application, others span applications or cannot be fully automated; these factors determine the capabilities of the design. The end user is a non-programmer who is nonetheless a proficient computer user. These skills suggest that PBD will be useful in the context of the user's workspace, but should be simple and minimise the user's role in program generation. The user's attitudes will affect our design—users are risk averse, reluctant to program, and have preconceptions about agent-based interfaces—so the system should clearly convey its capabilities and limitations, and explain its inference and intentions.

These observations are distilled into four broad guidelines for designing a PBD system: the use of existing applications, simplicity, minimising user effort, and educating the user.

A domain-independent PBD system for automating iteration in existing applications, as posited in the thesis statement of Section 1.2, is consistent with

these guidelines. The existing systems described in Section 2.3 are not, for two reasons. First, it is technically difficult to instrument PBD in the user's workspace, a problem that is discussed at length in Chapter 6, and comparatively easy to build research applications. Second, PBD is primarily a research area, and researchers are interested in developing ideas, not saleable software. Their implementations are constrained by requirements incompatible with giving the end user a complete and polished implementation.

Familiar is a PBD system for automating iteration designed for end users. It strives to be simple and minimise user effort. Though it uses agent functionality, the agent metaphor is eschewed, and natural language descriptions are used to explain its intentions and reasoning. Familiar is application-independent in an immediate and practical sense—it works with new, unseen applications as soon as they are installed on the host computer.

Familiar uses unmodified Macintosh applications and exposes users to an established scripting language, AppleScript. Every Familiar instruction presented to the user is an AppleScript command, and the user—possibly without realising it—gradually learns the syntax. Appendix E explains how Familiar's straight-line programs can be transformed into full programs using variables, conditionals, and iteration, and then presented to the user as source code or as executable programs.

Familiar's inferencing is described in Chapter 4, its use of machine learning in Chapter 5, its platform in Chapter 6, its capabilities and evaluation in Chapter 7.

This chapter considers design. The first section gives an overview of the designs present in the literature, ranging from feature-centric research prototypes to successful end-user programming systems. Section 3.2 examines the user's motivation, skills, and attitudes, and their effect on the design. These features determine a set of guidelines for PBD design. Familiar is introduced in Section 3.3 through a series of examples. The last section discusses Familiar's limitations and the guidelines with which it fails to comply.

3.1 Existing PBD designs

Most of the PBD systems for automating iteration in Chapter 2 are intentionally feature-centric, and unsatisfactory for everyday use. This section considers the

implications of these designs, designs that consider their user base, and other successful end-user programming systems.

3.1.1 Feature-oriented designs

The PBD systems for automating iteration in Section 2.3 are designed and implemented as prototypes, and none are in widespread use. Most are implemented to explore a new learning technique or interface feature: a minimal application is created, PBD functionality added, and the feature evaluated. The design does not begin with the needs and abilities of the user; instead, the role of the user and application is to test the innovative component.

The systems produced by feature-oriented design are unusable outside the laboratory environment (where, by and large, they satisfy the requirements of their creators). Neither the applications nor the PBD systems are suitable for end users. The applications are incomplete and error prone because the developers focus on features relevant to their evaluation and ignore irrelevant ones. They often rely on special environments and cannot work in a normal setting. Further, they are unfamiliar to end users, so any evaluation tests the end user's understanding of the new application as well as their ability to program by demonstration. The PBD systems are application specific, or task specific, and even those that are domain independent in principle are not demonstrated on more than one application. New programs must be written, or significant changes made to existing applications and operating systems, to apply their techniques.

3.1.2 User-oriented designs

Some PBD systems are designed for specific groups of users and optimised to suit their needs. KidSim, for example, is tailored to users who are children (Smith *et al.*, 1994). Maulsby (1994) considers a broader range of users in the design of communication channels for CIMA.

KidSim, one of the few commercial applications based on PBD, was consciously designed to target a specific user group (Smith *et al.*, 1994). As the name suggests, these users are children, and the purpose of the program is to build simulations. The aptitudes of the end users are anticipated by eliminating the need for a programming language, and refined according to a set of user interface principles for programming environments. Briefly, the authors advocate visibility (interactive, modeless interfaces), copying and modifying (as opposed to creating

from scratch), seeing and pointing (not remembering and typing), concrete (not abstract) representations, familiar conceptual models, and minimising the translation distance between people's mental models and the system's behaviour. They observe that millions of end users can use their computers as editors by directly manipulating visual representations of objects, and attempt to apply this philosophy to programming.

It is likely that hundreds (even thousands) of end users have created simulations in KidSim, making it one of the most successful demonstrational environments, and many more have downloaded finished simulations from the internet and executed them. This is testimony to the systems ease of use. However, a formal evaluation suggests that KidSim's use of familiar representations causes children to assume that objects will comply with real-world expectations rather than examining their actual behaviour, and that children lack the precision and perspective to properly understand the simulations they create (Rader *et al.*, 1997). These criticisms are not severe when children build worlds in an exploratory manner and personally assign the behaviours they think appropriate, but become problematic when the child is asked to work in a directed manner and to interpret an existing simulation.

Simulation problems differ considerably from iteration problems. Iterative tasks are necessarily directed, and though the user may explore alternative solutions, the ultimate goal is fixed. More encouragingly, this thesis concentrates on tasks that adults perform, so the users can be credited with greater precision and experience than children, and the tasks are performed on computer data, not simulated physical objects, so the danger of faulty assumptions is reduced. KidSim's design principles are applicable to most end-user programming environments, though they are unattainable ideals when tasks involve hidden assumptions and relationships, when data is beyond the systems control, and when applications cannot be fully accessed or described.

Maulsby (1994) discusses the design of PBD systems that exploit the end user's teaching skills. He proposes five goals for an agent that learns data descriptions from end users: learn a wide range of appropriate concepts, learn efficiently, minimise the effort of teaching, minimise the effort of learning how to teach, and give users control over the agent's learning and performance. These goals are geared towards learning complex data descriptions using feasible machine learning techniques. They are explored through a Wizard of Oz experiment (a

human actor stands in for the agent, providing learning, speech recognition, speech synthesis, and application control) which allows researchers to observe "natural" forms of instruction and distil the purely functional requirements of a concept learner.

Maulsby explored the descriptions it is possible teach to feasible machine learning technology, formalising user iteration in instructions for classifying examples, rules, and features. The present research has a different emphasis, surveying the tasks that PBD can solve in existing environments. Although the two designs have many similarities, this chapter ultimately advocates simplicity over expressiveness, and the resulting implementation relies on examples (Maulsby's example classification) and simple feedback (rule classification), but ignores user hints (feature classification).

3.1.3 Successful end-user programming

Spreadsheets are the most successful end-user programming environment, and have been studied in detail. Hendry and Green (1994) give a comprehensive summary of this work. Nardi and Miller (1990) interviewed spreadsheet users to find out why the spreadsheet model is so popular among end users. They concluded that the main advantages of the environment are the clear visual format and the set of useful, task-specific, high-level commands. They also stressed that users can build programs to model their interests very quickly, even if their knowledge of the formula language was incomplete.

Macro recorders are often touted as the single example of PBD in widespread use. They are widely available because they are comparatively easy to implement; it is because they are widely available that they are supposed to be in widespread use. The user evaluation described in Chapter 7 concluded with an interview that asked the participants about their experience with macro recorders. Of the ten participants, five had never heard of macro recorders, two did not recall them until prompted, two did not think they were appropriate to the repetitive tasks they were attempting, and only one (who had professional experience with their use) considered recording a macro. These results suggest that though they may be widespread, the interface is too difficult for many users to access, learn, and use.

3.2 PBD for the end user

Good user interface design starts with the user's needs and abilities, and creates interfaces to support them (Nielsen, 1993; Hix and Hartson, 1993; Preece, 1994).

Many principles and guidelines can be applied to the design of user interfaces, but these can be difficult to apply to PBD, and particularly to application-independent systems. Systems that interact with existing applications can only apply basic principles like consistency, learnability, and efficiency where the applications permit it. Further, because they manipulate other applications, PBD systems do not fit well in the standard object-action metaphor, suggesting instead the agent conceptual model (Erickson, 1997). Agent design guidelines are appropriate, but still evolving, and few agent interfaces have been subjected to controlled testing (Shneiderman, 1997).

This section surveys the end user's motivations, abilities, and attitudes, and discusses how they will affect the uptake and use of PBD systems.

3.2.1 End user motivations

The end user's motivation for using PBD is to write programs that solve problems as they are encountered. This thesis considers iterative tasks and their solution with PBD; this is the end user's motivation for the purposes of the design.

Appendix A contains a set of iterative tasks from a variety of sources. The immediate aim of this design is to be able to automate these tasks, and others like them. The simplest tasks occur in single applications and involve little complexity. Others require detailed domain knowledge, and some require intuition so cannot be fully automated by a computer. A PBD system must therefore be capable of automating parts of tasks.

Several of the example tasks take place in more than one application. PBD systems to automate these must be application-independent and available on a system-wide basis. Past PBD systems have exhibited four levels of application independence, ranging from those that are entirely application-specific (e.g. predictive calculator, Metamouse), to those that can in theory be applied to new domains (e.g. CIMA, Eager), to those that are application independent but rely on collaboration from the application developer (e.g. AIDE, Topaz), to those that have practical application independence and work with any existing application

(e.g. macro recorders, Triggers). An ideal PBD system will fall into the last category, and work with any application.

3.2.2 End user abilities

The discussion in Section 2.1 suggests that end users be defined in terms of their computing goals—the use they make of their computer—not their computing skills. While this definition clearly identifies the range of people this research aims to help, it is less useful in assessing the end user's influence on the design of a PBD system.

In this design we make two assumptions about the end user's abilities: they are skilled computer operators in their area of expertise, but are not able to write computer programs.

End users are assumed to be skilled computer operators because they are people who use applications (Section 2.1) and their motivation is to solve complex iterative tasks (Section 3.2.1). Nardi (1993) suggests that end users are primarily interested in using computers as tools to accomplish these tasks, and are not interested in the computer as an object. Thus they are unlikely to learn new skills unless they perceive a significant benefit, and are unlikely to able to program their computers. This conclusion is supported by estimates that as few as 1% of computer users are programmers (Smith *et al.*, 1994).

3.2.3 End user attitudes

The user's attitude to programming will affect the way in which they use a programming environment. This section identifies subjective views that end users hold that might adversely affect their use of PBD, and how the design can minimise their influence.

Reluctance to program

End users are often reluctant to learn programming, but will learn a new tool if it can be done quickly and they perceive an immediate benefit. PBD systems that describe their function as "programming" may fall victim to the end user preconception that programming is hard. Instead, PBD systems can be described as *tools* or *programs*, with an emphasis on the benefits they provide. For example, the Familiar tutorial (reproduced in Appendix C) describes the PBD system as "a program that helps you perform repetitive tasks on the Macintosh computer."

It never uses *program* to describe the user's actions or the AppleScript they generate. Pilot studies showed that some users had difficulty with certain other terms including "iteration", which was replaced in the tutorial and interface with "cycle". The Metamouse instructions introduce the system through "Basil the turtle", an anthropomorphic interface agent (Maulsby *et al.*, 1989b). Metamouse's capabilities are explained in terms of teaching Basil, who has the ability to draw lines and "carry" boxes. The user is primed to use its constraint system by Basil's description of itself: "I have a good memory but I don't see to well. Instead I work mainly by feel." (Maulsby *et al.*, 1989b, p. 132).

Familiar and Metamouse both present the system as a tool that is (or appears to be) both immediately useful and easy to learn, and then lets the user learn advanced features incrementally. These are among the features that contribute to the success of spreadsheet programming languages (Nardi, 1993).

End users are risk-averse

End-user programmers are often risk-averse. The Turvy experiment found that most users liked the demonstrational interface; but all had concerns about completeness, correctness, and autonomy; and none would leave it unsupervised (Maulsby, 1994). End-user uptake of a tool will depend on the likelihood of success and the cost of failure, measured in terms of damage to data and the opportunity cost of programming (i.e. the time the user could have spent exploring other solutions). A system should fail gracefully, and if it is unable to complete the user's task it should at least do no harm. Error correction mechanisms like undo ensure that changes made by an agent cause no lasting harm. Error anticipation mechanisms, such as step-by-step execution, allow the user to debug a program before committing to its use. The opportunity cost of programming is a less quantifiable risk, but PBD systems have a low opportunity cost. Demonstrations are performed directly on the task data, so even if the system fails, part of the task is completed manually (Potter, 1993b).

The agent metaphor

The agent metaphor differentiates PBD systems from the conventional applications that they control: agents have specific characters; they notice things, do things, know things, and go places (Erickson, 1997). Users have high expectations of software that mimics human abilities, and may feel they have lost control if an agent takes the initiative.

Users have high expectations of agents, which lead them to attempt complex tasks with simple tools, and to expect specific reasons for agent actions (Erickson, 1997). To prevent the user from attempting the impossible, and to convey what the system is able to do, an agent's abilities should be self-evident and its actions explicable. Anthropomorphic interfaces should be used with caution (Shneiderman, 1997).

Ensuring that the user feels in control is essential in any interface, particularly in one that acts autonomously. Users who feel they have lost control will not trust the agent, and will not use it. Maulsby (1994) suggests three areas that users should be able to control—task execution, learned data descriptions, and learning methods—and lists a number of functional requirements. The most important include undo mechanisms, seeking user consent, minimising error, deferring to the user, and providing feedback. Ensuring that the user initiate agent action, allowing flexible execution, and motivating the user to exploit the agent will also contribute to the feeling of control.

3.2.4 Design guidelines

Table 3.1 lists a set of guidelines for the design of PBD systems that reflect the end user's combination of motivations, skills, and attitudes. The end user's experience with existing applications suggests that they be used (Guideline G1). Their inability to program in the conventional sense introduces several difficulties as the PBD system attempts to communicate with the user about the task and applications. The remaining three guidelines are strategies for overcoming these problems: simplicity (Guideline G2), minimising user effort (Guideline G3), and educating the end user (Guideline G4).

These guidelines are both minimal and high-level. They are not intended as a complete prescription for the design of practical systems, but to summarise the end user's influence on the design processes that culminated in the interface described in Section 3.3. The remainder of this section discusses each of the guidelines in detail and discusses how they can be applied.

Guideline 1: Use existing applications

End users are unlikely to give up the environments and applications they know for new, inferior, research prototypes. Even if a new product were as polished as an existing equivalent, it is unrealistic to expect end users to abandon the applications they are familiar with and learn new ones for the sake of unproven

G1	Use existing applications
G2	Simplicity
G3	Minimise user effort
G4	Educate the end user

Table 3.1 Guidelines for designing end-user PBD.

tools. Instead, PBD must be added to existing applications if it is ever to be useful—or even evaluated—in anything other than toy examples.

Guideline 2: Simplicity

End-user programming should be simple enough that the end user can use it without learning to program, or even knowing they are programming—the typical spreadsheet user does not equate writing formulas, an everyday task, with programming, an arcane discipline they feel is beyond their abilities.

Three areas of particular interest are accessibility, consistency, and program representation.

Accessibility. The first step towards using a system is to access it. If it cannot be found it will not be used. PBD systems have two access points: program creation and program execution (or invocation). Access to program creation must be consistent across applications. Access to the generated programs—in other words, program invocation—is trivial for iterative tasks because they are invoked while the program is created, and not independently executed.

Consistency or specialisation? PBD systems face a trade-off between specialisation and consistency. Specialised techniques can exploit knowledge of the application to solve application-specific tasks and use pertinent representations for programs. Consistency helps the user transfer skills from one application to the next, allows tasks to span applications, and prevents duplication of effort. Obviously, a single, consistent user interface will be easier for the user to learn. Further, task- and application-specific features increase the expense and decrease the independence of system-wide tools.

Program representation. Successful end-user programming systems work at a high level, concealing details from the user. The form of the high-level representation varies greatly, from artificial programming language syntax to polished visual depictions. When PBD is added to existing environments, however, the system may have little choice but to continue the design decisions made by application programmers.

Any language for describing interface actions necessarily introduces a layer of abstraction. However, PBD has three features that, when exploited, can make programs easier to comprehend: they represent commands the user has given, so the user knows what a program does; the user sees them as descriptions, not programs, so is more comfortable with them; and they contain no variables, instead using real data as placeholders for abstractions.

Many PBD systems use visual languages to represent programs. Nardi (1993) argues that there is little support for the assumption that visual notations are more "natural" than text-based alternatives, and cites studies that cast doubt on the assertion that they are easier to comprehend. Modugno *et al.* (1996) compared two visual languages for PBD, and found that users understood both. Thimbleby *et al.* (1992) analyse natural-language syntax; their work is discussed in the context of PBD in Section 6.5.1. Ultimately, the feedback language should minimise the distance between the application interface and the program representation (Smith *et al.*, 1994). Most modern applications have graphical user interfaces, so languages capable of graphical representations are likely to be preferable to any alternatives.

Guideline 3: Minimise user effort

Two barriers that Potter identified to just-in-time programming are the effort of entering and invoking a program (Potter, 1993b). Users with little experience programming and little time to spend will not use a PBD system if it requires a lot of effort. Minimising the user's effort is therefore imperative.

User effort can be reduced by minimising administrative commands, generating programs, and tolerating noise.

Start recording, stop recording. Although the user demonstrates a program in the application interface, they must still give the system additional commands to specify what and when they are demonstrating. Three approaches are commonly used: the user signals both the start and end of the demonstration, the user signals the start of the demonstration and the system infers the end, or the users actions are continuously recorded and the user retroactively signals when a demonstration has taken (or is taking) place.

If the user is giving a single example, they must signal its beginning and end (e.g. macro recorders, Halbert, 1993). This becomes unwieldy when several demonstrations are given because two additional commands per demonstration

are required, each of which is another opportunity for error. In iterative tasks, where many examples are given, the user signals the beginning of a demonstration, then performs the task until it is learned and the agent able to take over (e.g. Witten and Mo; 1993, Maulsby, 1994). This requires less effort because the user does not need to signal the boundaries of each iteration, and can give several examples in a single demonstration. In Maulsby's (1994) design the agent then conducts a dialog with the user. In contrast, Cypher (1993b) advocates minimal intrusion, which requires less effort of the user and gives them the option of ignoring the system, but may learn more slowly, necessitating extra demonstrations. An extension to this approach is to continuously record the user's actions and let them signal that they have already completed some or all of a task (e.g. Masui and Nakayama, 1994; Myers, 1998). Some systems take this idea still further by monitoring the user's actions and initiating interaction when they detect repetition (e.g. Cypher, 1993b; Ruvini and Dony, 1999).

There are strong arguments for allowing the user (or system) to retroactively identify repetition. Additional commands are minimised and the task need not be premeditated. Often users will not realise that they are starting an iterative task until they have performed the first few iterations. Cockburn and Bryant (1997), for example, observed their users (children learning programming) attempting to demonstrate a procedure "before initiating the loop or starting the procedure recorder". An overriding constraint on any agent that continuously monitors the user's actions is that it must be unobtrusive.

Program recognition or generation? It is easier for the user to recognise a correct program than to generate one, so the PBD system should infer as much of the program as possible from the user's actions. Existing systems vary in the amount of collaboration they require from the user to generate a program. Some attempt to infer the user's intention completely, while others create a framework and ask the user to supply details. As the system performs more inferencing on the user's behalf, it reduces the effort that the user must make to generate a program, but increases the likelihood of inferencing errors.

Even the best of systems will make mistakes, and users will need to give feedback about inferencing, either directly (e.g. changing an inference to a specific alternative) or indirectly (e.g. rejecting a prediction, providing a new example). Some users feel pressured when asked to explain features, but are comfortable responding to suggestions (Maulsby, 1994). Different situations and users may

prefer different techniques: new examples are easier to give when more data is available, and non-programmers may be able to recognise inferencing errors but be unable to articulate corrections.

Tolerate noise. End users may indulge in exploratory behaviour or simply make mistakes in the demonstration (Maulsby *et al.*, 1989a). As non-programmers, they are likely to misunderstand the abilities and limitations of a PBD system, or change the task during the demonstration. The PBD system should expect these inaccuracies, and treat them as normal: the user should not have to explicitly "fix up" a demonstration.

Guideline 4: Educate the end user

The end user is a non-programmer, but need not remain so. PBD systems expose the end users to programming language syntax, control structures, and the possibilities of programming. They should be encouraged to build on this knowledge. Some macro recorders record the user's actions in scripting language syntax and display them in a programming environment; end users who modify the recorded commands are beginning to become programmers. PBD systems can aid this progression by using existing languages.

3.3 The Familiar user interface

Having considered the implications of user-centric design for PBD systems, the Familiar user interface was created and tested. Only one of the guidelines places a technical constraint on the design: it should use existing applications (Guideline G1). This guideline led to the choice of the Macintosh as a platform, and the AppleScript language for communication with the user and applications. This decision is discussed in detail in Section 6.5.1; the limitations and tradeoffs it entails are described in Section 3.4.

Familiar learns from examples, and is best explained in the same way. By examining a selection of iterative tasks, this section demonstrates how it is used, the extent of its abilities, and what makes it unique.

The first example, rearranging a set of files into a horizontal row, introduces the interface and demonstrates the ability to iterate over sets and extrapolate sequences. In a variation of this task, the user asks Familiar to explain its predictions and gives feedback about their accuracy. The second example is to

Figure 3.1 The Familiar menu.

sort a set of files into two folders. Before it can safely be asked to finish the task Familiar must learn the sort criteria and convince the user it has learned them. The third is to convert a set of files from one image format to another. It demonstrates the ability to work across multiple domains and infer long cycles from noisy demonstrations.

In all cases the user first asks the agent to start observing their actions by selecting *Begin Recording* from the *Familiar* menu (Figure 3.1), which is available in every application. They then proceed to demonstrate the task in the standard user interface of the appropriate application.

3.3.1 Arranging files

In the first task, the user positions a set of files into a horizontal row in their folder window. Figure 3.2 shows the user's screen as they choose *Begin recording* from the *Familiar* menu. The menu appears in the Finder menubar next to the *Help* menu, just as it does in every application. The files that the user will rearrange are visible in the window on the left-hand side of the screen. One of the Familiar windows is visible in the background halfway down the right-hand side, indicating that Familiar is already loaded (usually, Familiar is automatically loaded when the user selects *Begin recording*). The Familiar windows cover only a small part of the total screen area.

The user begins the first task by moving a convenient file, *plum*, to the top left corner of the folder window (Figure 3.3a). These actions are recorded by Familiar and displayed in the *Familiar history* window (Figure 3.3b). The *activate* command (event 1) indicates that the user is working in the Finder. The *select* (event 2) and *set* (event 3) commands describe the positioning of the first file. The user

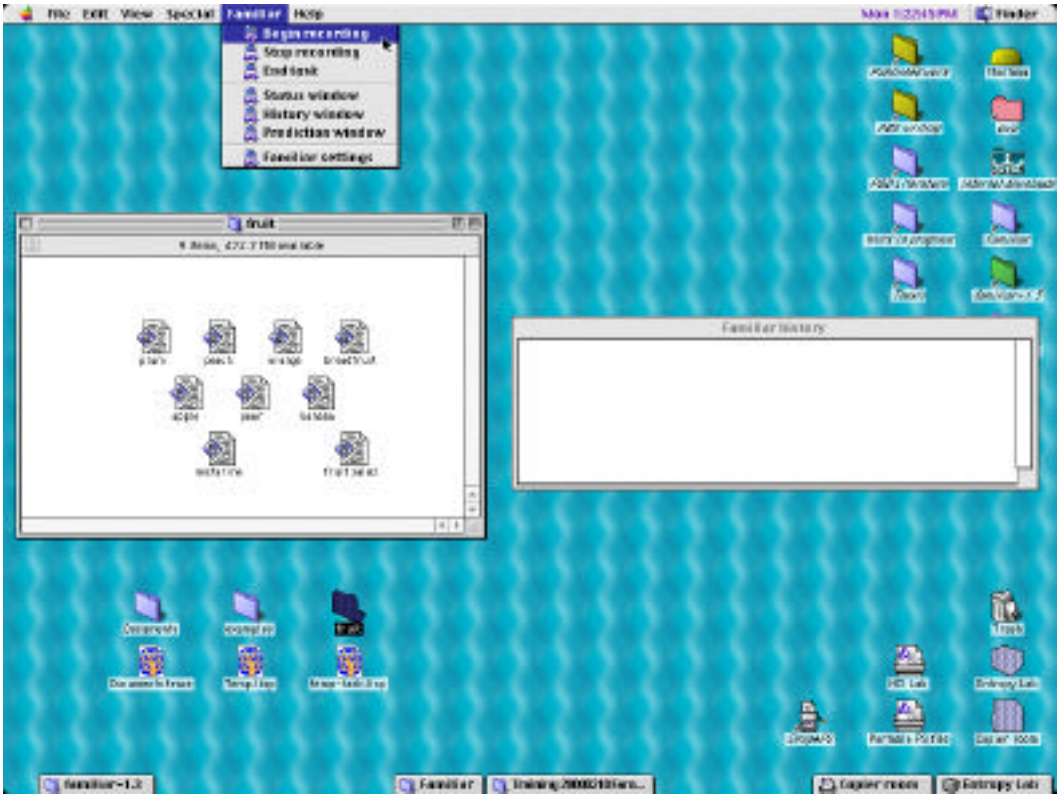


Figure 3.2 The screen before the arranging files task.

continues the demonstration by moving file *peach* (Figure 3.3c); again their actions are recorded and displayed (Figure 3.3b, events 4 and 5).

Each time it records a user action, Familiar attempts to generalise the event trace and infer the user's intent. After event 5 it detects a cycle and predicts the next iteration in the *Familiar predictions* window (Figure 3.3d). In this case it suggests that the next actions will be to *select file "apple"* (event 6) and *set* its position (event 7). The user is satisfied with this prediction because the task involves arranging all the files in a row, irrespective of order, and *apple* has yet to be moved.

Figure 3.4 shows the entire screen as it appears after the user has demonstrated the first two examples. Familiar is a stand-alone application, so its windows remain in the background until the user selects one, bringing it to the foreground.

The *predictions* window can be used to perform the task. The simplest interaction uses the *one time* (1x), *two times*, *five times* and *ten times* buttons, which execute the corresponding number of complete iterations of the cycle (Figure 3.5a). The user presses *one time* to tell Familiar to execute its predictions for events 6 and 7, and it responds by sending the commands to the Finder, which selects and

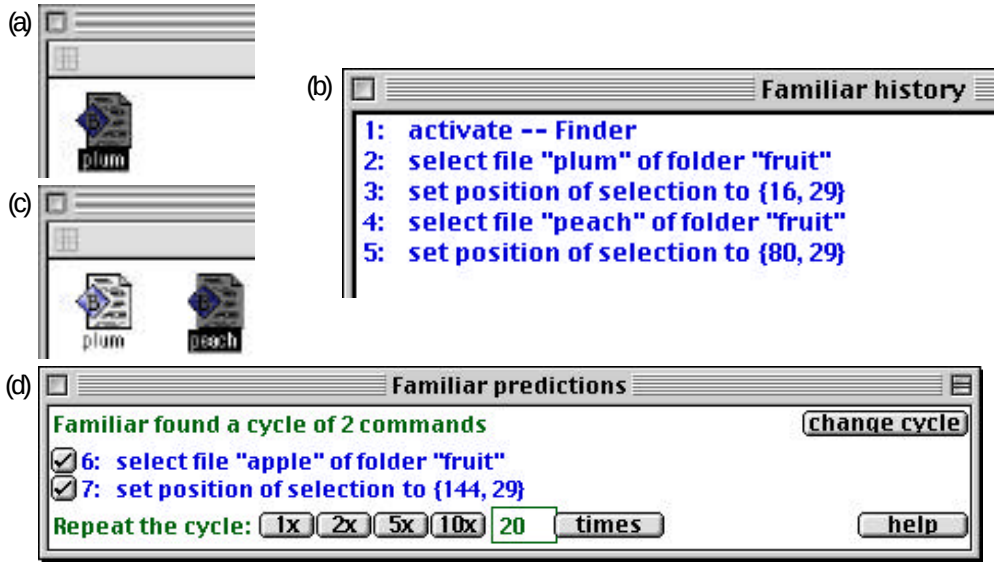


Figure 3.3 Using Familiar to arrange files. The user's two demonstrations (a,c) are recorded (b) and Familiar correctly predicts the next iteration (d).

positions file *apple*. The user follows the agent's progress by observing its actions in the Finder and watching the Familiar interface. As each command is executed it is added to the *history* window and its color is changed in the *prediction* window.[1] When the entire iteration has been executed, the prediction for the next iteration appears in the *predictions* window (Figure 3.5b). The user can instruct Familiar to execute two, five, or ten iterations of the task by pressing the appropriate button, or an arbitrary number by entering it from the keyboard. In this example the user, knowing how many files are left to position, types 6 (replacing the default value of 20 visible in Figure 3.5b) and presses *times*. After each iteration the *predictions* window is redrawn and the number of cycles remaining decremented. When six iterations are finished the task is complete (Figure 3.5c).

3.3.2 Arranging files when Familiar makes an error

The *predictions* window describes Familiar's predictions and accepts feedback about them. The simplest way to correct a mistake is to demonstrate another example with the standard application interface. The new demonstration—and the fact that the old predictions were incorrect—will be incorporated into

[1] Predictions are initially displayed in blue text, but are changed to red as they are executed.

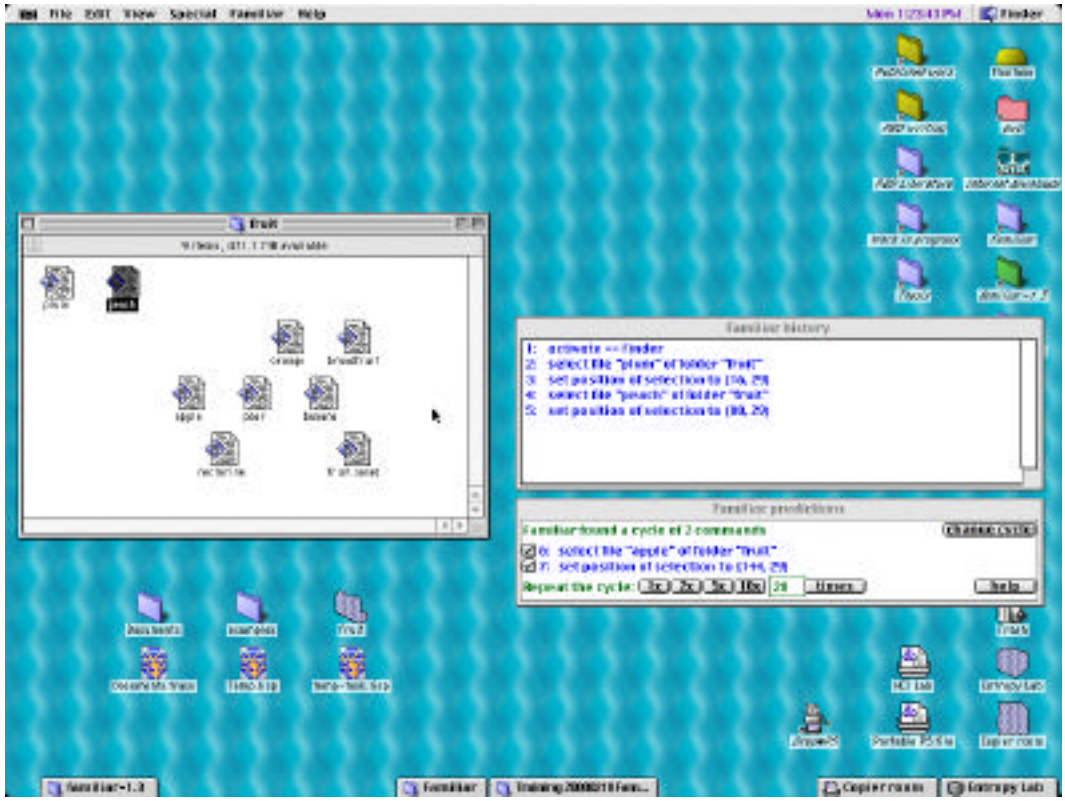


Figure 3.4 The screen after two demonstrations of the arranging files task.

subsequent predictions. This behaviour complements Familiar's ability to execute part of a cycle. If the user clicks on the *tick* button beside any command in the predictions window, the steps up to this point are executed. For example, if a cycle of six commands is predicted but only the first four are correct, the user can click on the fourth and then demonstrate the remaining two. Familiar will incorporate all six events into its subsequent predictions.

The *change cycle* button is used to reject the current iterative pattern and display another. Suppose, for example, that the user demonstrated the first two examples of the task (Figure 3.3a–c), but Familiar incorrectly reasoned that the *activate* command is important and predicted that each cycle should be composed of three events, *activate*, *set*, and *select* (Figure 3.6a). The user presses the *change cycle* button, and Familiar replaces the three-event cycle with a two-event cycle (Figure 3.6b).

The *help* button gives feedback explaining how predictions are made. The iterative pattern in Figure 3.7a is consistent with the two demonstrations of the task (Figure 3.3a–c), but the parameter of the *select* command (event 6) has not been extrapolated correctly: Familiar has predicted that the user will select *peach*,

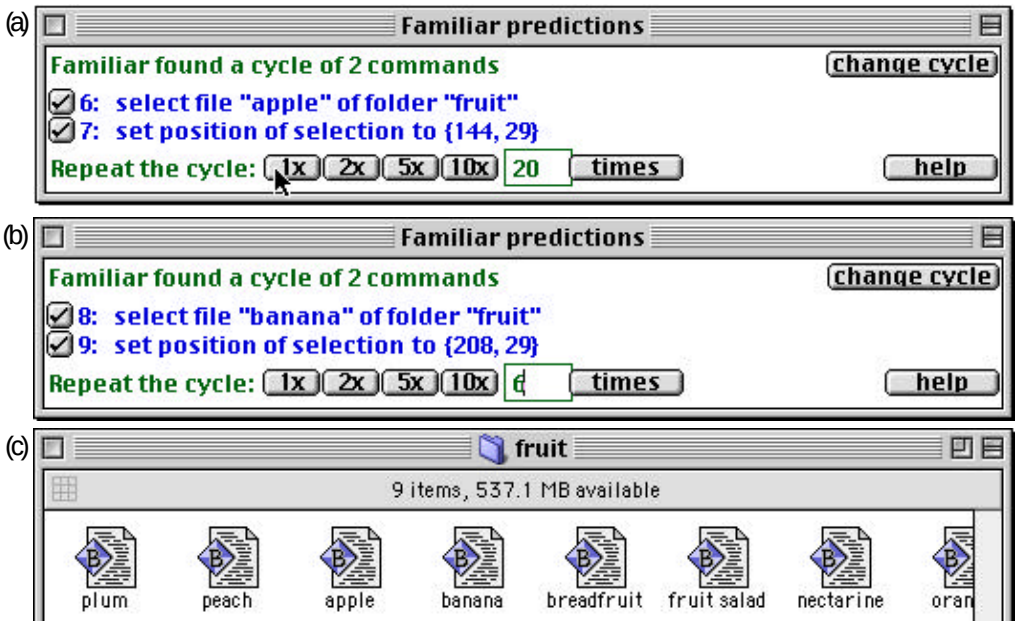


Figure 3.5 Completing an iterative task with Familiar.

but the user already moved this file (events 4 and 5) and wants to move a new one. To find out why the agent has made the erroneous prediction, the user clicks on *help*. The Macintosh *Balloon Help* feature is activated (Figure 3.7a) and used to explain predictions (Figure 3.7b,c). The user is concerned that the *select* parameter is incorrect (Figure 3.7b) and a balloon explains that Familiar has reasoned that the user is selecting the same *file* in every iteration. The prediction can be changed by option-clicking it, whereupon Familiar replaces *peach* with *apple* (Figure 3.7c). The new balloon explains that this prediction is made by assuming that the user is iterating over all the *file* objects in folder *fruit*. The agent's reasoning—and thus its prediction—is correct, and the task can now be completed.

3.3.3 Sorting files

The second task is to sort a set of files into folders for word-processor and spreadsheet documents. The user selects Begin Recording (Figure 3.1), and starts by creating a new folder and renaming it *word processor*. These commands are recorded and displayed (Figure 3.8a, events 1–4) but do not contribute to any iteration because they are once-only initialisation steps. The user then moves *ACC01.doc*, a word-processor document, into the new folder (Figure 3.8a, events 5,6). To demonstrate the second iteration, the user moves *ACC99.doc* into the word-processor folder (Figure 3.8b, events 7,8). The third file, *Balance sheet*, is a spreadsheet so the user creates a folder called *spreadsheets* (Figure 3.8c, events

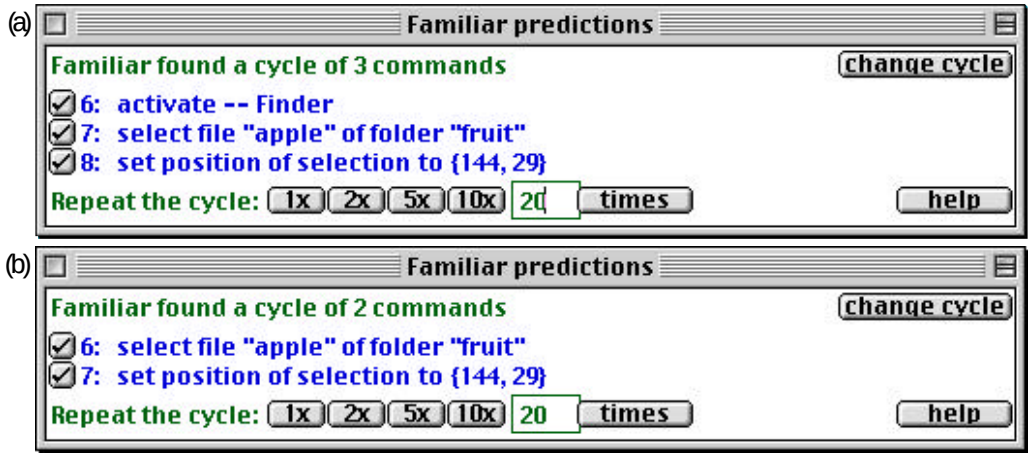


Figure 3.6 Changing an incorrect cycle.

9–11) and moves the file into it (Figure 3.8c, events 12,13). Three iterations of the task have been demonstrated, but over half of the recorded events are initialisation commands that do not contribute to the iterations.

Familiar detects an iterative pattern of seven events in the demonstration, and makes a corresponding prediction (Figure 3.8d). Unfortunately it is completely wrong, and the entire cycle it is predicting is incorrect—each iteration mimics the last seven events. The user rejects the pattern with the *change cycle* button. Familiar then suggests a two-step pattern that is correct for the next file (Figure 3.8e). The user presses the *one time* button and watches Familiar executing the commands to move file *Balance Sheet, 1996* to the *spreadsheets* folder.

When Familiar displays its prediction for the next iteration (Figure 3.8f) it becomes apparent that there is more teaching to do, for it predicts that the user will select *Corrections*, a word processor file, and move it into the *spreadsheets* folder (Figure 3.8f, event 17). Noticing the error, the user clicks on *help* and moves the mouse over the *move* command's *to* parameter. The ensuing help balloon (Figure 3.9a) explains that the prediction is the constant value *spreadsheets* (the value in the last two iterations), and that the user can change this by giving a new example. Familiar gives this advice because it has no other suggestions to make: three examples are insufficient to teach this classification task.

The user returns to the Finder and moves *Corrections* into the *word processor* folder. These actions are recorded (Figure 3.9b) and used to make a prediction for the next file (Figure 3.9c). Unfortunately, the prediction is incomplete: the agent correctly anticipates that the user will select expenses for, 1996, but fails to predict the destination folder, instead giving no current prediction. The user

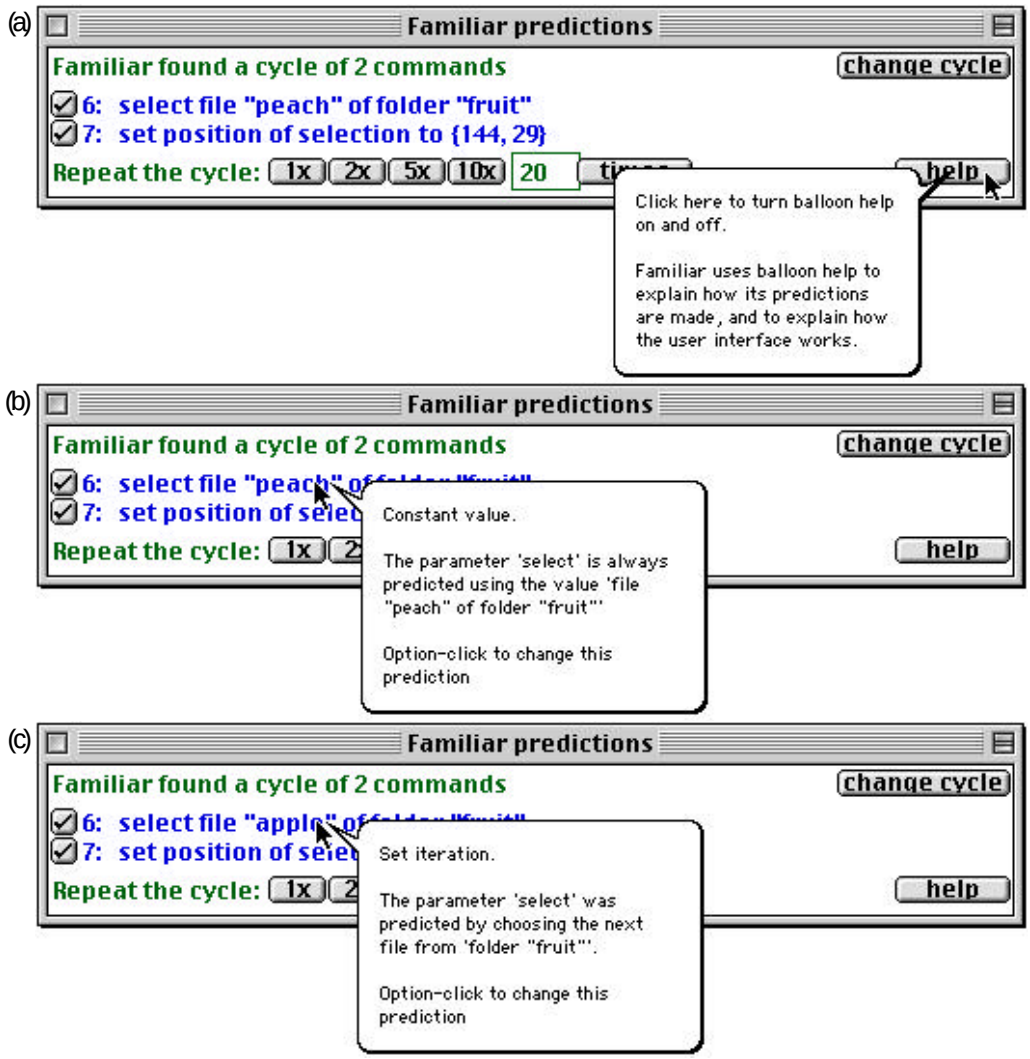


Figure 3.7 Examining and changing an incorrect prediction.

activates balloon help and asks for an explanation. Familiar has found a relationship between the to parameter and the kind attribute of the previous event, and will only make a prediction of event, 19 after event 18 has occurred. To test the prediction, the user clicks on the *tick* button beside event 18. Familiar executes it (Figure 3.9d), adds this event to the history window, and displays its prediction of the next two events (Figure 3.9e).

Familiar correctly anticipates that the next action will be to move the selected file into the *spreadsheets* folder. Confident that Familiar has grasped the idea, the user types 1000 into the number of iterations field and presses *times* (Figure 3.9e). After 135 iterations no files are left in the folder. Since Familiar can neither predict nor select the next file, it stops performing the task and awaits new

Figure 3.8 Changing an incorrect cycle.

instructions from the user, who chooses *Stop recording* from the *Familiar menu* and continues with their work.

3.3.4 Converting images

Complex tasks may involve multiple domains, longer demonstrations, and noise. Figure 3.10 shows a user automating part of the task Harvey delegated to his assistant in Section 1.3.1. In it, the user changes the format of a set of image files.

The subtask shown requires two applications and has a noisy event trace. The Familiar history window is shown after the first (Figure 3.10a) and second (Figure 3.10b) iterations have been demonstrated. The first three events of the

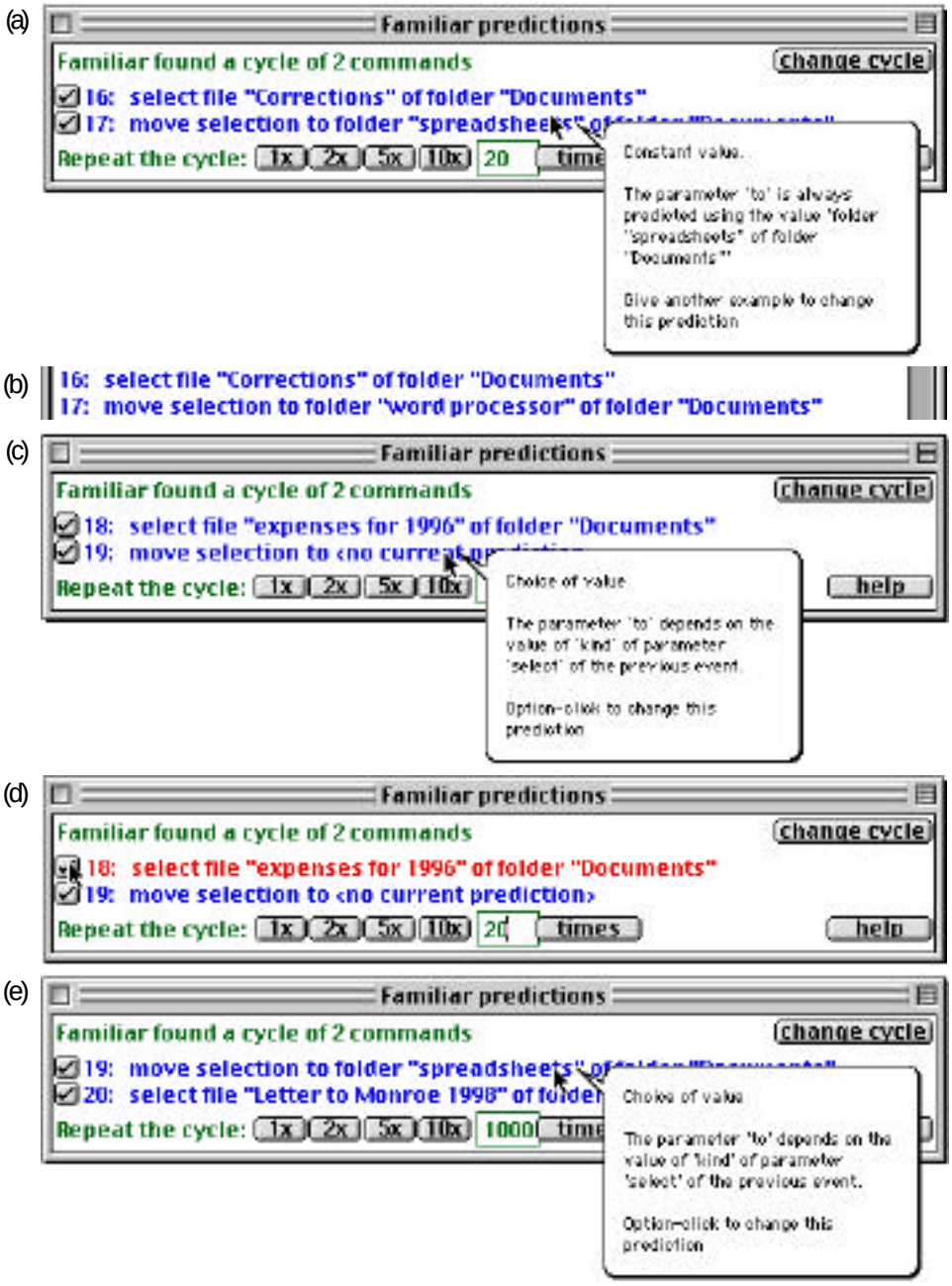


Figure 3.9 Changing an incorrect parameter.

first iteration initialise the environment and are not part of the iterative loop. Event 15 is a singular noise event generated when the user shifted a window to get a better view, and will not be repeated in future iterations. In Figure 3.10c we see that Familiar has correctly identified a cycle of six significant events and predicted the next full iteration.

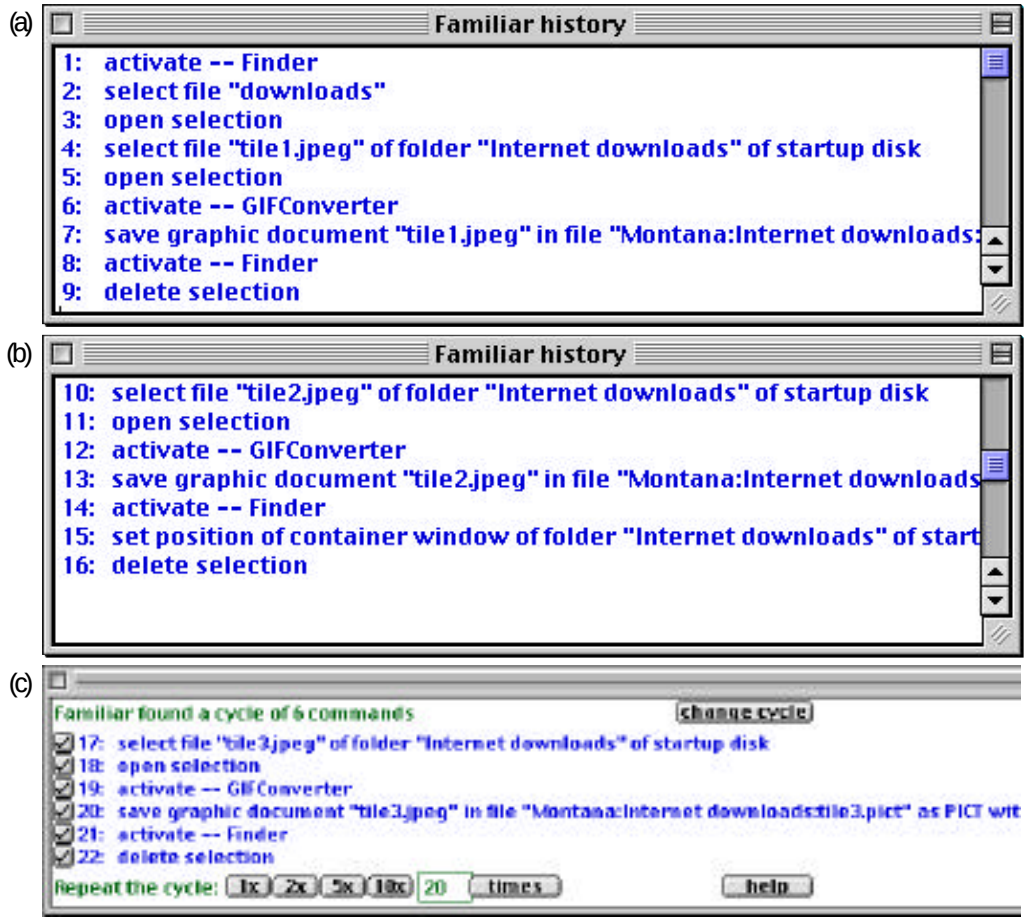


Figure 3.10 Converting image files.

3.3.5 Converting images with the evaluation interface

Figure 3.11 shows the same image conversion task being performed with an earlier version of Familiar, one that was used in the human evaluation described in Chapter 7. One reason for the evaluation was to gather feedback on the interface. The history and prediction windows in Figure 3.11 were generated from the actions of a participant in the evaluation, and were later reproduced to generate Figure 3.10.

The evaluation version of Familiar relied on the user to select an iterative pattern. In this example, the participant demonstrated the first two iterations and they were recorded in the history window iterations (Figure 3.11a,b, which are identical to Figure 3.10a,b). Familiar then predicted the user's next action, and suggested two alternatives (Figure 3.11c). The user confirmed that the first was correct by clicking on the *expand* button (labelled "e") next to it, causing Familiar

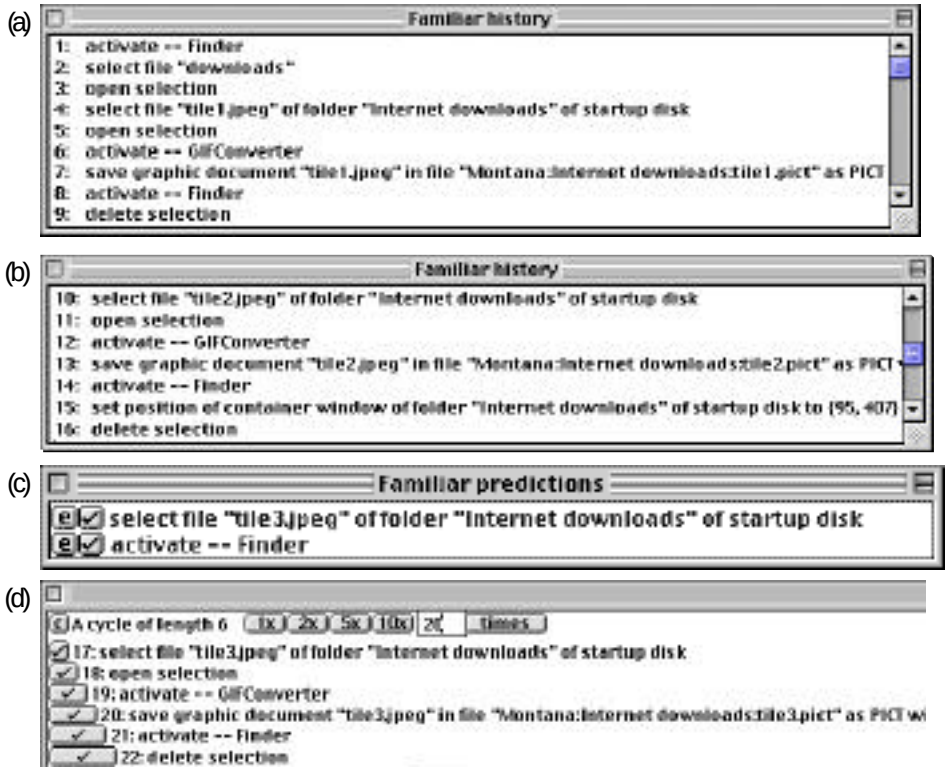


Figure 3.11 Converting image files using the evaluation interface.

to display all complete cycles that start with this action. The first of these is visible in Figure 3.11d (it is the same as Figure 3.10c).

The evaluation provided valuable feedback about the initial interface, which has been incorporated into the current version (Section 7.1.6). The underlying inferencing capabilities of the system are unchanged, but the user interface was substantially altered. The participants were confused by the use of two modes, one for choosing a cycle (Figure 3.11c) and another for executing it (Figure 3.11d). This style of interaction complicated the interface further when two cycles were displayed at once, and made it more difficult to change a cycle. The new version of the interface only ever shows complete cycles, and shows only one at a time, so is both simpler and requires less effort of the user. The prediction explanations and the ability to change the parameter predictions with an option-click action were also added to the evaluation interface (Figure 3.7). These features increase the control the user has over the system's inferencing, and can be used to correct inferencing errors. They can also reduce the number of demonstrations necessary to teach a task, and consequently the user effort required.

3.4 Familiar's limitations

In practice, the usefulness of PBD systems to end users is limited by their platform and the aims of their creators. Using a particular platform may mean that application data and functionality are inaccessible, that there is no way to work with the application's interface, that there is no convenient feedback language, or that other shortcomings impede PBD (Chapter 6). This necessitates trade-offs: to give the user one of the features they require, another may have to be sacrificed. The goals of a system's creator will also affect its implementation. Familiar has goals that occasionally conflict with the user's needs, including evaluating the Macintosh platform, eliminating domain knowledge, exploring the use of machine learning, and creating an adaptive interface.

Using existing applications

One major factor limiting Familiar is its platform, and its reliance on AppleScript to monitor, control, and examine other applications. The platform which was chosen on the basis of the first guideline: it enabled the system to work with existing applications. Familiar's ability to comply with the other guidelines is affected by this choice.

Familiar gathers domain knowledge directly from recordable applications at run-time, and needs no prior knowledge of their domain or implementation. Consequently it is domain-independent, and works with new, unseen applications as soon as they are installed. There are problems with the platform: the number of suitable applications is limited, some are poorly designed, and AppleScript has no support for undo. These and other limitations of AppleScript are discussed in detail in Section 6.5.

Simplicity

Though it is accessible and consistent, the interface would arguably be simpler if it used a visual language that represented objects as they appeared in applications. Instead, it uses the English-like text of AppleScript commands so that it is fully domain independent and because implementing a visual language in existing applications is technically very difficult (Section 6.4 discusses this problem). Some applications have poor AppleScript implementations, and Familiar could be programmed to clean up or augment their syntax. These improvements were not made because they would compromise the goals of domain-independence and educating the user.

Minimising user effort

Familiar successfully detects noisy patterns and extrapolates them. The user evaluation placed a limit on the tasks it could learn because the inferencing system was not developed far beyond the ability to automate tasks like those in the evaluation. This is in fact a broad range of tasks, including all the tasks in this chapter, but omits some tasks. The task evaluation in Chapter 7 explores the implementation's ability to automate iterative tasks.

Familiar is currently unable to infer termination conditions. This occasionally causes the unnecessary effort as they try to calculate how many iterations the interface should execute, or automate the task by repeatedly requesting a modest number of iterations. A practical system for inferring termination conditions is described in Section 8.4.1.

The discussion of the user effort guidelines recommended that an agent record all the users actions and allow them to retroactively identify a demonstration. In practice, this advice is ignored because AppleScript recording is sometimes obtrusive: it affects program behaviour.

Educating the end user

The user interface exposes users to the AppleScript scripting language so that they become accustomed to its syntax, allowing the end user to slowly, perhaps unknowingly, acquire programming skills. In many cases, the original poorly formed AppleScript commands are displayed, rather than "corrected" versions that are easier for the user to comprehend, so that the user is not misled about the language.

3.5 Summary

This chapter has considered the end user's motivations, skills, and attitudes as a basis for the design of a PBD system. It advocates a application-independent approach using existing applications and environments, simplicity, minimising the user's role in program generation, and attempting to educate the end user. These principles are followed to design Familiar, a PBD system for automating iterative tasks, though some compromises are made to technical restrictions and to satisfy other requirements of the research.

4 The Familiar architecture

This thesis argues that a system-wide, application-independent PBD interface for performing iterative tasks in existing applications can be made available to end-users. Familiar is an agent that provides this functionality on the Apple Macintosh computer. This chapter explains Familiar's inferencing and its interaction with applications. The user interface is described in Chapter 3, and the machine learning components in Chapter 5.

Users employ Familiar to automate iterative tasks. They start by signalling that a task is about to begin, causing the agent to monitor their actions. The user begins performing the task, and Familiar identifies patterns in the event trace and makes predictions. The user does not have to signal when one iteration ends and the next begins; they need give no other instruction than the demonstration. Familiar usually detects the repetition when the user demonstrates the second iteration of the task, and displays its predictions in a special window. Users can, at their convenience, execute predictions, reject and change them, or request an explanation of Familiar's reasoning. Alternatively the user can ignore Familiar and continue the task unaided; Familiar will not interfere.

Familiar learns tasks that change from one iteration to the next by generalising and instantiating data descriptions. From the user's perspective, Familiar is seen to learn a task by making correct predictions, and by changing incorrect predictions in response to new examples and feedback. The inferencing system has to satisfy a number of constraints. Like any system, it must learn accurately, from few examples, and incorporate feedback. Familiar is application independent, and exploits domain knowledge without introducing a bias toward a particular domain. The interface minimises user interaction, so the inferencing system must infer when one iteration ends and the next begins. It must be able to explain every prediction. Finally, Familiar interacts with the AppleScript language, so the inferencing engine must work with high-level events.

The high-level nature of AppleScript lends some structure to the learning problem. The Familiar implementation works with AppleScript commands, but is applicable to any high-level event model.

Familiar enjoys a significantly greater level of domain independence than any other inferencing PBD system because of its ability to treat all application knowledge as data, including both class knowledge and instance knowledge. Class knowledge describes the capabilities of each application, including the commands it responds to and the types of data it uses. Familiar regenerates the class information for an application the first time it is used in each new session, and because it can generate class information on the fly, it is able to work with previously unseen applications. Instance knowledge describes the actions the user is performing in the application, and the current state of the application. Instance data cannot be interpreted without class knowledge. The task of the inferencing system is to predict future instances of user commands.

Familiar's inferencing is based on the certainty that every event of the same type has the same set of parameters. It is performed on two levels. First, Familiar searches for iterative patterns in the types of events the user has demonstrated. Currently, two sequence recognition schemes, called *SRS-simple* and *SRS-noisy*, search for candidate patterns and make zero or more suggestions. Familiar selects the best candidate pattern and extrapolates it. Second, Familiar attempts to extrapolate each command in the next iteration of the best candidate pattern. Five competing pattern analysis schemes, *PAS-constant*, *PAS-extrapolation*, *PAS-previous*, *PAS-set*, and *PAS-ML*, are used to extrapolate future values of command parameter values; each makes zero or one candidate predictions. Familiar selects the best candidate prediction of each command parameter, and suggests it to the user.

Familiar exploits competing algorithms to make predictions. At both levels, it is difficult to decide which candidate is best because the predictions made by different schemes are not directly comparable. How reliable, for example, is a prediction made by *PAS-extrapolation* compared to one made by *PAS-ML*? This is an important question because a PBD system is only useful if it can be trusted to work autonomously, and that trust is based on the reliability of its predictions. Correct predictions will build a correct program that can be used to complete the user's task. Incorrect predictions build an incorrect program that will at best cost the user time, and at worst give erroneous commands to

applications, damaging the user's data and their confidence in the agent. Familiar uses a set of rules to estimate the likelihood that a prediction will be acceptable to the user; this estimate is used to choose between candidate predictions. The derivation of these rules using machine learning is described in Chapter 5.

This chapter introduces Familiar. Section 4.1 gives an overview of Familiar's architecture, and subsequent sections describe the major components. The event recorder monitors the user's actions and maintains Familiar's application knowledge (Section 4.2). The sequence recognition manager searches the event trace for iterative patterns, and chooses the best candidate (Section 4.3). The pattern analysis manager takes this pattern and uses it to predict the next full iteration of the pattern (Section 4.4).

4.1 System overview

Familiar is a stand-alone application. It is implemented in Macintosh Common Lisp (Digitool, 1995–1999) and communicates with the AppleScript language (analysed in Section 5.6). The system-wide *Familiar* menu uses an independent menu enhancement utility to load compiled Lisp expressions (Widemann, 1992–1999). Familiar's requirements are the Macintosh Operating System (version 7.5 or later), AppleScript, the Macintosh Common Lisp interpreter, the AliasMenu utility, and recordable applications. It was developed on a 200Mhz Power Macintosh with 32MB RAM, but a faster machine is recommended for regular interactive use.

Figure 4.1 depicts Familiar's architecture. The major system components are represented by circles, and the Familiar interface by rectangles. The topmost oval represents the user's applications and environment, which are external to the PBD system. When the user performs an action in an application, the AppleScript recording mechanism reports it to Familiar's event recorder, causing a series of data flows that follow the solid grey arrows around the diagram and culminate at the Familiar prediction window. The prediction window can be used to send new commands to other applications.

The event recorder

The event recorder monitors the user and applications. It is activated when the user selects *Begin Recording* from the *Familiar* menu: AppleScript recording is initiated and every subsequent user action is reported to Familiar. The event

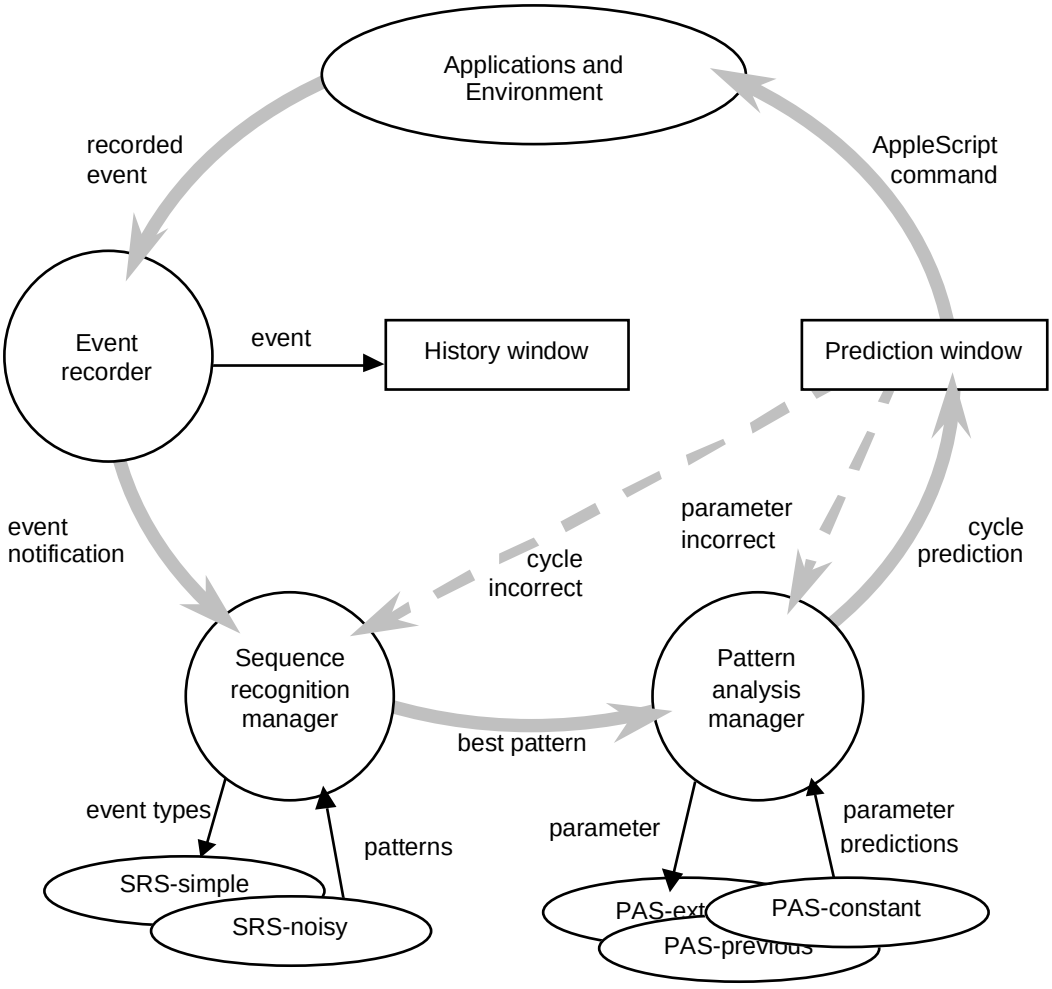


Figure 4.1 The Familiar architecture.

recorder parses each action, displays it in the *Familiar history* window, updates the event trace, gathers basic contextual information, and notifies the sequence recognition manager that a new event has occurred. The event recorder maintains a model of every application that it encounters in the event trace.

The sequence recognition manager

The sequence recognition manager detects and maintains patterns in the event trace. The first time the sequence recognition manager is notified of an event, it uses sequence recognition schemes (including *SRS-noisy* in Figure 4.1) to search the event trace for iterative patterns. If one (or more) of the schemes detects a pattern, the sequence recognition manager chooses the best pattern and sends it to the pattern analysis manager; this pattern is now the *current pattern*.

When the sequence recognition manager receives subsequent event notifications, two courses of action are available:

- If it has a current pattern, and the new event is consistent with that pattern, the current pattern is updated and sent to the pattern analysis manager.
- If it has no current pattern, or the new event is not consistent with the current pattern, it re-initialises itself and treats this event as if it were the first.

The pattern analysis manager

The pattern analysis manager predicts the next iteration (or cycle) of a task based on a given pattern. Each pattern is broken down into its commands, each command is broken down into its parameters, and the previous values of each parameter are calculated. The pattern analysis manager uses several analysis schemes (including *PAS-constant* and *PAS-extrapolation* in Figure 4.1) to predict the next value of the parameter, then selects the single best prediction of each parameter. (If the pattern has been analysed before, the parameter predictions are updated rather than recalculated.) The best parameter predictions are used to build command predictions, the command predictions are used to build a prediction of the next iteration of the cycle, and this prediction is sent to the prediction window.

The prediction window

The prediction window displays a cycle of predicted commands, and lets the user send these commands to application programs. When a command is sent to an application it is also sent to the event recorder, causing an event notification to be sent to the sequence recognition manager, which updates the current pattern and sends it to the pattern analysis manager, which updates the cycle prediction and passes it back to the prediction window, where the display is updated appropriately.

Feedback

The user can give Familiar explicit feedback through the prediction window, or implicit feedback by ignoring it. If the user ignores the prediction window and performs new commands, they are recorded by the event recorder, and processed as has been described. The *prediction window* lets the user reject parameter and cycle predictions (represented by dashed grey lines in Figure 4.1). When the user signals that a parameter prediction is incorrect, the pattern analysis manager is sent a notification, and finds the next best prediction of the parameter using the knowledge that its last choice was rejected. The cycle prediction is then recalculated, incorporating the new prediction, and passed to the prediction

window. When the user signals that an entire predicted cycle has the wrong form, the sequence recognition manager is notified and discards the current pattern, replacing it with the next best one from the event trace. This new pattern is sent to the pattern analysis manager and the prediction window.

4.2 The event recorder

The event recorder monitors the user and manages the information Familiar uses to make predictions. Knowledge of AppleScript is intrinsic to both functions. AppleScript is a scripting language for the Apple Macintosh. It is analysed as a platform for PBD in Section 5.6; this section describes its use in Familiar.

4.2.1 AppleScript

Macintosh applications are called *scriptable* if they support a set of AppleScript commands that can be invoked by scripts and other programs. Some (but not all) scriptable applications are also *recordable*: they can be asked to report the user's actions as they are made. When the user selects *Begin recording* (Figure 3.1), Familiar activates AppleScript recording, and any subsequent user action in a recordable application is reported to Familiar. Actions performed in an application that is not recordable are not reported and will not appear in the history window.

Figure 4.2 shows the commands recorded by Familiar as the user begins a demonstration. The *tell* command (line 1) identifies the application that has reported the command. Familiar recognises that the user is now working in an application named "Microsoft Excel". This is the first time in this session that it has encountered an application with this name, so it has no knowledge of the application. Familiar immediately builds a model of "Microsoft Excel" using the methods described in Section 4.2.2. This model is used to parse the actions reported by the application. Familiar does not display the *tell* command because it is designed to be read from the perspective of a user issuing the command, and is out of context in the history window. Instead, the application name is appended to the command that follows—invariably an *activate* command (line 2)—resulting in expressions like *activate -- Microsoft Excel* in the *history* window. In AppleScript syntax, any text following the token "--" is ignored as a comment, so *activate -- Microsoft Excel* and *activate* are equivalent, and valid, commands.

```
1  tell application "Microsoft Excel"
2      Activate
3      Select Range "R1C2"
4      set FormulaR1C1 of ActiveCell to "July"
5      Select Range "R1C3"
6      set FormulaR1C1 of ActiveCell to "August"
7      Select Range "R1C4"
```

Figure 4.2 AppleScript recorded in Microsoft Excel.

Familiar requires knowledge of AppleScript to parse the event trace. The language itself has a core functionality, including the *tell* command, that Familiar is given as background knowledge (Section 4.2.3), but every scriptable application extends the language by adding its own objects and commands, which appear frequently in the event trace.

Objects and commands differ from one application to the next, but have a consistent structure. AppleScript is superficially an object-oriented language, and applications maintain a hierarchy of object classes. Every class has a set of properties (e.g. *capacity* is a property of *disk*) and containee classes (e.g. *folders* can contain *files* and other *folders*); some also have superclasses and plural forms. Instances of classes (objects) are identified by a *reference*: their class name and a unique property (typically *name* or *index*) combined with their position in the containment hierarchy (indicated by the keyword *of*). Data descriptions in Familiar, like *file "apple" of folder "fruit" of application "Finder"* and *word 6 of document 1 of application "Scriptable Text Editor"* are references. Application commands are identified by name, and may also have a direct parameter and a set of named parameters. For example, an application implementing a command named *move* with a required direct parameter, a required parameter named *to*, and an optional parameter named *as* would respond to instructions like *move selection to cell "A2" as integer*. Parameters are typed, but often have very general types like *reference* or *anything*.

4.2.2 Application knowledge

Familiar models each application to infer intent and make predictions. Its application knowledge falls into two categories: *class* (or type) information, and *instance* information (or data). Class information describes the capabilities of the application, including the commands it supports and the classes it uses, information which remains unchanged from one invocation of the program to the next. Instance information describes the current state of the application, and

Application	Version	Number of commands	Number of classes	Number of enumerations
Finder	8.6	26	41	7
Fetch	3.0.3	14	14	4
GIFConverter	2.4d18	25	17	8
Microsoft Excel	98	172	95	223

Table 4.1 Application size measured by quantity of high-level commands, classes (ignoring plurals), and enumerations.

includes the event trace and contextual information about the user's actions. Instance information is constantly changing in response to user actions.

Class information

Familiar's primary source of class information is the application dictionary. This is a list of all the commands, objects, and enumerations used in the application. Table 4.1 lists the number of commands in some common applications. The Finder dictionary, for example, contains 26 commands, 41 classes, and 7 enumerations. These totals ignore "plural" classes (those that represent more than one of some other class).

Dictionary information is augmented by external application models. These are manually created by the developer, and though they are not necessary, they can make prediction faster and more accurate. Familiar currently has external models of the Finder (version 8.1 and 8.6) and Microsoft Excel (version 5.0) The complete models are reproduced in Figures 4.3 and 4.4. Each consists of a set of alias statements, inheritance statements, and load warnings.

Alias statements record alternative names for objects: for example, *startup disk* is commonly used by the Finder (version 8.1) to describe a specific disk, but does not appear in its dictionary (Figure 4.3, line 4). *Inheritance statements* make superclasses explicit: for example, the Finder (version 8.1) *file* class inherits from the *item* class (Figure 4.3, line 11), and the Excel *Application* class inherits the properties of a hidden *_Global* class (Figure 4.4, line 6). Inheritance relationships are frequently omitted from application dictionaries. Alias and inheritance statements repair omissions in application dictionaries that can impair Familiar's ability to gather contextual information, and consequently to infer intent.

```
 1 (let ((d (domain-named "Finder")))
 2
 3       ; Alias statements
 4       (model-alias d "disk" '("startup" "disk"))
 5       (model-alias d "desktop-object" '("desktop"))
 6       (model-alias d "trash-object" '("trash"))
 7
 8       ; Inheritance statements
 9       (model-inheritance d "item" "disk")
10       (model-inheritance d "item" "folder")
11       (model-inheritance d "item" "file")
12       (model-inheritance d "file" "document file")
13       (model-inheritance d "file" "alias file")
14       (model-inheritance d "file" "application file")
15       (model-inheritance d "file" "sound file")
16       (model-inheritance d "file" "desk accessory file")
17       (model-inheritance d "file" "font file")
18       (model-inheritance d "window" "container window")
19       (model-inheritance d "window" "information window")
20       (model-inheritance d "container" "disk")
21       (model-inheritance d "container" "folder")
22
23       ; Evaluation warnings
24       (model-load-warning d "item" "icon" "Property cannot be displayed")
26       (model-load-warning d "application" "about this computer" "do not evaluate")
25       (model-load-warning d "container" "entire contents"
                             "Takes too long to retrieve")
27       (model-load-warning d "item" "creation date obsolete"
                             "Mac OS 8 backward compatability hack")
28       (model-load-warning d "item" "folder obsolete"
                             "Mac OS 8 backward compatability hack")
29       (model-load-warning d "item" "modification date obsolete"
                             "Mac OS 8 backward compatability hack")
30       (model-load-warning d "item" "file type obsolete"
                             "Mac OS 8 backward compatability hack")
31       (model-load-warning d "item" "locked obsolete"
                             "Mac OS 8 backward compatability hack"))
```

Figure 4.3 Familiar's model of the Finder (version 8.1).

Load warnings are attached to the properties of application objects to indicate that Familiar should not include them in its contextual data. This is an efficiency measure: examples include binary data such as icons in the Finder (Figure 4.3, line 24), large objects like the *EntireRow* property in Microsoft Excel (Figure 4.4, line 11), and properties the developer has deprecated like those in the Finder (Figure 4.3, lines 27–31). Familiar is able to load this data, but it may take a long time and is unlikely to benefit the user.

Familiar gathers additional class information by inference from the event trace. It learns the order of command parameters by observing their use in recorded

```
1   (let ((d (domain-named "Microsoft Excel")))
2
3       ; Alias statements
4
5       ; Inheritance statements
6       (model-inheritance d "_Global" "Application")
7
8       ; Load warnings
9       (model-load-warning d "Range" "CurrentRegion" 'loads-very-slowly)
10      (model-load-warning d "Range" "EntireColumn" 'loads-very-slowly)
11      (model-load-warning d "Range" "EntireRow" 'loads-very-slowly)
12      (model-load-warning d "Range" "Style" 'parser-has-trouble-with-formatting))
```

Figure 4.4 The Familiar model of Microsoft Excel (version 5.0).

events—parameters are generally stored alphabetically in the dictionary, but should be displayed in the order that forms the best and simplest English sentences. Occasionally Familiar records a reference to a command, object, or property that is not described in the dictionary, usually because of an error by the developer. Familiar will attempt to parse the expression and deduce class information from the syntax of the instance (see Section 6.3.2).

Instance Information

The primary source of instance information is AppleScript recording. As commands are recorded and parsed, Familiar builds an event trace of the commands and objects it observes.

The event trace is augmented by contextual information. Familiar gathers context by sending AppleScript commands to applications requesting instance data. It gathers three types of context: the properties of objects, the evaluation of properties, and the contents of container objects. Figure 4.5a,c,e contains command templates that Familiar uses to gather each type of context. The explanations below cite examples from the Finder, but the templates can be used with any application, as is illustrated by additional examples from Microsoft Excel (Figure 4.5b, example 2), GIFConverter (Figure 4.5b, example 3; figure 4.5d example 2), and Fetch (Figure 4.5f, example 2).

Familiar retrieves the properties of every object that occurs in the event trace. For example, in event 2 of Figure 3.3 the command *select file "plum" of folder "fruit"* is recorded in the Finder. Familiar recognises that *plum* is a *file* object and uses the command template in Figure 4.5a to gather context. In this case, the *application name* is *Finder*, the *reference* is *file "plum" of folder "fruit "*, and the *list of property names* is found by examining the class data to find all the properties of a *file*

(a)
```
tell application <application name> to get <list of property names> of <reference>
```

(b)
```
tell application "Finder" to get { creator type, file type obsolete, file type, locked obsolete, locked, product
    version, stationery, version, bounds, comment, container, content space, creation date, creation
    date, description, disk, folder obsolete, folder, icon, id, information window, kind, label index,
    modification date obsolete, modification date, name, physical size, position, selected, size, window }
    of file "plum" of folder "fruit"

tell application "Microsoft Excel" to get { AddIndent, Address, AddressLocal, Column, ColumnWidth,
    CurrentRegion, EntireColumn, EntireRow, Font, Formula, FormulaArray, FormulaHidden,
    FormulaLocal, FormulaR1C1, FormulaR1C1Local, HasArray, HasFormula, Height,
    HorizontalAlignment, Interior, Left, Locked, Next, NumberFormat, NumberFormatLocal, Orientation,
    Parent, PrefixCharacter, Row, RowHeight, SoundNote, Style, Text, Top, UseStandardHeight,
    UseStandardWidth, Value, VerticalAlignment, Width, WrapText } of Range "R4C1"

tell application "GIFConverter" to get { closeable, titled, index, modal, resizable, zoomable, zoomed,
    name, selection, class } of window "Glacier National Park"
```

(c)
```
tell application <application name> to get <property reference>
```

(d)
```
tell application "Finder" to get selection of application "Finder"

tell application "GIFConverter" to get selection of graphic document "tile54.pict"
```

(e)
```
tell application <application name> to get every <class name> of <reference>
```

(f)
```
tell application "Finder" to get every file of folder "fruit"

tell application "Fetch 3.0.3" to get every remote file of application "Fetch 3.0.3"
```

Figure 4.5 Templates for AppleScript commands to (a) get properties, (c) evaluate a property, and (e) get containee objects, with (b,d,f) examples of each.

object that do not have load warnings. This information is used to formulate the first command in Figure 4.5b. The command is executed, and the Finder returns a list of objects that are added to the stock of instance data. If the application is unable to return any of the properties that Familiar requests, load warnings are automatically generated for those properties so that Familiar does not try to retrieve them again. Figure 4.5b contains three different commands for retrieving the properties of objects. All are generated from the template in Figure 4.5a, but all retrieve object-specific context from separate applications.

Familiar evaluates any object property filling the role of an object. The most common example is the *selection* property. In the Finder *selection* is a property of the application, but in event 3 of Figure 3.3 it is treated as an object: the command *set position of selection to {16,29}* contains a reference to the *position*

property of the *selection* property of the application. Familiar uses the command template in Figure 4.5c to evaluate the property and recover its actual value. In this example, the *application name* is *Finder* and the *property reference* is *selection of application "Finder"*, so the first command in Figure 4.5d is formulated. When it is executed the selection is returned as a reference, which Familiar uses to requests the properties of the selection using the template in Figure 4.5a. The second command in Figure 4.5d is an example of Familiar evaluating a property in the GIFConverter application.

Familiar sometimes retrieves objects contained by another object. If the user examines a *file* object in *folder "fruit"*, as occurs in Figure 3.3b, the command template in Figure 4.5e can be used to find all the *files* in this folder, resulting in the topmost command in Figure 4.5f. This information would be useful if the user was iterating over the set of files in the folder, a scenario which is discussed in more detail in the description of *PAS-set* in Section 4.4.3.

4.2.3 Background knowledge

Familiar contains two types of background knowledge: inherent and explicit. Familiar's suppression of the *tell* command (Section 4.2.1), and its use of the command templates in Figure 4.5, are implicit background knowledge. Explicit background knowledge is drawn from two sources. First, Familiar finds the *English Dialect* description of AppleScript when it is launched, and reads from it the basic commands, objects, and enumerations in the AppleScript language. Most of these commands are for program control and inter-application communication. Familiar builds straight-line programs with no variables, so it uses only the *tell* and *activate* commands. The objects and enumerations generally represent computer-related concepts, and are rarely used by applications. Second, Familiar has been given a set of "general knowledge" enumerations, shown in Figure 4.6. These include simple sets like the letters of the alphabet (Figure 4.6, lines 1–4), and the names of days and months (lines 5–28), and roman numerals (lines 29–32) in common formats. It is not clear how much background knowledge a PBD system requires, or at what point background knowledge becomes task specific. Familiar could easily be given more enumerations, like the number of days in each month or the Kings and Queens of England.

```
1  (make-enumeration *AppleScript-domain* "uppercase letters"
2                    '("A" "B" "C" … "Z"))
3  (make-enumeration *AppleScript-domain* "lowercase letters"
4                    '("a" "b" "c" … "z"))

5  (make-enumeration *AppleScript-domain* "Days 1"
6                    '("M" "T" "W" "T" "F" "S" "S"))
7  (make-enumeration *AppleScript-domain* "Days 2"
8                    '("m" "t" "w" "t" "f" "s" "s"))
9  (make-enumeration *AppleScript-domain* "Days 3"
10                   '("Mon" "Tues" "Wed" "Thu" "Fri" "Sat" "Sun"))
11 (make-enumeration *AppleScript-domain* "Days 4"
12                   '("mon" "tues" "wed" "thu" "fri" "sat" "sun"))
13 (make-enumeration *AppleScript-domain* "Days 5"
14                   '("Monday" "Tuesday" "Wednesday" "… "Sunday"))
15 (make-enumeration *AppleScript-domain* "Days 6"
16                   '("monday" "tuesday" "wednesday" … "sunday"))

17 (make-enumeration *AppleScript-domain* "month 1"
18                   '("J" "F" "M" "A" "M" "J" "J" "A" "S" "O" "N" "D"))
19 (make-enumeration *AppleScript-domain* "month 2"
20                   '("j" "f" "m" "a" "m" "j" "j" "a" "s" "o" "n" "d"))
21 (make-enumeration *AppleScript-domain* "month 3"
22                   '("Jan" "Feb" "Mar" … "Dec"))
23 (make-enumeration *AppleScript-domain* "month 4"
24                   '("jan" "feb" "mar" … "dec"))
25 (make-enumeration *AppleScript-domain* "month 5"
26                   '("January" "February" "March" … "December"))
27 (make-enumeration *AppleScript-domain* "month 6"
28                   '("january" "february" "march" … "december"))

29  (make-enumeration *AppleScript-domain* "Roman numerals (uppercase)"
30                   '("I" "II" "III" "IV" "V" "VI" "VII" "VIII" "IX" "X" "XI" "XII"))
31 (make-enumeration *AppleScript-domain* "Roman numerals (lowercase)"
32                   '("i" "ii" "iii" "iv" "v" "vi" "vii" "viii" "ix" "x" "xi" "xii"))
```

Figure 4.6 Familiar's background knowledge.

4.2.4 Application independence

Familiar is application independent: it maintains knowledge of each application independently of the others, and its inferencing is based only on this information and the background information. Unlike other PBD systems, Familiar does not store information about most applications, it simply rebuilds its models from the sources described above as each is observed in the event trace. In the future, Familiar will generate more complex application models to preserve inferred class information between sessions.

The command templates in Figure 4.5 allow Familiar to gather domain knowledge impartially from each application. Familiar does not need special knowledge of the Finder, or any other application, when it uses them. They can be applied to any application. Every object instance in the examples has been observed, and even the application names like “Finder” and “Microsoft Excel” are drawn from the event trace. Some applications are unable to respond to commands based on these templates; these return an error. Familiar is left without context but the user is not informed of the problem.

Familiar’s level of application independence has an immediate, practical benefit: if the user installs a completely new scriptable application that Familiar has never before encountered, issues the *Begin Recording* command, and starts interacting with the application, programming by demonstration will be available right away. The user’s success will depend on how well the application implements AppleScript.

4.3 Sequence recognition

The sequence recognition manager detects iterative patterns in the event trace. Patterns are found in the sequence of command types without reference to the parameter values of the commands—these are the domain of the pattern analysis manager, described in Section 4.4.

After each user action, Familiar searches the event trace for iterative patterns. In Figure 3.3b, for example, the sequence of observed event types is *activate*, *select*, *set*, *select*, *set*. The subsequent predictions in Figure 3.3d show that Familiar elicited a simple cycle of two steps, *select* and *set*, from this sequence.

4.3.1 Detecting patterns

The sequence recognition manager uses a set of *sequence recognition schemes* to search for patterns. Sequence recognition schemes are executed in parallel, and each can detect any number of candidate patterns. When the event trace of Figure 3.3 is passed to the sequence recognition schemes after the fifth event, they report three candidate patterns. *SRS-simple* reports a single pattern of two steps, *select* and *set*, and *SRS-noisy* reports two patterns, one the same as *SRS-simple*, the other comprised of the three steps *activate*, *select*, and *set*.

```
 1  if kind-of-SRS = SRS-noisy then
 2      if times-confirmed-correct-passively > 1 predict 97.1%
 3      else if times-confirmed-correct-passively <= 1 then
 4          if number-of-complete-cycles <= 1 predict 11.1%
 5          else if number-of-complete-cycles > 1 predict 69.0%
 6  else if kind-of-SRS = SRS-simple then
 7      if number-of-commands <= 40 predict 97.0%
 8      else if number-of-commands > 40 then
 9          if proportion-of-agreeing-patterns > 0.285714 predict 100%
10          else if proportion-of-agreeing-patterns <= 0.285714 then
11              if number-of-commands <= 41 predict 70.6%
12              else if number-of-commands > 41 predict 0%
```

Figure 4.7 Rules for estimating the probability a pattern is correct.

Patterns have two components: a sequence of event types, and a set of observed events consistent with the pattern. The first pattern above is described by the event sequence *(select, set)** and has been observed for two iterations, the first comprising the events 2 and 3 in Figure 3.3d, the second events 4 and 5. The other pattern is described by the sequence *(activate, select, set)** and the iterations by events 1,2,3 and *nil*,4,5. The *nil* value indicates that no observed event corresponds to the first command of the second iteration.

The sequence recognition manager must choose a single pattern, which will form the basis for the cycle presented to the user. It estimates which candidate pattern is most likely to represent the user's task using the rules in Figure 4.7. Chapter 5 describes how these rules are derived from a set of training tasks using machine learning techniques. In Figure 3.3d, Familiar chose the pattern predicted by *SRS-simple*. The rules in Figure 4.7 estimate the probability it is correct to be 97.0% (Figure 4.7, line 7), as opposed to 69% for the two-event candidate predicted by *SRS-noisy* (line 5), and 11.1% for three-event candidate (line 4).

The most likely pattern becomes the *current pattern* and is passed to the pattern analysis manager, which uses it as a basis for Familiar's predictions. A new current pattern is chosen in two circumstances: when the user presses the *change cycle* button, as described in Section 3.3.2, or when a new event is recorded that is incompatible with the current pattern.

In Figure 3.3d, Familiar chooses the pattern correctly, and the user quickly automates the task. Figure 3.4a proposes an alternative scenario: Familiar has chosen the wrong pattern, and the user clicks on the *change cycle* button to correct it. In response, Familiar replaces the incorrect pattern with one of the others. The *change cycle* button is implemented by examining each of the competing patterns,

estimating the probability that each is correct, and sorting them accordingly. The most probable pattern is displayed initially, and the next most likely replaces it each time the *change cycle* button is pressed.

4.3.2 Sequence recognition schemes

Familiar currently implements two sequence recognition schemes, *SRS-simple* and *SRS-noisy*, but is designed to make it easy to add new ones. Each scheme must be able to perform these tasks:

- detect patterns in a given event trace,
- update its patterns when a new event occurs,
- provide internally consistent accuracy estimates for each pattern, and
- report its patterns to the sequence recognition manager.

SRS-simple

SRS-simple detects noise-free cycles. It is unsuitable for detecting long patterns because it makes no allowance for mistakes and digressions in the demonstration. Using only *SRS-simple*, Familiar is able to detect the same patterns as Eager (Cypher, 1993b).

SRS-simple often reports more than one pattern at a time. Given the sequence *abababab* (where *a* and *b* represent different types of event), it will report that *abab* is repeated twice and *ab* is repeated four times. The sequence recognition manager will choose between these patterns. The algorithm is sensitive to noise: given the sequence *abababababxab* (where *a* and *b* are relevant event types, and *x* is a noise event), it will observe that the last five events (*abxab*) are consistent with the pattern *xab* and the last seven events (*ababxab*) are consistent with the pattern *abxab*, but will not detect the simple pattern *ab*.

SRS-noisy

SRS-noisy makes allowances for any common mistakes in the user's demonstration. It will report the correct pattern if the demonstration contains at least one correct iteration, or if every iteration contains all the important steps in the pattern.

SRS-noisy obtains the type of the most recent event, finds all events of that type in the event trace, and assumes that these are the final events in each iteration. A pattern is created corresponding to each iteration, and a final pattern is created

1	event number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	event type	a	b	c	b	c	d	e	a	f	b	c	d	e	a	g	f
3					first iteration						second iteration						
4	first pattern	1	2	3	4	5	6	7	8	9	4	5	6	7	8		9
5	second pattern				1	2	3	4	5	7	1	2	3	4	5	6	7
6	combined pattern				1	2	3	4	5	6	1	2	3	4	5		6

Figure 4.8 Candidate patterns found by *SRS-noisy* in the event trace shown in Figure 3.10a,b.

from the event types that occur in every iteration. The history of each pattern is calculated by searching every iteration for an event corresponding to each event type in the pattern.

Returning to Figure 3.10, the types of events 1 through 16 in the demonstration might be represented by *abcbcdeafbcdeagf* (where *a = activate Finder*, *b = select*, *c = open*, *d = activate GIFConverter*, *e = save*, *f = delete*, and *g = set*). These events are shown in Figure 4.8, rows 1 and 2. The most recent event has type *f*, so *SRS-noisy* splits the sequence into two iterations ending with event *f* (Figure 4.8, row 3). It then creates a pattern based on each iteration: *abcbcdeaf* and *bcdeagf*. It creates a third pattern from the events that occur in both iterations: *bcdeaf*. The event history constructed for the first pattern is 1,2,3,4,5,6,7,8,9 and nil,nil,nil,10,11,12,13,14,16 (Figure 4.8, row 4); for the second 4,5,6,7,8,nil,9 and 10,11,12,13,14,15,16 (row 5); and for the third 4,5,6,7,8,9 and 10,11,12,13,14,16 (row 6). *SRS-noisy* reports each of these patterns to the sequence recognition manager, which selects the third as the basis for the prediction in Figure 3.10c.

Sequences that are extremely noisy, or that contain two or more occurrences of each event type (for example, *aabbaabb*) are not detected by *SRS-noisy*.

4.4 Pattern analysis

Once the most likely pattern has been determined, the pattern analysis manager is given an iterative pattern and predicts the next iteration of the task. The next iteration is broken down into its constituent commands, commands are broken down into parameters, and parameters are predicted by invoking pattern analysis schemes. Predicted parameters are combined to build command predictions, and the sequence of command predictions form the predicted iteration of the task. This prediction is then displayed in Familiar's *prediction* window.

Returning to the simple example in Figure 3.3, suppose the sequence recognition manager has detected an iterative cycle composed of a *select* command and a *set* command. The *select* command has one parameter, the direct (or *select*) parameter. Familiar predicts that the its next value will be *file "apple" of folder "fruit"*, and constructs the prediction *select file "apple" of folder "fruit"* for event 6. The *set* command has two parameters, *set* and *to*. Familiar predicts that their next values will be *position of selection* and {144,29} respectively, and constructs the command *set position of selection to {144,29}* for event 7. The two commands make up the predicted iteration in Figure 3.3d.

4.4.1 Evaluating competing predictions

Each parameter is predicted in parallel by a set of competing pattern analysis schemes. Each scheme makes zero or one predictions based on the event trace and Familiar's other knowledge sources. In the current implementation, five schemes can each make a prediction, and their predictions may conflict. Consider the direct parameter of the *select* command in the example above. According to the pattern, the previous values of the parameter were *file "plum"* and *file "peach"* (Figure 3.3b, events 2 and 4). Two pattern analysis schemes predict the value at event 6: *PAS-constant* predicts *file "peach"* (the previous value) and *PAS-set* predicts *file "apple"*.

Familiar must choose exactly one prediction of each parameter to display, but predictions made by different schemes are not directly comparable. The pattern analysis manager uses a complex set of rules to estimate the probability that each prediction is correct. Figure 4.9 shows the rules relating to *PAS-constant* and *PAS-set*. The rules pertaining to the other branches are not shown here. The derivation of all the rules is described in Section 5.3.

In the example above, the rules estimate the probability of the *PAS-set* prediction as 7.3% (line 46), and of the *PAS-constant* prediction as 0% (line 3). Familiar therefore prefers the *PAS-set* prediction, and suggests *file "apple"* to the user (Figure 3.3d).

4.4.2 Explanations

One of the advantages using competing pattern analysis schemes is that each one follows a simple heuristic. Heuristics are intended to be easily understood by human users, consequently each prediction can be explained in a few concise

```
1  if PAS-kind = PAS-constant then
2      if prop-agreeing-predictions <= 0.666667 then
3          if PAS-specific-2 <= 3 predict 0%
4          else if PAS-specific-2 > 3 then
5              if number-of-iterations > 5 predict 1.7%
6              else if number-of-iterations <= 5 then
7                  if agreeing-predictions > 0 predict 100%
8                  else if agreeing-predictions <= 0 then
9                      if PAS-specific-1 > 1 predict 100%
10                     else if PAS-specific-1 <= 1 then
11                         if number-of-iterations > 2 predict 0%
12                         else if number-of-iterations <= 2 then
13                             if other-predictions <= 0 predict 75.0%
14                             else if other-predictions > 0 then
15                                 if PAS-specific-2 <= 5 predict 75.0%
16                                 else if PAS-specific-2 > 5 predict 0%
17     else if prop-agreeing-predictions > 0.666667 then
18         if PAS-specific-1 > 3 predict 99.7%
19         else if PAS-specific-1 <= 3 then
20             if number-pp <= 0 predict 97.1%
21             else if number-pp > 0 then
22                 if other-predictions <= 1 predict 14.3%
23                 else if other-predictions > 1 predict 100%
   …
43 else if PAS-kind = PAS-set then
44     if proportion-true-pp > 0 predict 100%
45     else if proportion-true-pp <= 0 then
46         if agreeing-predictions <= 0 predict 7.3%
47         else if agreeing-predictions > 0 predict 100%
```

Figure 4.9 Rules for estimating the probability that a prediction is correct.

sentences. This is consistent with design guidelines that suggest decomposing an agent interface into conceptual units that reflect the way the user will think about the agent and task (Sengers, 1999).

Parameter predictions are explained to the user in natural language through the Macintosh bubble help mechanism (Figures 3.7 and 3.9). The explanations are generated by the pattern analysis manager in tandem with the schemes making the prediction.

An explanation consists of three paragraphs: the title, the explanation text, and the instruction for changing the prediction. The explanation title and text are generated from templates by the individual pattern analysis schemes described in Section 4.4.3 below. The instructions for changing prediction are appended by the pattern analysis manager. If only one scheme predicts the parameter, the instruction reads *Give another example to change this prediction* (Figure 3.9a). If

more than one prediction of the parameter has been made, the instruction reads *Option-click to change this prediction* (Figure 3.9c).

4.4.3 Pattern analysis schemes

Familiar maintains pattern analysis schemes for detecting constants; numeric, alphabetic, and enumerated sequences; iteration over sets of objects; relationships between parameters; and conditional rules. Given the past values of a parameter, each must be able to:

- predict the next value,
- update its predictions in the next iteration,
- report its predictions,
- explain its predictions in simple sentences,
- provide internally consistent accuracy estimates of its predictions, and
- evaluate its predictions by comparing them to the event that the user executes.

Most pattern analysis schemes are implemented with two modes: they are either searching for a prediction, or maintaining a prediction. In search mode the scheme has made no prediction and must search for one; this involves initialising the scheme's parameters, detecting a pattern in the event trace, storing it, and extrapolating it into the future. Each prediction is assigned a unique identifier which the pattern analysis manager can later use to request that the scheme report the prediction, explanation, or accuracy estimates.

Once a prediction is made, the scheme switches to maintenance mode. Each prediction is ultimately tested when the user performs the action it predicts. It is then evaluated: the scheme calculates whether it is correct or incorrect by comparing the observed value to the one it predicted. If the prediction is incorrect, it is discarded, and the scheme returns to search mode. If it is correct, the prediction is updated to predict the next iteration.

PAS-constant

The simplest scheme, *PAS-constant*, always predicts that the next value of a parameter will be the same as the last. Extrapolating from the trace in Figure 4.10 it correctly predicts that the *set* parameter in event 7 is *FormulaR1C1 of ActiveCell*, and incorrectly predicts that the *to* parameter is "*August*".

```
1  activate – Microsoft Excel
2  Select Range "R1C2"
3  set FormulaR1C1 of ActiveCell to "July"
4  Select Range "R1C3"
5  set FormulaR1C1 of ActiveCell to "August"
6  Select Range "R1C4"
```

Figure 4.10 An event trace from Microsoft Excel.

PAS-constant maintains a particular prediction for as long as the parameter remains constant, then abandons it and makes a new prediction corresponding to the new parameter value. Unlike the other pattern analysis schemes, *PAS-constant* always makes a prediction so that Familiar always has at least one prediction to display.

Table 4.2a summarises the *PAS-constant* scheme. It predicts the last parameter value, and its accuracy estimates are the number of times the parameter value has remained constant and an enumeration indicating the kind of predicted object. The explanation template explains that the prediction is constant and its value, resulting in explanations like those in Figure 3.7b and Figure 3.9a.

PAS-extrapolation

The *PAS-extrapolation* scheme finds patterns in the text of parameter values. Given the event trace in Figure 4.10, the scheme will extrapolate from the three *Select* parameters to predict that *Range "R1C5"* is the next value, and will recognise that *July* and *August* are elements of an enumeration whose next element is *September*.

PAS-extrapolation works by coercing parameter values (including numbers and object descriptions) into strings, breaking them into tokens, and detecting constants, numeric series (arithmetic and geometric), and the enumerations described in Section 4.2.3 (extrapolated as integers using modulo arithmetic). *PAS-extrapolation* makes predictions after a sequence has occurred for two or more iterations, and maintains it for as long as it is evident in the data.

The scheme is summarised in Table 4.2b. The scheme's accuracy estimates are the number of times the parameter value has been consistent with the pattern it is extrapolating and an enumeration describing the type of sequence. The explanation text is constructed from the parameter name and descriptions of each of the tokens that make up each predicted string.

(a)		
	Name	PAS-constant
	Summary	predict the last value of the parameter
	Accuracy estimate 1	number of times the parameter value has been constant
	Accuracy estimate 2	kind of object predicted (e.g. string, number, object, etc)
	Explanation title	Constant value.
	Explanation text	The parameter <parameter name> is always predicted using the value <prediction>

(b)		
	Name	PAS-extrapolation
	Summary	find alphabetic, numeric, and enumeration sequences
	Accuracy estimate 1	number of times the sequence is consistent
	Accuracy estimate 2	kind of object predicted (e.g. string, number, object, etc)
	Explanation title	Sequence of values.
	Explanation text	The parameter <parameter name> is predicted in <number of tokens> parts. Part 1 is found by <part 1 explanation>. Part 2 is found by…

(c)		
	Name	PAS-previous
	Summary	find identical parameter values
	Accuracy estimate 1	kind of object predicted (e.g. string, number, object, etc)
	Accuracy estimate 2	none
	Explanation title	Matches previous value.
	Explanation text	The parameter <parameter name> is the same as the <parameter name> parameter of event <event number>.

(d)		
	Name	PAS-set
	Summary	iterate over containee objects
	Accuracy estimate 1	none
	Accuracy estimate 2	none
	Explanation title	Set iteration..
	Explanation text	The parameter <parameter name> is predicted by choosing the next <containee file type> from <container>.

(e)		
	Name	*PAS-ML*
	Summary	create conditional rules based on contextual information
	Accuracy estimate 1	statistical significance of rule
	Accuracy estimate 2	accuracy of rule
	Explanation title	Choice of value
	Explanation text	The parameter <parameter name> depends on the value of <contextual attribute description>.

Table 4.2 Summary of the pattern analysis schemes.

PAS-previous

The *PAS-previous* scheme searches the event trace for parameters that consistently have the same value as the target parameter. This is useful in tasks where more than one operation is performed on the same object, and tasks that cannot be fully automated: an object used in the first step of a cycle of the task can be referred to in later commands.

The event trace is searched relative to the current pattern. Only events that form part of the pattern are searched, and the search starts with the most recent event and extends back a full single iteration. If a parameter value is found that is the same as the target parameter in every iteration of the task, *PAS-previous* predicts that the target parameter will always be the same as this parameter.

Table 4.2c summarises the *PAS-previous* scheme. It provides only one PAS-specific accuracy estimate, an enumeration describing the object predicted, and the explanation text identifies the parameter name and event number that the prediction is based on.

PAS-set

PAS-set detects iteration over sets of objects that have the same container object. In Figure 3.3, for example, the user iterates over the set of *files* in a *folder*.

When the same container appears in consecutive iterations, *PAS-set* asks the application for all the containee objects, and predicts the first one it has not already seen. In events 2 and 4 of Figure 3.3b, the container object is *folder "fruit"*, and two containees are observed: *file "plum"* (line 2) and *file "peach"* (line 4). *PAS-set* assumes that the user is working through all the *file* objects contained by *folder "fruit"*, and fetched the remaining files with the command *tell application "Finder" to get every file of folder "fruit"*. (This command is based on the template in Figure 4.5e, discussed in Section 4.2.2.) The Finder returns a list of *file* objects, which *PAS-set* sorts (alphabetically by name). It then predicts the first one not already seen in the event trace (Figure 3.3d, event 6).

PAS-set handles situations where the user iterates over a set of objects in alphabetical order (or where order is unimportant), but cannot filter and sort objects based on their attributes. The ability to filter a set would allow the user to iterate over some, but not all, of a set of objects. If the objects could be sorted arbitrarily, the user could automate tasks that rely on a strict ordering offf the data. Filtering and sorting were not added to *PAS-set* because they were not required in the user evaluation described in Chapter 7, but form part of the future work described in Section 8.4.

PAS-set makes no further predictions after iterating over all the containee objects. The scheme is summarised in Table 4.2d. It formulates an explanation by identifying the containee type and the name of the container object, as can be seen in Figure 3.7c.

event	recorded command
5	select file "ACC01.doc" of folder "Documents"
6	move selection to folder "word processor" of folder "Documents"
7	select file "ACC99.doc" of folder "Documents"
8	move selection to folder "word processor" of folder "Documents"
12	select file "Balance sheet" of folder "Documents"
13	move selection to folder "spreadsheets" of folder "Documents"
14	select file "Balance sheet 1996" of folder "Documents"
15	move selection to folder "spreadsheets" of folder "Documents"
16	select file "Corrections" of folder "Documents"
17	move selection to folder "word processor" of folder "Documents"
18	select file "expenses for 1996" of folder "Documents"

Figure 4.11 Excerpts from the event trace in Figure 3.8 and 3.9.

PAS-ML

PAS-ML predicts parameters using simple conditional rules. The rules are formed using machine learning techniques based on the contextual information gathered by the event recorder described in Section 4.2.2. This section gives an overview of the *PAS-ML* scheme; it is described in detail in Section 5.3.

Section 3.3.3 described a simple classification task: the user sorted a set of files into two folders, one for word processor documents, the other for spreadsheets. Figure 4.11 shows parts of the event trace generated as the user demonstrated the task. The sequence recognition manager has detected a cycle of two steps, *select* and *move*. The user has demonstrated up to event 18; Familiar must now predict event 19, a *move* command with two parameters called *move* and *to*.

The difficult aspect of this task is learning the value of the *to* parameter, which determines where each file is moved to. There is no obvious pattern in the destination folders—the previous values are *word processor*, *word processor*, *spreadsheet*, *spreadsheet*, and *word processor*—so the event trace alone does not provide enough information to predict the next value. *PAS-ML* therefore searches for contextual information that will help it make the prediction.

Table 4.3 lists the values of the *to* parameter in each *move* command of Figure 4.11, with some contextual information. The context includes the *selection* at the time of the *set* command, the *creator type* of the selection, and the *creation date* of the selection. Contextual information is also provided for event 19, even though it has not yet occurred. Familiar will use the contextual information for event 19 to predict its value.

event	*to* parameter	selection	creator type of selection	creation date of selection
6	folder "word processor"	file "ACC01.doc"	MSWD	15 Jun 1997
8	folder "word processor"	file "ACC99.doc"	MSWD	4 Apr 1999
13	folder "spreadsheets"	file "Balance sheet"	XCEL	4 Apr 1999
15	folder "spreadsheets"	file "Balance sheet 1996"	XCEL	8 Jun 1996
17	folder "word processor"	file "Corrections"	MSWD	16 Dec 1997
19	unknown	file "expenses for 1998"	XCEL	18 May 1998

Table 4.3 Contextual data for the *to* parameter in Figure 4.11.

PAS-ML uses events 6–17 of Table 4.3 to build a rule for predicting the *to* parameter using the other attributes: this rule is shown in Figure 4.12. The rule correctly predicts each of these events based on the *creator type of selection* attribute. This rule, and the contextual information that the event recorder has gathered about event 19, are used to predict the next value of the *to* parameter. The *file type of selection* at event 19 is *XCEL* (Table 4.3, event 19), so *PAS-ML* predicts that the next value of the *to* parameter is *folder "spreadsheets" of folder "Documents"* (Figure 4.12, line 4).

This example is extremely simplified. It glosses over two significant problems. First, there are hundreds or thousands of contextual parameters; only three are shown here for simplicity. Familiar must choose between them when it forms a rule. Second, rules must be formed in real-time based on very few examples. *PAS-ML* handles this problem using permutation tests, a machine learning technique adopted from statistics that is suited to problems with very little data. A full explanation is in Section 5.3.

Table 4.2e summarises the *PAS-ML* scheme. The two accuracy estimates calculated for each prediction are the statistical significance and accuracy of the rule (both explained in Section 5.3). The explanation does not describe full rule like that is Figure 4.12, it simply explains which contextual value the rule is based on. An example is visible in Figure 3.9c,e.

4.5 Summary

This chapter has described Familiar's architecture. Familiar works with existing recordable applications, and interacts with each using AppleScript commands based on standard command templates. Consequently, it achieves a level of application independence that goes beyond other inferencing PBD systems: it is able to work with completely new applications that it has never before

```
1  if creator type of selection = MSWD then
2      predict folder "word processor" of folder "Documents"
3  else if creator type of selection = XCEL then predict 3
4      predict folder "spreadsheets" of folder "Documents"
5  otherwise make no prediction
```

Figure 4.12 A rule for predicting the *to* parameter based on the file type of the selection.

encountered, but at the same time it operates at a high-level, exploiting detailed knowledge of each application to make predictions.

The Familiar inferencing system is applicable to any high-level event system. It works in two phases, first detecting iterative patterns, then extrapolating command parameters. Its modular design is easily extended, and tolerates noise, incorporates feedback, and generates explanations of its predictions. Familiar incorporates standard machine learning techniques to guide prediction and to learn conditional rules from the user's demonstration: these features are described in detail in Chapter 5.

5 Machine Learning

This thesis argues that a system-wide, application-independent PBD interface for performing iterative tasks can be made available in existing applications, and presents the Familiar system to support this claim. Familiar's design, interface, and architecture were described in the previous two chapters. This chapter completes the description by showing how machine learning is used to guide prediction and infer conditional rules.

Machine learning performs two distinct roles in Familiar. The first is to guide prediction, and manifests itself as the two sets of rules that the sequence recognition and pattern analysis components used to evaluate candidate predictions (Sections 4.3 and 4.4). The second is to recognise and explain parameter values that are based on conditional rules, as in the example of Section 3.3.3 where the user sorts documents into different folders depending on their type. These calculations are made by the *PAS-ML* pattern analysis scheme (Section 4.4.3).

Familiar reuses existing technology whenever possible. Existing machine learning techniques are used in the inferencing system, just as existing applications provide functionality and an existing scripting language is used to communicate with the user and other applications. This chapter describes how Familiar uses two machine learning algorithms, the C4.5 decision tree learner and the simple 1R rule learner, and examines their strengths and weaknesses in PBD systems.

Both C4.5 and 1R build classifiers. Classifiers learn to assign objects to categories based on their attributes. These categories are called *classes*. Classifiers have been applied to a variety of problems, such as determining the species of Iris plants based on biologist's observations (Fisher, 1936) and predicting whether contracts are acceptable or unacceptable based on the terms of employment (Bergadano *et al.*, 1988). Classifiers can be expressed in many forms, but this thesis prefers

decision trees and rules because their output is easily interpreted and implemented in computer programs.

Machine learning algorithms examine a training dataset to learn the reasons that classes are assigned and build classifiers to express these reasons. The classifier can then be applied to new examples whose class is unknown. Familiar uses classifiers trained on historical data to make predictions about new examples. They fulfil two roles: guiding prediction and learning conditional rules.

The C4.5 decision tree learner is used to guide prediction. The decision trees it constructs are used to build the rule sets that choose the best sequence recognition and pattern analysis schemes, as described in Sections 4.3 and 4.4. Both sets of rules are based on a decision tree trained on data gathered from actual iterative tasks. An evaluation shows that this is an effective selection criteria, and can be improved by adapting the decision tree to individual users.

The *PAS-ML* pattern analysis scheme builds classifiers that learn conditional rules from the user's demonstration. These classifiers are then used to predict future values of parameters. The original version of Familiar used the 1R algorithm to infer rules, but suffered from a bias towards attributes with large numbers of values. This problem was eliminated by changing the rule selection criterion to use a permutation test. The permutation test is particularly appropriate to PBD problems because it works well with very small datasets and calculates the statistical significance of the resulting rule.

The next section gives a brief overview of how classifiers are trained and used. Section 5.2 explains the derivation of Familiar's rules for evaluating candidate patterns and candidate predictions. Section 5.3 explains how the *PAS-ML* pattern analysis scheme uses machine learning to infer conditionals in users' demonstrations.

5.1 Training and using classifiers

A large class of learning problems are broadly described as classification problems. Classification is much studied, and many solutions have been published. A thorough treatment of the field, including descriptions of machine learning algorithms and applications, can be found in Witten and Frank (2000).

	play?	outlook	temperature	humidity	windy
1	no	sunny	85	85	false
2	no	sunny	80	90	true
3	yes	overcast	83	86	false
4	yes	rainy	70	96	false
5	yes	rainy	68	80	false
6	no	rainy	65	70	true
7	yes	overcast	64	65	true
8	no	sunny	72	95	false
9	yes	sunny	69	70	false
10	yes	rainy	75	80	false
11	yes	sunny	75	70	true
12	yes	overcast	72	90	true
13	yes	overcast	81	75	false
14	no	rainy	71	91	true

Table 5.1 A simple machine learning problem (adapted from Quinlan, 1986).

5.1.1 The classification problem

The goal of a classifier is to assign *class* labels to records, usually called *instances* or *examples*. Machine learning systems are trained on a set of instances, each of which has a known class, and output a classifier capable of assigning classes to new instances whose class is unknown. Classifiers are evaluated by measuring the accuracy with which they classify new examples.

Table 5.1 shows a fictitious machine learning dataset. Each row is an instance, and Table 5.1 shows 14 instances describing whether an unspecified game was played on a given day. The *play?* column contains the class of each instance and can have one of two values: *yes* and *no*. The remaining columns are called the *attributes* of the instances. Here they describe the atmospheric conditions on the day: the *outlook*, the *temperature*, the *humidity*, and whether or not it is *windy*.

The goal of machine learning is to explain the class value of each instance in terms of the other attributes. In this example, the classifier attempts to learn whether play will proceed based on the prevailing atmospheric conditions.

5.1.2 Trees and rules

Most classifiers represent what they have learned using trees or rules. These are a convenient form of output because they explain the data in a way that is easily understood and implemented.

Figure 5.1 shows a decision tree learned from the data in Table 5.1. To use a decision tree to classify a new instance, start at the root node (labelled *outlook*)

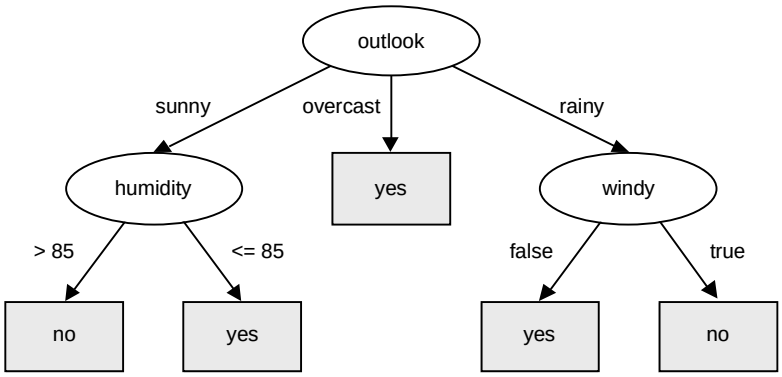


Figure 5.1 Decision tree for the weather problem.

and trace a path through the internal nodes of the tree, leaving each node by the arc whose label matches the instance's attribute value. The path terminates at a leaf node, which is labelled with the predicted class. In Figure 5.1, for example, if the *outlook* is *sunny* and the *humidity* is 80, the classifier would predict the class *yes*.

Figure 5.2a shows the same tree in the text format output by the software used in Section 5.2 (Witten and Frank, 2000). In this representation, leaf nodes are labelled with their class, either *yes* or *no*, and one or two numbers in brackets. When the leaf node has a single number (lines 1, 3, 6, 7) it is the number of instances in the training data correctly classified by the leaf node. For nodes with two numbers (line 4) the first represents the number of training instances correctly classified by the node, the second the number incorrectly classified. The leaf node at line 4 of Figure 5.2a, for example, correctly predicts that play will proceed for two training instances (Table 5.1, lines 2,8) and incorrectly predicts that it will on another occasion (Table 5.1, line 1).

Any decision tree can be expressed as an equivalent set of rules. Figure 5.2b shows rules equivalent to the tree in Figure 5.2a. The rules classify new instances the same way as the tree does. Rules can be expressed in a variety of forms, including classification rules, decision lists, and ripple-down rules (Witten and Frank, 2000). This thesis uses sets of ordered nested rules, resembling pseudocode, that exactly duplicate the structure of the decision tree. Each set of rules can be used to classify any instance in the data.

(a)

```
1  'outlook' = overcast : yes (4)
2  'outlook' = sunny
3  |   'humidity' > 85 : no (2)
4  |   'humidity' <= 85 : yes (2,1)
5  'outlook' = rainy then
6  |   'windy' = true : no (2)
7  |   'windy' = false : yes (3)
```

(b)

```
1  if outlook = overcast then predict yes
2  else if outlook = sunny then
3      if humidity > 85 then predict no
4      else if humidity <= 85 then predict yes
5  else if outlook = rainy then
6      if windy = true then predict no
7      else if windy = false then predict yes
```

Figure 5.2 Predicting the class of the weather problem with (a) a textual representation of a tree, and (b) an equivalent rule.

5.1.3 Training, testing, and prediction

Machine learning schemes are evaluated by training them on one dataset and testing them on another. The training dataset, like the one in Table 5.1, has known class values. The machine learning algorithm generates a classifier, often a decision tree or a set of rules, that can be used to assign a class to any instance whose attributes are known.

A classifier is evaluated by using it to assign classes to each instance of the test dataset. The number of classes assigned correctly and incorrectly are recorded, and used to calculate the proportion of correct classifications. This proportion is called the *accuracy*, and is used to measure the performance of the classifier.

Machine learning can be used for prediction by building a classifier using training data from past performances of a task, where the class value is some feature we wish to predict. Future performances can be predicted by creating a new instance, calculating its attribute values, and using the classifier to predict its class value. This is the strategy that Familiar uses to guide prediction and learn conditionals.

5.1.4 Machine learning algorithms

Machine learning is currently an active area of research, and there are many different algorithms for generating classifiers from training data. This thesis requires algorithms that offer reasonable performance and produce interpretable

output. C4.5 and 1R were chosen after reviewing a number of machine learning schemes.

C4.5 is a decision tree learner based on a divide and conquer strategy (Quinlan 1993). It is known to have good performance, and has become the standard algorithm against which others are compared. Section 5.2 describes how Familiar uses an implementation of C4.5 to guide prediction.

1R constructs rules based on a single attribute value using an exhaustive search (Holte, 1993). Although it produces very simple classifiers, they meet the requirements of the user evaluation (Section 7.1) and of most of the other tasks described in this thesis (some exceptions appear in Section 7.2.3). Section 5.3 describes how Familiar uses an adaptation of 1R to infer conditional rules from data in the event trace.

Most machine learning algorithms work best with hundreds or thousands of instances and only a few attributes. Unfortunately, PBD systems are in the opposite situation: they must learn from one or two user examples (instances), and each prediction has hundreds or thousands of attributes. Familiar's solution to this problem is discussed in Section 5.3.

5.2 Guiding prediction

Chapter 4 explains Familiar's two levels of inferencing. First, the sequence recognition manager searches for iterative patterns in the types of events that the user has demonstrated (Section 4.3). Each of Familiar's sequence recognition schemes can suggest zero or more candidate patterns, and the sequence recognition manager selects the single best candidate pattern. Second, the pattern analysis manager predicts the next value of every parameter of every step in the best pattern (Section 4.4). It uses pattern analysis schemes, each of which can make zero or more candidate predictions, to extrapolate future values of the pattern. The best candidate prediction of each command parameter is suggested to the user.

Both the sequence recognition manager and the pattern analysis manager exploit competing schemes to make predictions, then select the best candidate and present it to the user. In each case, the best candidate is determined by using a set of rules to estimate the probability that each candidate is correct. The most probable candidate is selected and displayed to the user. This section describes

the derivation of two rules for evaluating predictions, one for sequence recognition and the other for pattern analysis.

5.2.1 Sequence recognition

The sequence recognition manager uses a set of rules to calculate the probability that a particular candidate pattern is the user's intended pattern (Section 4.3). These rules are shown in Figure 4.7. They were constructed from a decision tree built by the C4.5 decision tree learner. In order to create the tree, and then the rules, C4.5 was trained on appropriate data.

Training data

Training data was gathered by performing the iterative tasks described in Appendix B. The tasks were performed by a single experienced user. As they were performed, Familiar detected many patterns, and calculated and stored a set of attributes for each. Later, it recorded whether the pattern was correct (consistent with the commands subsequently executed by the user) or incorrect (inconsistent with subsequent user actions).

Each candidate pattern is represented by an instance in the training dataset; each instance consists of a class variable and a set of attributes. The class value is whether the pattern is *correct* or *incorrect*. Although this value is unknown when the pattern is detected, it will be discovered later: if the user executes commands from the pattern it is correct; otherwise it is not. The twelve attributes listed in Figure 5.3 are calculated after each event for every pattern detected in the event trace. These attributes are chosen because they are easily calculated and were factors in earlier attempts to define heuristics for choosing patterns, and their performance suggests they are at least adequate, though they have not been evaluated. Perhaps better attributes exist for classifying patterns, but their discovery is beyond the scope of this thesis.

The first three attributes in Figure 5.3 are the particular sequence recognition scheme that detected the pattern, and the internal estimates of the pattern's likelihood described in Section 4.3.2. Attribute 4 is the number of high-level commands in each iteration, and attributes 5–7 describe the number of observed user actions consistent with the pattern. The next two attributes represent the number of other sequence recognition schemes that have found the same pattern of events (possibly with a different associated history). Attributes 10–12 are the

	Attribute name	Description
1	kind-of-SRS	kind of sequence recognition scheme (SRS)
2	SRS-specific-1	first accuracy estimate calculated by SRS
3	SRS-specific-2	second accuracy estimate calculated by SRS
4	cycle-length	length of the pattern cycle in commands
5	number-of-cycles	number of partial cycles in pattern history
6	number-of-complete-cycles	number of complete cycles in pattern history
7	number-of-commands	number of commands in pattern history
8	number-of-agreeing-patterns	number of agreeing pattern cycles
9	proportion-of-agreeing-patterns	proportion of agreeing pattern cycles
10	times-confirmed-correct-passively	times pattern confirmed correct passively
11	times-confirmed-correct-actively	times pattern confirmed correct actively
12	times-confirmed-incorrect	times pattern confirmed incorrect

Figure 5.3 Attributes of candidate patterns.

number of times the pattern has been confirmed correct by the user performing matching actions in the application interface, the number of times the pattern has been confirmed correct by the user executing commands in the prediction window, and the number of times it has been confirmed incorrect by the user clicking on the *change cycle* button. Some of these attributes are updated if the user interacts with Familiar or the application.

The training data consists of 1466 instances, each representing a candidate pattern. Of these, 1380 were correct and 86 incorrect. There were many more correct than incorrect patterns because the event trace contains little or no noise after the user starts interacting with the Familiar interface, and patterns are easily detected and consistently correct.

Learning algorithm

The C4.5 decision tree learner (Quinlan, 1993) was used to build a classifier from the data gathered as the training tasks were performed. The decision tree is used to compare predictions from different pattern analysis schemes, so the C4.5 implementation was modified to split on the *kind-of-SRS* attribute at the root node of the tree. Building a decision tree with this restriction is equivalent to splitting the dataset on the *kind-of-SRS* attribute, learning a tree for each scheme, and classifying each prediction with the appropriate tree. This restriction makes the decision tree easier to understand and implement, and ensures that the decision tree only branches on the internal accuracy estimates (*SRS-specific-1*, *SRS-specific-2*) after the *kind-of-PAS* attribute. This is important because accuracy estimates are only relevant in the context of the sequence recognition scheme they describe.

```
 1  'kind-of-SRS' = SRS-noisy
 2  |   'times-confirmed-correct-passively' > 1: correct (370.0/11.0)
 3  |   'times-confirmed-correct-passively' <= 1
 4  |   |   'number-of-complete-cycles' <= 1: incorrect (24.0/3.0)
 5  |   |   'number-of-complete-cycles' > 1: correct (29.0/13.0)
 6  'kind-of-SRS' = SRS-simple
 7  |   'number-of-commands' <= 40: correct (1013.0/31.0)
 8  |   'number-of-commands' > 40
 9  |   |   'proportion-of-agreeing-patterns' > 0.285714: correct (12.0)
10  |   |   'proportion-of-agreeing-patterns' <= 0.285714
11  |   |   |   'number-of-commands' <= 41: correct (12.0/5.0)
12  |   |   |   'number-of-commands' > 41: incorrect (5.0)
```

Figure 5.4 Decision tree for predicting whether a pattern is correct.

The entire decision tree is shown in Figure 5.4. The current version of Familiar employs two sequence recognition schemes, described in Section 4.3.2, and a branch has been created for each. The *SRS-noisy* branch (Figure 5.4, lines 1–5) uses the number of times the pattern was confirmed correct by user actions in the application interface and the number of complete cycles observed. The most prominent attribute in the *SRS-simple* branch (lines 6-12) is the number of observed user actions consistent with the pattern, which branches both at 40 and 41, suggesting that it may be overfitting the training data.

Although the structure of the tree and the rules in Figure 4.7 are the same, each leaf node of the tree classifies the instance as *correct* or *incorrect*, whereas the rules estimate the probability that it is *correct*. This probability is the proportion of the training instances classified by the leaf node that have the class *correct*. The number of training instances classified correctly and incorrectly are attached to each leaf node in the tree. For example, line 2 of Figure 5.4 is a leaf node, labelled *correct*, and corresponds to line 2 of the rules in Figure 4.7. The leaf node classifies 370 true instances and 11 false instances from the training data; this corresponds to the estimate that any prediction classified by this node has a probability of $370/(370 + 11) = 97.1\%$ of being correct. Line 4 of Figure 5.4 is also a leaf node, but instances falling on this node have the class *incorrect*. It classifies 24 training instances with class *incorrect* and 3 with class *correct*, corresponding to the $3/(24 + 3) = 11.1\%$ estimated probability of being *correct* on line 4 of Figure 4.7.

5.2.2 Pattern analysis

The pattern analysis manager must estimate the probability that each candidate parameter prediction is correct so that it can display the best prediction to the

user, as described in Section 4.4.1. When a new candidate prediction is made, the pattern analysis manager calculates 18 attribute values (described below) and uses a set of rules, some of which are shown in Figure 4.9, to estimate the probability that it is correct. These rules are derived from a decision tree in a similar fashion to the sequence recognition rules described in Section 5.2.1. The decision tree was trained on data from the same set of example tasks, and transformed into rules in the same way.

Training data

Training data for the pattern analysis classifier was gathered at the same time as the training data for sequence recognition classifier described in Section 5.2.1. Each time a pattern analysis scheme made a prediction, Familiar calculated and stored the eighteen attributes listed in Figure 5.5. Later, it recorded whether the prediction was correct (executed by the user) or incorrect (rejected or ignored by the user). Each parameter prediction is encoded as an instance, consisting of a boolean class value (*correct* or *incorrect*) and the set of attribute values calculated when the prediction is made.

Figure 5.5 lists the attributes of a parameter prediction. These attributes were chosen on they basis that they are easily calculated and appear (in some form) in the heuristic calculations that were used to evaluate predictions before the machine learning techniques were explored (Section 5.2.3). Better attributes for evaluating prediction correctness probably exist, but again they lie beyond the scope of this thesis. The evaluation in Section 5.2.3 shows that the attributes used are of a sufficient standard to improve on the original heuristics.

The first attribute in Figure 5.5 identifies the pattern analysis scheme making the prediction and the next two are its internal accuracy estimates. These are derived differently for each pattern analysis scheme, as described in Section 4.3.3, and should only be used in conjunction with the *kind-of-PAS* attribute. The training tasks deliberately use a diverse range of applications and commands, consequently the fourth attribute, *event and parameter name*, was highly correlated with the class and likely to overfit. It was removed from the dataset. Attributes 5 and 6 identify the length of the current pattern, and the number of iterations seen. Attributes 7–9 are concerned with the number of other predictions agreeing with this one, and the remaining attributes detail the performance of the parameter prediction in previous iterations of the task.

	Attribute name	Description
1	PAS-kind	kind of pattern analysis scheme (PAS)
2	PAS-specific-1	first accuracy estimate calculated by PAS
3	PAS-specific-2	second accuracy estimate calculated by PAS
4	event	event and parameter name
5	cycle-length	length of each iteration
6	number-of-iterations	number of iterations in pattern history
7	other-predictions	number of other predictions made
8	agreeing-predictions	number of agreeing predictions made
9	prop-agreeing-predictions	proportion of other predictions agreeing
10	number-pp	number of iterations with prediction
11	number-true-pp	number of iterations with a true prediction
12	proportion-true-pp	proportion of iterations with a true prediction
13	number-false-pp	number of iterations with a false prediction
14	proportion-false-pp	proportion of iterations with a false prediction
15	number-true-consec-pp	number of consecutive true predictions
16	proportion-true-consec-pp	proportion of consecutive true predictions
17	number-false-consec-pp	number of consecutive false predictions
18	proportion-false-consec-pp	proportion of consecutive false predictions

Figure 5.5 Attributes used to build the model of prediction correctness.

The final training dataset contains 1875 instances representing parameter predictions, 1459 of which are *correct*, and 416 *incorrect*.

Learning algorithm

The classifier was built off-line by an implementation of the C4.5 decision tree learner using the data gathered from the training tasks. Because it is used to compare predictions from different pattern analysis schemes, the classifier was forced to split on the *kind-of-PAS* attribute at the root node of the tree. Consequently the decision tree is easier to understand and implement, and only branches on *PAS-specific accuracy estimates* after the *kind-of-PAS* attribute.

The entire decision tree produced by this process is shown in Figure 5.6. The rules for estimating the probability of a prediction in Figure 4.9 were derived from this tree by the process described in Section 5.2.1. The rules that are applied to *PAS-constant* predictions, for example, appear on lines 1–23 of Figure 4.9, and are based on lines 1–23 of Figure 5.6. The tree contains a branch for each of the five pattern analysis schemes described in Section 4.4.3. Interestingly, the most prominent attribute in three of these branches (*PAS-constant*, *PAS-kind*, and *PAS-previous*) are all variations on the number of predictions that agree with current prediction. This suggests that "majority vote" might be a good heuristic for choosing predictions, but in practice it does not work because there is seldom a

clear majority. The estimated probability of correctness attached to the rules in Figure 4.9 handle this problem gracefully.

5.2.3 Evaluation

The user evaluation of Familiar described in Chapter 7 provided an opportunity to assess Familiar's inferencing. As each participant performed the iterative tasks, Familiar recorded the predictions made for each parameter. This data was used to test the accuracy of the rules for selecting parameter predictions by comparing them to four other selection criteria: three heuristics and a simulated incremental learning scheme.

The interface used in the evaluation let users select the best pattern themselves (Section 3.3.5); consequently, we consider only parameter predictions here. The evaluation version of Familiar has two additional pattern analysis schemes, not described in Chapter 4: *PAS-ML-1R*, and *PAS-ML-information-gain*. Both are variants of *PAS-ML*, and are described below in Section 5.3.3.

Training and test data

Training data was gathered as the iterative tasks described in Appendix B were performed. Although the training tasks are identical to those in Section 5.2.2, a different classifier was necessary for the evaluation. It is similar to the tree in Figure 5.6, but has additional branches for *PAS-ML-1R* and *PAS-ML-information-gain*, and the remaining branches are different reflecting improvements in the pattern analysis schemes (e.g. bug fixes, new features) and variation in the performance of the training tasks.

The test data was gathered from the users who participated in an evaluation of the Familiar user interface. As they worked, Familiar recorded the predictions made for each parameter. Table 5.2 gives the size of the training and test datasets used in the experiment. There was a wide variety of training tasks, but they were performed only once by a single user; whereas both test tasks were performed by ten users, and some were performed more than once.

The standard machine learning evaluation criterion is to measure a classifier's accuracy on a test dataset. After training, the classifier successfully classifies 97.89% of the training instances and 96.83% of the test instances. In this case the accuracy statistic is not very informative. Standard machine learning evaluations use the accuracy to compare different learning techniques, or in situations where

```
PAS-kind = PAS-constant
|   prop-agreeing-predictions <= 0.666667
|   |   PAS-specific-2 <= 3: incorrect (245.0)
|   |   PAS-specific-2 > 3
|   |   |   number-of-iterations > 5: incorrect (59.0/1.0)
|   |   |   number-of-iterations <= 5
|   |   |   |   agreeing-predictions > 0: correct (2.0)
|   |   |   |   agreeing-predictions <= 0
|   |   |   |   |   PAS-specific-1 > 1: correct (2.0)
|   |   |   |   |   PAS-specific-1 <= 1
|   |   |   |   |   |   number-of-iterations > 2: incorrect (14.0)
|   |   |   |   |   |   number-of-iterations <= 2
|   |   |   |   |   |   |   other-predictions <= 0: correct (6.0/2.0)
|   |   |   |   |   |   |   other-predictions > 0
|   |   |   |   |   |   |   |   PAS-specific-2 <= 5: correct (3.0/1.0)
|   |   |   |   |   |   |   |   PAS-specific-2 > 5: incorrect (3.0)
|   prop-agreeing-predictions > 0.666667
|   |   PAS-specific-1 > 3: correct (354.0/1.0)
|   |   PAS-specific-1 <= 3
|   |   |   number-pp <= 0: correct (34.0/1.0)
|   |   |   number-pp > 0
|   |   |   |   other-predictions <= 1: incorrect (6.0/1.0)
|   |   |   |   other-predictions > 1: correct (2.0)
PAS-kind = PAS-extrapolation
|   agreeing-predictions > 0: correct (414.0/3.0)
|   agreeing-predictions <= 0
|   |   other-predictions <= 1: correct (109.0/4.0)
|   |   other-predictions > 1
|   |   |   PAS-specific-2 > 2: incorrect (6.0)
|   |   |   PAS-specific-2 <= 2
|   |   |   |   number-of-iterations > 10: incorrect (2.0)
|   |   |   |   number-of-iterations <= 10
|   |   |   |   |   cycle-length > 2: correct (28.0/2.0)
|   |   |   |   |   cycle-length <= 2
|   |   |   |   |   |   number-pp <= 0: correct (4.0/1.0)
|   |   |   |   |   |   number-pp > 0: incorrect (2.0)
PAS-kind = PAS-ML: correct (69.0/5.0)
PAS-kind = PAS-previous
|   agreeing-predictions > 1: correct (334.0/3.0)
|   agreeing-predictions <= 1
|   |   PAS-specific-1 <= 4: incorrect (11.0)
|   |   PAS-specific-1 > 4: correct (17.0)
PAS-kind = PAS-set
|   proportion-true-pp > 0: correct (91.0)
|   proportion-true-pp <= 0
|   |   agreeing-predictions <= 0: incorrect (51.0/4.0)
|   |   agreeing-predictions > 0: correct (7.0)
```

Figure 5.6 Decision tree for classifying parameter predictions.

improving the accuracy has a direct real-world consequence, such as a decrease in the costs associated with a process. Neither of these conditions apply in Familiar's case, so we propose an alternative metric that takes into account Familiar's ultimate use of the decision tree.

The new metric measures the number of times that Familiar chooses a correct prediction from each set of competing predictions made of the same parameter. As an example, consider a situation where two pattern analysis schemes attempt to predict the direct parameter of a *select* event, one of which is correct and the other incorrect (exactly this scenario is described in Section 4.4.1). For the

	Tasks	Users	Times performed	Instances in dataset
Training dataset	8	1	1	4280
Testing dataset	2	10	1–3	21300

Table 5.2 Evaluation dataset statistics.

purpose of the evaluation, the two predictions competing to predict the *select* parameter are grouped into a set, and Familiar is asked to choose the best prediction. If it chooses the correct one, the *number of correct predictions chosen* statistic is incremented. The purpose of the exercise is to identify selection criteria which maximise the number of correct predictions chosen.

The test data contained 7737 sets of one to six predictions. 7455 sets contained a true prediction (sometimes more than one), and 4381 sets contained only true predictions. These figures represent upper and lower bounds on the number of correct predictions that can be chosen from this dataset: no method can choose fewer than 4381 or more than 7455 correct predictions. These limits are shown in rows 1 and 7 of Table 5.3.

Performance comparison

Three heuristics for selecting predictions were used in earlier versions of Familiar, but later superseded by the machine learning technique. The *random choice* method simply chooses a prediction randomly. The *complexity* heuristic ranks the predictions in each set in a fixed, and rather arbitrary, order of pattern analysis scheme complexity (reflecting the order in which the schemes were implemented) and chooses the most complex. *PAS-constant* is the simplest, followed in order of increasing complexity by *PAS-extrapolation*, *PAS-previous*, *PAS-ML-1R, PAS-ML-C4.*5, *PAS-ML*, and *PAS-set*. The *consecutive true* heuristic chooses the prediction that has been correct for the greatest number of iterations in a row. These methods proved increasingly accurate for selecting correct pattern analysis schemes. Their performance on the test data, measured by the number of correct predictions chosen, is shown in rows 2–4 of Table 5.3.

The number of correct predictions chosen by the rules constructed from the C4.5 decision tree is shown in row 5 of Table 5.3. The machine learning technique makes 151 more correct predictions than the best of the three heuristics, a substantial improvement. Only 55 false instances were chosen when true alternatives were available.

	Selection criteria	Correct predictions chosen	Incorrect predictions chosen
1	lower limit	4381	3356
2	random choice	5936	1801
3	complexity	6792	945
4	consecutive true	7249	488
5	C4.5 rules	7400	337
6	adaptive rules	7430	307
7	upper limit	7455	282

Table 5.3 Comparison of techniques for choosing the best prediction.

Adapting to the user

The C4.5 model is learned off-line from the training data and implemented statically in the Familiar program. A better technique is to use incremental learning to dynamically update the classifier each time a new prediction is made and its accuracy confirmed. Over time, the classifier would adapt to individual users. Similar techniques are used in learning apprentices (Mitchell *et al.*, 1994).

A second experiment simulated incremental learning. An initial C4.5 decision tree is built from the training data and used to choose from the first set of instances. After each set is predicted, the classifier is retrained to incorporate the new examples, thereby modelling a dynamic learner that updates itself after each of the user actions.

The experiment was repeated for each of the users who took part in the evaluation (starting from the original training data each time). The combined results, in row 6 of Table 5.3, show a further increase in performance: over half the remaining incorrect selections with correct alternatives (30 of 55) have been replaced by correct selections. Only 25 false instances were chosen when true alternatives were available.

The improvement appears to be made in situations where the static classifier is unsuited to the user's style of demonstration. In the user evaluation we observed the static decision tree choosing the same incorrect prediction for two or more consecutive iterations despite new demonstrations from the user that showed that the prediction was incorrect. The adaptive version immediately incorporates these corrections into its model, so repeated mistakes occur less often.

The benefits of adapting the model to the user's behaviour seems clear, but there are some caveats. First, only two tasks were considered. Second, the incremental influence of each new instance decreases as the number of training instances

increases, and it may eventually be necessary to weight recent instances more highly. This issue cannot be examined without test data based on long term use, but was encountered by Mitchell *et al.* (1994). Third, the simulated incremental strategy is computationally very expensive. A PBD system must offer instant feedback, and even a few seconds is too long to spend rebuilding the classifier between predictions. A practical implementation will have to use a much faster incremental learner.

5.2.4 Discussion

Familiar successfully uses machine learning to guide prediction at two different points in its inferencing procedure. An evaluation demonstrates that this strategy improves on hand-crafted heuristics, and that incremental algorithms can improve performance still further. The evaluation is practical and shows the real-world utility of the machine learning approach, but has some shortcomings.

First, the evaluation considers only two tasks performed by ten users. A more complete evaluation would necessarily consider a much broader set of user activities and gather more test data.

Second, the composition and execution of the training tasks will affect Familiar's performance on the test data, so there is a risk that the results in Section 5.2.3 are dependent on training task selection. Seven iterative tasks described in Appendix B were used to train the classifiers. The training tasks were chosen because they use a range of applications and test all the pattern analysis schemes, and are different from those used to generate the test data in the user evaluation. Ideally, many more tasks would be used to avoid the risk of overfitting. In practice as many were demonstrated as time and resources allowed, and we speculated that the classifier in Figure 5.4 (lines 6–12) may overfit the training data. A more extensive evaluation is proposed as future work (Section 8.4.2).

A third area for further investigation is the selection of attributes used to build the decision trees and rules. Although the performance in the evaluation suggests that the current sets of attributes are adequate, it is possible that more suitable attributes exist, but were overlooked in this analysis. It is interesting to observe the attributes that were used in the classifiers in the light of this concern.

The sequence recognition tree uses only five of the 12 attributes that were available when the decision tree was constructed. They describe the number of

commands in the event trace that are consistent with the detected pattern, the number of times the pattern has been confirmed correct, and the number of agreeing predictions (Figure 5.4). The pattern analysis tree uses ten of the eighteen available attributes, describing many different aspects of each prediction (Figure 5.6). The number and proportion of agreeing predictions appear frequently, and many of the attributes not used are closely related and describe past performance (Figure 5.5, lines 11–18). In both cases, there is not enough evidence to identify particularly good (or bad) attributes, and it is entirely possible that the most prominent attributes will change if additional training data were to be generated, if different training tasks were used, or if the training tasks were executed by a different user or with fewer (or more) mistakes.

5.3 Learning conditional rules

Familiar's second use of machine learning techniques is to infer conditional rules. Section 3.3.3, for example, describes a scenario where a user sorts a set of files into two folders, one for word processor documents, the other for spreadsheet documents. Section 4.4.3 explains that the *PAS-ML* pattern analysis scheme learns to distinguish between the two cases, and forms a conditional rule that chooses the destination folder for any given file. Section 4.4.3 gives an overview of *PAS-ML* that should be read before the details in this section.

5.3.1 Finding attributes in instance information

PAS-ML learns conditional rules that predict the next value of a parameter. The class value that is predicted is the value of the parameter. The attributes that are used to make the predictions are drawn from the Familiar's instance information (Section 4.2.2), and comprise the data appearing directly in the event trace and the contextual information Familiar gathers when it examines application data.

The first challenge is to massage the contextual data into a format that a machine learning algorithm can use to build a classifier. Consider an example where the user iterates over a set of files and assigns each a label based on its size.[1] Figure 5.7 shows an event trace generated as a user performs this task. There is no

apparent pattern in the recorded sequence of *to* parameter values: 4, 2, 1, 4, 3, 5, 2, and 3.

In this example, the event trace does not provide enough information to predict future values, so a conditional rule is required. The *size* attribute of each file does not occur in the event trace, so the contextual information gathered by the event recorder will be necessary to predict future values. Table 5.4 lists the values of the *to* parameter (the new *label index* values) in Figure 5.7, and some potentially important contextual information. This context includes the *current selection* at the time of the *set* command, the *file type* of the selection, and the *size* of the selection. Contextual information is also provided for event 19, even though it has not yet occurred. Familiar will use the contextual information for event 19 to predict its *to* parameter value using a conditional rule.

It is difficult to choose good contextual information. In a typical PBD problem there are very few training instances and an infinite number of potential attributes. If too little contextual information is considered, a crucial attribute may be omitted, and the correct inference not made. If too much context is retrieved, the learning system will take too long for interactive use and attributes that classify the training data perfectly by chance will be discovered.

Section 4.2.2 described how Familiar gathers potentially useful contextual information about each object it encounters. The inferencing system has all of this context available, but the amount is overwhelming. Consequently, it restricts the contextual information that is added to the training dataset. The contextual information is sorted into the order it was gathered relative to the current event, and the first 20 attributes added to the initial training dataset. An additional 20 attributes are added each time an incorrect prediction is made.

The set of attributes added to the training dataset in the *label index* example is listed in Appendix C. These attributes were gathered in three batches: one when Familiar started inferencing, and two more following incorrect predictions.

[1] In the Macintosh OS, a file can be assigned one of eight labels and will subsequently be displayed in the label colour. In AppleScript, a file's label is accessed through its *label index* property.

1	Activate – Finder
2	select file “a” of folder “letters” of folder “tasks”
3	set label index of selection to 4
4	select file “b” of folder “letters” of folder “tasks”
5	set label index of selection to 2
6	select file “c” of folder “letters” of folder “tasks”
7	set label index of selection to 1
8	select file “d” of folder “letters” of folder “tasks”
9	set label index of selection to 4
10	select file “e” of folder “letters” of folder “tasks”
11	set label index of selection to 3
12	select file “f” of folder “letters” of folder “tasks”
13	set label index of selection to 5
14	select file “g” of folder “letters” of folder “tasks”
15	set label index of selection to 2
16	select file “h” of folder “letters” of folder “tasks”
17	set label index of selection to 3
18	select file “i” of folder “letters” of folder “tasks”

Figure 5.7 An event trace in the Finder.

5.3.2 Learning conditional rules with 1R

Table 5.4 resembles a machine learning problem, where the class is the *to* parameter and the attributes are contextual values. The rows with class values (events 3–17) are training instances, and event 19 is a test instance. Figure 5.8 shows some of the rules that a machine learning system might form to classify the *to* parameter based on events 3–17. Any of these might be applied to attributes of event 19 and used to predict its class.

The first rule, in Figure 5.8a, uses the *current selection* attribute to predict the class, but cannot be used to make a prediction of event 19 because it has no branch for its *current selection* value, *file “i”*. The second rule, in Figure 5.8b, is based on the *file type of selection* attribute, which is *JPEG* at event 19, and using it Familiar might predict that the next value of the *to* parameter is 3. The third rule in Figure 5.8c uses the *size of selection* attribute, currently 128, and using it Familiar predicts that the next label index is 2.

1R

Familiar uses the 1R method to learn conditional rules (Holte, 1993). 1R builds rules based on a single attribute, like those in Figure 5.8. The procedure is very simple: a rule is created corresponding to every attribute, and the one with the highest accuracy is selected and used to classify new examples. Familiar’s implementation uses all the training instances (i.e. it is equivalent to Holte’s 1Rw

event number	*to* parameter	current selection	file type of selection	size of selection
3	4	file "a"	TEXT	64
5	2	file "b"	GIF	128
7	1	file "c"	GIF	32
9	4	file "d"	TEXT	64
11	3	file "e"	JPEG	96
13	5	file "f"	APPL	256
15	2	file "g"	APPL	128
17	3	file "h"	JPEG	96
19	unknown	file "i"	JPEG	128

Table 5.4 Contextual data for the *to* parameter in Figure 5.7.

variant) but differs from the original specification in that it makes no prediction in the default case (whereas Holte predicts the most frequent class value).

The 1R algorithm was used to predict parameter values in the *PAS-ML-1R* pattern analysis scheme mentioned in Section 5.2.3. However, 1R was ultimately unsuitable for PBD problems and is not part of the system described in Chapter 4. It is instructive to consider why.

Problems with 1R

Given the data in Table 5.4, *PAS-ML-1R* will construct the rules in Figure 5.8, then choose the most accurate one and use it to make predictions. In this example, *file type of selection* is 75% accurate—that is to say, the rule in Figure 5.8b classifies 75% of the eight training instances in Table 5.4 correctly. Both *current selection* and *size of selection* are 100% accurate, so 1R chooses one of the rules based on these attributes at random.

In this case the user is assigning labels based on file size, and the *size of selection* rule matches the user's intention. However, the *current selection* rule is equally accurate, so is just as likely to be used by 1R. A close inspection of Table 5.4 shows the *current selection* attribute has a different value in every instance, which means that although the rule is 100% accurate on the training data, it is not likely to be useful for predicting future values. Since each new example has a previously unseen *current selection* value, the rule never has a branch that classifies new instances.

This problem occurs because 1R bases its rule on the attribute with the highest accuracy, but this measure is biased towards attributes with many distinct values. We can avoid it by using a different selection criteria to choose the best rule.

(a)
if *current selection* = *file "a"* then predict 4
else if *current selection* = *file "b"* then predict 2
else if *current selection* = *file "c"* then predict 1
else if *current selection* = *file "d"* then predict 4
else if *current selection* = *file "e"* then predict 3
else if *current selection* = *file "f"* then predict 5
else if *current selection* = *file "g"* then predict 2
else if *current selection* = *file "h"* then predict 3
otherwise make no prediction

(b)
if *file type of selection* = *TEXT* then predict 4
else if *file type of selection* = *GIF* then predict 2
else if *file type of selection* = *JPEG* then predict 3
else if *file type of selection* = *APPL* then predict 5
otherwise make no prediction

(c)
if *size of selection* = 32 then predict 1
else if *size of selection* = 64 then predict 4
else if *size of selection* = 96 then predict 3
else if *size of selection* = 128 then predict 2
else if *size of selection* = 256 then predict 5
otherwise make no prediction

Figure 5.8 Rules for predicting the *to* parameter based on (a) the current selection, (b) the file type of the selection, and (c) the size of the selection.

5.3.3 Permutation tests

PAS-ML disambiguates between noisy and good predictors by replacing the accuracy measure with a permutation test (Good, 1994).

A permutation test is a statistical measure of the probability with which an attribute will classify with a specific accuracy given the distribution of the class and attribute values (which are assumed to be independent). The permutation test is not an estimate; it is an absolute probability, calculated by considering all the possible permutations of class and attribute values for the distribution.

In Table 5.4, the class is the *to* parameter. The class values 2, 3, and 4 appear twice, while 1 and 5 appear once, so the class values have the distribution 2, 2, 2, 1, 1 for classes 2, 3, 4, 1, 5 respectively. Given this distribution, the distribution of an attribute, and the accuracy with which the attribute predicts the class, a permutation test can be used to calculate the probability that the given level of accuracy occurs by chance alone, and to calculate the statistical significance of the combination.

The *current selection* attribute has the distribution 1, 1, 1, 1, 1, 1, 1, 1 because every attribute value (*file "a"*, *file "b"*…) appears exactly once (Table 5.4, *current selection*

column). It classifies the training data with 100% accuracy using the rule in Figure 5.8a. Given this distribution, the probability that a rule based on *event number* will classify the training data with 100% accuracy is 1—no matter what the class values are, a rule based on this attribute is certain to be 100% correct. The statistical significance of the rule is found by subtracting this probability from one, so this result has zero significance.

The *size of selection* attribute has the distribution 2, 2, 2, 1, 1 because three values appear twice (64, 96, 128) and two appear once (32, 256) in the training data (Table 5.4). It predicts the class perfectly (using the rule in Figure 5.8c). The probability that this combination of class and attribute distributions yields 100% accuracy by chance is 0.014; which is statistically significant to the 98.6% level. This probability is computed by calculating every permutation of the class and attribute values consistent with the distribution (in this case there are 861), counting those that are at least as accurate as the observed accuracy (in this case, 12), and dividing the latter number by the former (12/861 = 0.014). A full explanation is available in Frank and Witten (1998).

Permutation tests handle less obvious cases. The *file type of selection* attribute has the distribution 2, 2, 2, 2 because every attribute value appears twice in the training data (Table 5.4). Its accuracy is 75%. The probability of this distribution yielding 75% accuracy by chance is 0.20, so the result is significant up to the 80% level.

PAS-ML

The PAS-ML scheme chooses a rule based on the most significant attribute. In this case, it correctly chooses the rule based on the *size of selection* attribute, because it is statistically more significant (98.6%) than the *file type of selection* (80%) or the *current selection* (0%).

PAS-ML begins making predictions after it has observed at least three iterations of a task and more than one class value. It subsequently makes predictions whenever possible, though early predictions have low significance because there are only four instances and several attributes, and with so little data it is impossible to have a high statistical confidence in the result. The scheme provides the pattern analysis manager with two estimates of its own accuracy: the significance level and accuracy of the chosen rule. These figures may be used

by the pattern analysis manager to estimate how likely it is that the prediction is correct (Section 4.4).

PAS-ML-1R and PAS-ML-information-gain

Section 5.2.3 mentions two other pattern analysis schemes based on machine learning. *PAS-ML-1R* was Familiar's original conditional rule learning algorithm, and formed rules with the 1R algorithm using the accuracy statistic to choose the best parameter. This scheme was unsatisfactory because it was biased towards attributes with a large number of values, as is described in Section 5.3.2. *PAS-ML-information-gain* solved this problem by replacing the 1R accuracy measure with the *information gain ratio* used in C4.5 (Witten & Frank, 2000, p. 95). This scheme performed better than *PAS-ML-1R*, but was discarded because this use of the information gain ratio may not be theoretically sound, and because permutation tests are better suited to small datasets. Consequently, both *PAS-ML-1R* and *PAS-ML-information-gain* have been abandoned and replaced by the *PAS-ML* scheme, which is based on permutation tests.

5.3.4 Discussion

Permutation tests are well-suited to PBD systems because they work well with very small datasets. Most machine learning researchers would consider a dataset of several hundred instances small. The dataset in Table 5.4 has eight training instances. This is large by PBD standards—most users will grow tired of demonstrating a task after giving eight examples—but minuscule compared to other machine learning problems. Datasets with very few instances and a large set of attributes are likely to contain attributes that are good predictors purely by chance. Permutation tests solve this problem by specifically calculating the probability that a rule will yield a given level of accuracy by chance. When there is little data available our confidence in a rule is necessarily low, but as the number of examples grows so does the potential significance of the rule. When there is a lot of data, permutation tests can become computationally demanding, and though this can slow the current implementation of Familiar, efficient algorithms for calculating significance on large datasets have been described (Frank, 2000).

Other machine learning schemes have been used in PBD systems. The Calendar apprentice (Mitchell *et al.*, 1994) and Gamut (McDaniel and Myers, 1998) use the ID3 decision tree learner, a precursor of C4.5. CIMA uses an adaptation of the

PRISM concept learner (Maulsby, 1994). These applications require more complex rules than Familiar because they work in specific domains and have detailed domain knowledge. The algorithms they employ, like those used in Familiar, provide output in the form of rules or trees that can easily be encoded and explained.

Familiar has focussed on rules based on a single attribute because few of the example tasks described in Appendix A require rules based on more than one attribute. Some do however (e.g. the *copy files* task in Appendix A; Maulsby, 1994), and this functionality could be slotted into Familiar in place of the schemes currently used. A practical difficulty is that datasets with a small number of instances and a large number of attributes are even more vulnerable to good predictors that occur purely by chance when rules are formed from combinations of attributes than they are when a single attribute is used. However, permutation tests can ameliorate this problem, as they do in the single attribute case, by calculating the statistical significance of a rule; this is an area for future investigation. The Calendar apprentice does not encounter this problem because it is applied to a single repetitive task with a fixed set of attributes and training data based on long-term use, while Gamut narrows the number of attributes through heuristic search and interaction with the user.

5.4 Summary

Familiar uses machine learning techniques to guide prediction and build conditional rules. Both the sequence recognition and pattern analysis components use a decision tree, trained off-line by C4.5 using data from a set of iterative tasks, to evaluate candidate predictions and choose the best one to display to the user. An evaluation shows that this method is better than three obvious heuristics, but can be improved by incrementally updating the classifier at run-time.

Familiar learns conditional rules by using a machine learning scheme based on 1R to find relationships between the parameter to be predicted and data in the event trace. Problems with the 1R algorithm can be overcome by using a permutation test to select significant attributes instead of the standard accuracy measure. Permutation tests work well with small datasets, so are useful solutions to PBD problems.

6 Platform requirements

This thesis argues that a system-wide, application-independent PBD system for completing iterative tasks in existing applications can be made available to end-users. This chapter explains how demonstrational techniques can be used with existing applications, and how future architectures can better serve this purpose.

The requirements of PBD are seldom satisfied by existing application environments. Architectural support for agents is approached as a research problem and not attempted in commercial situations, and is therefore not available for use by the majority of end-users. Commercial scripting architectures that purport to support agents, such as AppleScript, have shortcomings that present significant obstacles to general-purpose systems. One impediment to their development is that the requirements of PBD are poorly defined: historically, even domain-independent systems have been tightly coupled with prototype applications, and researchers seldom make the information they use explicit. Given this ambiguity, the lack of support is hardly surprising.

The first step towards architectures that support PBD is to specify their requirements. Unfortunately, no two PBD systems are the same, and all contain unique features not present in any of the others. This problem is simplified by identifying specific goals—such as completing iterative tasks—but any set of requirements will inevitably fail to support the speciality of one agent or another. Nevertheless, a basic set of requirements can be posited: users and applications, recordability, controllability, examinability, interface, and consistency.

An architecture that satisfies these requirements can support application-independent PBD for automating iterative tasks. They are met by a variety of platforms, whose idiosyncrasies affect the abilities and useability of agents. Two specific examples demonstrate that demonstrational techniques can be applied in a consistent, system-wide manner: simple macro recorders based on low-level events; and Familiar, described in Chapter 4, which uses high-level events.

Related research indicates that a range of platforms are amenable to system-wide support at intermediate levels of abstraction.

A second way to assess architectural support for PBD is to look at existing systems and ask if they can be supported by existing architectures. Most PBD architectures consist of an event trace generaliser and an interaction manager (Lau and Weld, 1999). The event trace generaliser requires class and instance information about the user's environment, and the interaction manager must know how to interact with the user and other application interfaces. In most systems, this information is provided by the developer, either as an explicit application model or by integrating the PBD system with the application. Neither approach is optimal for existing applications, so the PBD system must attempt to gather information directly from the platform.

A third perspective is offered by the analysis of the shortcomings of specific architectures. AppleScript is a high-level scripting language used in or adapted by a number of PBD systems (Cypher, 1993b, Gaxiola, 1995, Lieberman, 1998). Familiar uses AppleScript as an agent communication language and as a user feedback language, and is itself influenced by AppleScript's design and implementation (Section 3.4). AppleScript is an adequate platform for PBD, but has many weaknesses. Some are common to high-level architectures, some are shortcomings of the AppleScript language, others are problems in specific implementations of the AppleScript standard.

The first section in this chapter describes the requirements of PBD systems. Section 6.2 discusses low-level, mid-level, and high-level architectures, and how they satisfy these requirements. The next two sections discuss the information and access that specific systems require to generalise an event trace (Section 6.3) and implement user interfaces (Section 6.4), and how it can be extracted from existing applications. Section 6.5 analyses AppleScript, highlighting shortcomings that should be addressed by future scripting languages and suggesting guidelines for writing PBD-aware scriptable applications.

6.1 Requirements

A minimal set of technical and non-technical platform requirements must be satisfied before domain-independent PBD for automating iterative tasks is possible in existing applications. These are listed in Table 6.1. Three of the

R1	Users and applications
R2	Recordability: the ability to monitor the user's actions
R3	Controllability: the ability to control application programs
R4	Examinability: the ability to examine application information
R5	A user interface
R6	Consistency

Table 6.1 Platform requirements of domain-independent PBD systems.

requirements pertain directly to applications: the ability to monitor user actions, examine application data, and control the application program. Analogous requirements have been identified for intelligent tutoring systems—observation, inspection, and scripting (Ritter and Koedinger, 1995; Cheikes *et al.*, 1998). User studies that record user actions (Kay and Thomas, 1995; Linton *et al.*, 1999), animated help systems that require complete and exclusive control of the user interface (Bharat and Sukaviriya, 1993; Miura and Tanaka, 1998a), and attachable, application-independent tools (Olsen *et al.*, 1999) make similar demands.

Requirement 1: Users and applications

The principal justification for adding a demonstrational interface to existing applications and environments is that end users know them and are disinclined to alter their habits, so the most basic requirements are non-technical: a set of applications and a group of existing users. Both are met by any successful commercial computer platform, but not by prototypes and research systems. If there is no established user base, it will be easier and more effective to rewrite an application using an architecture like Amulet (Myers and Kosbie, 1996) or AIDE (Piernot and Yvon, 1993) that is designed to support programming by demonstration.

Requirement 2: Recordability

For most kinds of programming by demonstration it is necessary to monitor and record the user's actions. Each action is recorded, added to the event trace or command history, and analysed to infer the user's intent and predict subsequent actions. An ideal recording mechanism will be unobtrusive, so that users can demonstrate tasks under conditions identical to their standard working environment, and detailed enough to support reasoning about the user's intent.

Requirement 3: Controllability

To carry out tasks on the user's behalf, a programming by demonstration system must be able to control other applications. This can be accomplished either through application programming interfaces or by emulating the user (Lieberman's (1998) "marionette strings"). The latter is a natural choice because it allows a learned task to be performed in the same way that it was demonstrated by the user.

Requirement 4: Examinability

Information about the application is necessary to infer intent and make predictions. Such information falls into two categories: class information and instance information. The former describes the capabilities of an application, including the commands and objects it uses. The latter describes the data the user is working on and the commands they have executed recently. PBD systems require the ability to examine each application's instance information.

Requirement 5: User interface

Any demonstrational interface must interact with the user. Interaction design is especially challenging for systems that work with existing applications or with multiple applications. Few existing applications are designed to have truly extensible interfaces, and PBD must work around these limitations. Multiple-application systems face a trade-off between consistency and the benefit obtained from domain-specific interaction techniques (Section 3.2.4).

Requirement 6: Consistency

In practice, application independence requires that each application satisfies the technical requirements in the same way, so that a single system can work with every application, represent tasks that span applications, work with unseen applications, and present a single interface to the user.

6.2 Command architectures

This section discusses low-level, mid-level, and high-level commands, and the ways that they can be used to record user actions, control applications, and communicate with the user.

An agent can control an application in two ways: by issuing instructions through an application programming interface, or by emulating the user. Applications with programming interfaces can be asked to execute programs or scripts, allowing users to automate specialised tasks using the functionality built into an application. Most applications make no provision for scripting, but can be manipulated by agents that emulate user actions. Lieberman (1998) calls this a "marionette strings" approach, drawing a parallel between an agent manipulating an application and a puppeteer manipulating a marionette. As a rule, end-user programming languages use application programming interfaces, and macro recorders use marionette strings. Familiar uses a hybrid of the two when it sends AppleScript commands to an application.

Marionette strings are a natural approach in PBD because a program is learned by watching what the user does, and it intuitively follows that the system should perform the task as it was taught. It is more difficult to provide feedback if the system infers the user's intention from a demonstration, but elects to solve the problem in some other way. However, there are several problems with marionette strings. Most computer systems are designed for exactly one user who generates a single stream of input events, and an agent will interfere with the user's activities if it uses the machine at the same time (Olsen *et al.*, 1998). It is not easy to choose the level of abstraction and granularity of the event protocol (Lieberman, 1998). Finally, marionette strings cannot always be used to examine an application's data (Lieberman, 1998).

6.2.1 Low-level events

Low-level events represent user actions in their simplest form and correspond directly to specific machine inputs. They encode the user's physical actions, such as key-presses, mouse movements, and mouse clicks, without knowing the effect of those actions. Consequently they are easy to record and synthesise, but difficult to generalise. Macro recorders based on low-level events are available for most platforms; their operation is described in Section 2.4.1.

The variability of low-level event traces makes them difficult to relate to application data and search for regularities. Figure 6.1 shows an excerpt from a low-level event trace in a macro recorder (MJT Net Ltd., 1998). Although many extraneous mouse movements have been removed from the trace, it is difficult to tell what the user is doing, or even what program they are using. In fact, the user

1	MouseMove>313,157	13	Wait>1.26
2	Wait>0.15	14	Release Shift
3	LDown	15	MouseMove>313,157
4	Wait>0.07	16	Wait>1.38
5	LUp	17	Send Character/Text>e
6	MouseMove>313,157	18	MouseMove>313,157
7	Wait>0.96	19	Wait>1.56
8	Press Shift	20	Send Character/Text>d
9	MouseMove>313,157	21	MouseMove>313,157
10	Wait>1.17	22	Wait>3.37
11	Send Character/Text>w	23	Press Return
12	MouseMove>313,157	24	Wait>1.35

Figure 6.1 A low–level event trace.

is working in a spreadsheet application, and has selected a cell by moving the mouse over it and clicking on it (lines 1–5), typed *Wed* (lines 8–20), and pressed the Return key (line 23). If the user were now to type *Wed* in a second cell, the sequence of low-level events would be different, reflecting variations in the screen position of the cell, the user's typing speed, and random factors like spelling mistakes.

6.2.2 High-level events

High-level events describe the user's actions in an abstract form that omits details of how each operation is performed, making them easier to interpret and search for repetition. A high-level event is realised by a sequence of lower-level events. The low-level event trace in Figure 6.1, for example, is equivalent to two high-level events in Microsoft Excel (version 5):

Select Range "R2C1"
set FormulaR1C1 of ActiveCell to "Wed"

If the user typed *Wed* into a second cell, the high-level events recorded would be almost identical to those for the first cell, but the low-level event trace would contain many variations.

High-level events are difficult to record. They require that applications comply with standard protocols for inter-process communication. The costs of compliance mean that applications based on high-level events are not common, and are limited to situations the programmer has anticipated and thinks are important (Olsen *et al.*, 1999). Some architectures support high-level control and examinability, but are not recordable. Two other problems with high-level event

```
 1  tell application "Microsoft Excel"
 2      activate
 3      Create New Workbook
 4      set FormulaR1C1 of ActiveCell to "January"
 5      Select Range "R1C2"
 6      set FormulaR1C1 of ActiveCell to "February"
 7      Select Range "R1C3"
 8      set FormulaR1C1 of ActiveCell to "March"
 9      Select Range "R1C4"
10      set FormulaR1C1 of ActiveCell to "April"
11      Select Range "R1C5"
12      set FormulaR1C1 of ActiveCell to "May"
13      Select Range "R1C6"
14      set FormulaR1C1 of ActiveCell to "June"
15  end tell
```

Figure 6.2 A high-level AppleScript event trace recorded in Microsoft Excel.

architectures—the data description problem and input that does not match a high-level event—are discussed in Section 6.5.2.

AppleScript is an example of an application-independent high-level language (Apple Computer, 1993–1999). Figure 6.2 shows AppleScript commands for entering data in a row of spreadsheet cells. Familiar uses AppleScript, demonstrating that high-level events can be used to make PBD available in existing applications. AppleScript is discussed in detail in Section 6.5.

Some application-specific macro recorders record high-level commands. For example, Figure 6.3 shows a macro for finding the references section of a document in Microsoft Word in the Visual Basic language (Microsoft Corporation, 1992–1998). This is not a system-wide solution, and is difficult for third-party systems to exploit, though it has been used to monitor users (Linton *et al.*, 1998). ActiveX Scripting Host is a new, system-wide scripting architecture for Microsoft Windows (Microsoft Corporation, 1996; Box, 1997). It is similar to AppleScript but cannot (currently) monitor the user.

6.2.3 Mid-level events

It is convenient to discuss high-level and low-level events, but in reality these are the ends of a spectrum, not a simple dichotomy. User actions can be represented in intermediate forms: to say that a user clicked on a button labelled “Okay” is more meaningful than saying that they clicked at position (50,128), but does not explain the user’s purpose as well as a high-level event.

```
1  Sub gotoreferences()
2  '
3  ' gotoreferences Macro
4  ' Macro recorded 2/7/99 by Gordon Paynter
5  '
6   Selection.Find.ClearFormatting
7   Selection.Find.Style = ActiveDocument.Styles("Heading 2")
8   With Selection.Find
9   .Text = "References"
10  .Replacement.Text = ""
11  .Forward = True
12  .Wrap = wdFindContinue
13  .Format = True
14  .MatchCase = True
15  .MatchWholeWord = False
16  .MatchWildcards = False
17  .MatchSoundsLike = False
18  .MatchAllWordForms = False
19  End With
20  Selection.Find.Execute
21  End Sub
```

Figure 6.3 A macro recorded in Microsoft Word in the Visual Basic language.

Figure 6.4 shows a mid-level event trace recorded by a system called WOSIT, described below (Geier, 1999). It shows the events that occur when an application, called “pizza-tool”, starts (lines 1–3), when the user presses a toggle button (line 4), when the user presses a standard button labelled *Drinks* (line 5) causing the system to open a window with the same name (line 6), and when the user presses a button labelled *OK* in the *Drinks* window (line 7) and the window is closed (line 8). As this example illustrates, mid-level events convey information more concisely and with less variation than low-level events (Figure 6.1) but are not as succinct and descriptive as high-level events (Figure 6.2).

Many of the user’s interactions with graphical applications are performed through standard user interface features like menus, buttons, text fields, and dialog boxes. When they are part of a system-wide user interface library, and are dynamically linked by applications at run-time, they form a bottleneck where mid-level interaction can be recorded and synthesised. This bottleneck allows recordability, control, and consistency on platforms with existing users and applications, but it only provides superficial examinability. Internal application data is inaccessible through mid-level interaction: only the user interface components are examinable.

```
1  application(start, "pizza-tool", 487);
2  window(open, "pizza-tool", 487, main);
3  application(ready, "pizza-tool", 487, Widgets=27);
4  toggle(press, "Extra Cheese", "pizza-tool", user, 1513, "on");
5  button(press, "Drinks", "pizza-tool", user, 1602);
6  window(open, "Drinks", 1602, visible);
7  button(press, "OK", "Drinks", user, 1718);
8  window(close, "Drinks", 1718);
```

Figure 6.4 A mid-level event trace recorded by WOSIT (adapted from Geier, 1999, pp. 16–17).

Mid-level events have been used to facilitate PBD in applications that use specialised interface toolkits (Myers, 1998); and to monitor, control, and examine unmodified applications in intelligent tutoring systems (Cheikes *et al.*, 1998), animated help systems (Bharat and Sukaviriya, 1993), and interface attachments (Olsen *et al.*, 1999). They are implemented in virtual windowing systems like X Windows (Schiefler and Gettys, 1990), window managers like Gnome (Mason and Wheeler, 1999), and virtual machine environments like Java (Arnold and Gosling, 1996), many of which have freely available source code or specifications.

The Topaz system adds mid-level "scripting by demonstration" to any graphical application developed with the Amulet interface toolkit (Myers, 1998). Users create scripts by selecting the important steps from an event trace generated while the user interacts with the graphical objects in the system. Figure 6.5 shows the event trace recorded by Topaz when the user changes the colour of three drawing objects (lines 1–3), clears a fourth object (line 4), then resizes the first three objects (lines 5–7). Topaz also uses the interface toolkit to control and examine applications. For example, Topaz automatically generates *Search for* dialogs that let user find graphical objects based on their attributes at run time. The developer does not have to write extra code for these operations; they are generated by Topaz from the Amulet toolkit.

Topaz demonstrates that mid-level access is sufficient to add PBD to applications, but is only applicable to applications that use the Amulet toolkit. However, we can make Topaz-like PBD available in existing applications on any platform that allows mid-level recordability, examinability, and controlability through system-wide interface libraries. The remainder of this section describes existing systems that gain such access on two platforms: the X-windows windowing environment, and the Java virtual machine environment.

```
1  Change color <Sel_Polygon_Proto_2663> = Ax_Blue
2  Change color <Sel_Polygon_Proto_2679> = Ax_Red
3  Change color <Sel_Polygon_Proto_2696> = Ax_Yellow
4  Clear <Sel_Polygon_Proto_25276> = 1
5  Grow <Sel_Polygon_Proto_2663> = {50,100,270,251} u.r
6  Grow <Sel_Polygon_Proto_2679> = {60,120,260,241} u.r
7  Grow <Sel_Polygon_Proto_2696> = {70,120,270,251} u.r
```

Figure 6.5 A mid-level event trace recorded by Topaz (adapted from Myers, 1998, p. 537).

Accordingly, it is possible to make domain-independent PBD available in existing applications on these platforms.

The Widget Observation Simulation Inspection Tool (WOSIT) is a software tool for recording, inspecting, and controlling unmodified UNIX applications in the X-windows windowing environment (Cheikes *et al.*, 1998; Geier, 1999).[1] It was used to generate the event trace in Figure 6.4. WOSIT works by replacing the dynamically linked window library, which imposes a limitation on the applications—those few that use non-standard or statically-linked libraries are incompatible. Bharat *et al.* (1995) describe another system-wide approach to synthesising and capturing lower-level actions in X-windows. Their animated "help by demonstration" requires complete and exclusive control of the user interface, and they describe a mechanism for controlling (and potentially recording) existing applications (Bharat and Sukaviriya, 1993, Bharat *et al.*, 1995). Interestingly, they suggest that recorded and synthesised actions are a potential security risk.

Several projects implement, or aim to implement, the requirements of PBD in the Java virtual machine environment. JOSIT brings the basic functionality of WOSIT to Java using the accessibility API and accessibility utilities, which provide notification of user interface and model events (Geier, forthcoming). Olsen *et al.* (1999) build attachments to manipulate the surface representation of programs with few or no assumptions about the way the application is programmed. They build upon techniques that customise the output—and to a lesser extent the input—of unmodified Java applications at run-time (Edwards *et al.*, 1997). The Jedemo framework provides demonstrational help in Java applets (Miura and

[1] WOSIT (version 1.1) implements recording and inspection. Control of the user interface is currently limited. Full support is planned and designed, but the implementation is stalled for funding reasons (Marty Geier, personal communication).

Tanaka, 1998a, 1998b). The Jedemo recorder lets the applet developer record a series of commands, which the Jedemo player can play back in the applet, with a mock mouse pointer and annotations provided by the developer. Jedemo is a Java applet runner that can run any Java applet, and is itself implemented as a Java applet. It can be nested inside any other applet runner (including those in web browsers) and made transparently available to every applet user. Jedemo needs an initialisation method, so is not applicable to existing (compiled) applications. It is unclear whether this compromise was necessary or simply convenient.

6.2.4 Event hierarchies

An agent need not be limited to interpreting a single level of events. Every high-level event comprises mid-level events, which in turn comprise low-level events. These low-level actions are not examined in isolation because such great detail is seldom necessary and can conceal the user's intentions, but can provide useful information that is lost through abstraction.

The command history can be stored as a tree rather than a flat trace of events. In this model, low-level events are grouped to form higher-level events, which can in turn be grouped into yet higher-level events. The hierarchy offers a range of improvements over a "flat" event trace. User actions that do not correspond directly to a high-level event can be described with lower-level events. A hierarchy can be abstracted to a higher level than high-level events (Zeiliger and Kosbie, 1997; Piernot and Yvon, 1993). Other potential advantages include multi-level undo, improved pattern matching, improved error recovery, and exploiting low-level context (Kosbie and Myers, 1993b; Piernot and Yvon, 1995).

Hierarchical event systems can be implemented in a bottom-up fashion by aggregating low-level events into higher-level events. Low-level events are easy to monitor, but it is not always clear how to compose them into higher-level events and gather contextual information. Zeiliger and Kosbie (1997) use this technique to record a hierarchical event trace in unmodified Microsoft Windows applications. Low-level and mid-level events are captured, and an external model of the application is used to enrich them with contextual information and aggregate them into higher-level events, which may in turn be composed into higher-level events. Ultimately, the low-level event trace is generalised into a more manageable high-level event trace. This approach combines the relative

ease of trapping low-level events with the simplicity high-level events, at the expense of modelling the application.

Other hierarchical command architectures are available as toolkits for application developers (Piernot and Yvon, 1995; Kosbie and Myers, 1996). These appear to yield to a better quality of event trace, and correspondingly better generalisation, but require developer cooperation and cannot be applied to existing applications.

6.2.5 Alternatives to command architectures

An alternative to the command-driven PBD systems described above is to learn the user's task solely by examining an application's data. Once the agent has learned the task, it must then control the application to complete it, though it is less unintuitive in this context to use an application programming interface (as opposed to marionette strings). Tatlin attempts to learn programs by examining the application data, ignoring the user's commands (Gaxiola, 1995; Lieberman, 1998). When it discovers that data has changed it searches for regularities, attempts to infer a transformation from the original data to the new, and writes a program to propagate the changes in a systematic way. The inferred program is not based directly on user actions, so Tatlin is free to effect changes using techniques other than those in the demonstration. Similar but more specialised techniques are used to manage text styles in Tourmaline (Myers, 1990).

6.3 Detailed inferencing requirements

PBD uses inferencing to learn about a user or task, and later apply what is learned to the user's advantage. The amount and style of learning varies greatly, but every system requires some information—or "knowledge"—to learn from.

This section describes the information used in PBD systems and how it can be added to or extracted from existing applications. A discussion of application knowledge makes up the bulk of the section. User, task, and background information are discussed, but they are not dependent on the platform, so are not greatly affected by it.

6.3.1 Application knowledge

PBD systems require an intimate knowledge of the applications they manipulate.

Most PBD systems are based on commands and objects. Commands are the actions that the user can take in the user interface, and objects are the data the user works on. Application data may not be internally represented in an object-oriented fashion, but it is convenient for the PBD system to model it in this way.

A complete model of an application consists of class (or type) information and instance information (or data). The former describes the capabilities of an application, including the commands and objects it uses, and remains substantially unchanged from one invocation to the next. The latter describes the data the user is working on and the commands they have executed recently. It differs each time the program is run, and often changes in response to user actions. Consider a simple task where the user deletes the files in a folder that are larger than some size threshold. The PBD system needs instance information about the size of each deleted file, the other files in the folder, and the size of these other files. To gather this information, the system needs class information including the fact that files have a size attribute, the possibility that file size is the basis of selection, and the commands for retrieving files and the attributes.

6.3.2 Sources of class information

Class information is most important in high-level architectures where commands describe the user's intention and explicitly reference application objects. Each application has a range of functions and objects, and each can be described in a unique way, so the number of different high-level artefacts can easily run into the hundreds in practice, and in principle is unbounded. The four applications in Table 4.2, for example, have a total of 237 commands and 167 classes. Mid-level and low-level event architectures typically have only a few commands. The low-level event trace in Figure 6.1 is limited to *MouseMove*, *Wait*, *Ldown*, *Lup*, and a few other commands; and Bharat *et al* (1995) identify exactly nine system level events in X windows. The class information in these systems can be represented in its entirety by a relatively small model.

External application models

PBD systems that are decoupled from their applications typically use an "external" model of the application that describes how it may be controlled and examined. Low-level and mid-level models can be complete and succinct. High-level models are typically incomplete, expensive to build, and biased towards to the interests of the builder (e.g. Cypher, 1993b; Gaxiola, 1995). Zeiliger and

Kosbie (1997) acknowledge these issues by providing tools for model builders, exploiting partial models, and statistically identifying the most fruitful parts of an application to model.

Application-supplied models

In the AppleScript architecture, each application provides a dictionary of the commands that the user can issue in that application, and the classes of information that are used. Familiar reads this dictionary and uses this information as the basis of its class knowledge.

External models are usually of higher quality than those gleaned from applications; however, a model generated from the application requires less effort to build and provides better (and unbiased) coverage of the application. A hybrid of the two approaches combines the best of each: a basic model from the application can be refined by a human expert.

Learning from the event trace

High-level instance information can be used to infer class information.

AppleScript recording is intended to let users generate code by demonstration for educational purposes, and so that they can use it in their own scripts. Familiar was originally designed to learn class information in the same way: it began with no application knowledge, but was able to record and parse AppleScript. When the user began demonstrating a task it would observe their actions and update its model of the application by generalising from the instances in the recording to the classes they represent.

Figure 6.6 shows the event trace recorded as the user selects each file in a folder and sets its label. The agent learns from the first recorded command that the Finder has an *activate* command. From the second, *select file "apple" of folder "fruit"*, it learns that the Finder has a *select* command, that the *select* command has a single parameter, that there are object classes called *file* and *folder*, that *folders* can contain *files*, and that instances of both classes are identified by a string. As well as this type information, it learns that two specific instances—*file "apple"* and *folder "fruit"*—exist and may be relevant to the task. The next command, *set label index of selection to 3*, adds more class information, including the existence of a *set* command with a *to* parameter, and a *selection* object with a property called *label index*. The agent's mental model of the application becomes more complex and complete as more actions are recorded, and it can start using

```
1  Activate
2  select file "apple" of folder "fruit"
3  set label index of selection to 3
4  select file "carrot" of folder "fruit"
5  set label index of selection to 2
6  select file "banana" of folder "fruit"
7  set label index of selection to 3
```

Figure 6.6 A simple iterative task recorded in AppleScript.

the model to examine the application (Section 6.3.3). If it records another reference to the *selection*, for example, it can send an AppleScript command to the application to get the *label index* property of the new *selection*.

Although it is agreeable to think that a PBD system might learn about its environment by demonstration, this approach has significant shortcomings. The first problem is that some commands cannot be parsed correctly without knowing the syntax in advance. For example the command *set label index of selection to 3* might be interpreted as a *set* command with optional parameters *label* and *to*, which in turn implies that *index* is a property of the *selection* object. Parsing the selection is another problem: references to the keyword *selection* apply indirectly to the selected object, which can have a different class in each command. Finally, the application cannot build a complete model of an object unless all its attributes occur explicitly in the event trace. In Figure 6.6, for example, the user is setting the *label index* of each file based on the *size* attribute, but the *size* attribute can never occur in an event trace (because the user cannot affect the size of a file directly) so the system does not know about it and is unable to explain the users behaviour.[2]

Zeiliger and Kosbie (1997) use event traces to build models of application commands. Their approach differs from the one described in that human experts and specialised tools construct the models, which are built off-line.

6.3.3 Sources of instance information

Agents require instance information, particularly the commands they have used and the objects present, to infer the user's future actions. Information about an application's data is usually only available at a high level of abstraction. In some

[2] An early version of Familiar prompted the user for attribute names, but this solution is unreliable and too complex for end users.

circumstances data is inaccessible, but user interface components can be examined. Olsen *et al.* (1999) call this the "surface representation" of a program, and suggest that it lets an agent see the application as the user does, which can be more relevant than knowledge of internal representations that are hidden from the user. Low-level architectures yield little instance information because recordings like the one in Figure 6.1 cannot be related to application data.

The event trace

The primary source of instance information is the event trace. When the user is observed to perform an action on an object, both the action and the object enter the agent's mental model of the application. They are added to the event trace, and used to infer the user's intent.

Contextual information

Contextual information about a command or object are details that describe it, but which are not necessary to replicate it. In Figure 6.6, each file is identified by its *name* and *folder*, but a PBD system needs contextual information to complete the task. Pursuit, for example, considers the *name*, *date*, *size*, and *owner* of each file (Modugno and Myers, 1997). This context provides enough information to infer a relationship between a file's size and the label it is assigned.

A number of PBD systems use specific contextual information. Such information is chosen by the developer based on their knowledge of the application and the tasks the user is likely to perform in the application. For example, the choice of file attributes in Pursuit was based on an informal survey of Unix users (Modugno and Myers, 1997). Contextual information is either embedded in each user action (e.g. Cypher, 1993b) or gathered immediately before or after a user action is recorded (e.g. Piernot and Yvon, 1993).

There are two practical problems with contextual information. First, the developer must choose good context. Correctly anticipating all the potentially related data that might explain an action is both difficult, as developers may not even predict the eventual uses of their application, and time-consuming, as contextual information must be added to every command and object.[3] Second,

[3] As an example, the domain knowledge for the first, application-specific version of Eager consists of sixteen densely filled A4 pages of Lisp code in ten point font, and builds a partial model of HyperCard (Allen Cypher, Personal communication).

the developer must add the contextual information to the recorded events. This is a technically difficult problem when working with existing applications. In Eager, for example, the application and operating system were rewritten to this end,[4] and AIDE requires that the developer build contextual information into the application (Piernot and Yvon, 1993). Zeiliger and Kosbie (1997) draw context from information always visible in interface components, such as the names of windows, and do not need to add this information explicitly. Though the data is limited, useful information (e.g. file names in save dialogs) is available.

Examinability

The problems with specific contextual information are choosing the correct contextual information, and adding it to the event trace. They can be overcome by examining application data at run-time and dynamically extracting the data relevant to its task. Identifying useful context then becomes a problem for the PBD system, not the application developer.

Application data can be examined directly, for example by allowing the agent a pointer to an object in the memory of the application; or indirectly by letting the agent send queries to an application that identify the object by describing it. The former technique is faster and allows the agent to manipulate the application's data directly, but the security risks are much greater and it may be impossible to implement on platforms with protected memory. Direct access to underlying data structures is problematic, but the surface representation is sometimes accessible (Myers, 1998, Olsen *et al.*, 1999). Indirect access is safer, but slower, and can be complicated by the data description problem (Section 6.5.2). Familiar gathers contextual data indirectly through AppleScript.

6.3.4 User and task knowledge

Information about the kinds of task a user is likely to perform in an application, or has performed before in an application, is a potential learning aid. Like "domain knowledge", neither user nor task knowledge is clearly defined, and they often manifest themselves as heuristics or bias towards particular situations. Some demonstrational interfaces can adapt their behaviour to specific users,

[4] Cypher (1993b) notes that HyperCard was modified (p. 209) and describes a version of AppleScript recording that was not available outside Apple (p. 212). It was implemented

particularly learning apprentices (Mitchell *et al.*, 1994) and Familiar (Section 5.3). Designing a PBD system with user or task knowledge is no harder for existing applications than for research prototypes because the information about each user is handled independently of the applications and application knowledge.

6.3.5 Background knowledge

Background knowledge represents the information an agent has about the world that is relevant to many applications, tasks, or users. It can be explicitly represented in data structures, or embedded in algorithms and defaults by the programmer. Background knowledge is simple to add to any PBD system and is independent of the platform and other knowledge sources.

6.4 Detailed user interface requirements

Requirement 5 acknowledges that PBD systems work with the user and require a user interface. This section surveys common PBD interaction techniques, and considers the information and services necessary to support them in an system-wide, application-independent fashion. Some problems, like storing and editing programs, require little integration with applications, so are not discussed in this section.

6.4.1 Start recording, stop recording

Although the user demonstrates a program in the application interface, they must still signal what and when they are demonstrating (Section 3.2.4). The user might signal the start or end of a demonstration, or that a demonstration has taken (or is taking) place. A system that allows the user's actions to be monitored can implement any or all of these styles.

An alternative is to allow the user to retroactively designate a series of actions as a demonstration. The primary constraint on an agent that continuously monitors the users actions is that it must be unobtrusive. Familiar only monitors the task at the user's request for two reasons: some applications behave differently when AppleScript recording is activated, and its additional event processing and

by Ed Lai at Apple Computer, but not released. It is not compatible with current versions of AppleScript (Ed Lai, personal communication).

memory use can affect the user's interaction with other applications, particularly when it is consuming resources looking for regularities that do not exist.

6.4.2 Feedback language

Most PBD systems use a feedback language to explain the system's actions and inferencing. These languages are predominantly visual, because visual languages are thought to be easier for inexperienced users to learn, and so that graphical objects can be represented exactly as they appear in an application. Some are integrated with specific application interfaces, and are difficult to apply to existing applications. Techniques like anticipation highlighting, comic strips, and visual programming languages require a degree of application cooperation. Less sophisticated interfaces, using programming languages, natural-language descriptions, scripting by example, and standard interface components are less integrated with, and less dependent on, the application.

Some systems do not attempt to explain actions to the user, and consequently do not need a feedback language. An example is the dynamic macro interface for text editing, where predictions are not described, they are simply performed in the application interface (Masui and Nakayama, 1994). The user then has the option of accepting, rejecting (changing), or undoing the prediction. This style of interaction is potentially applicable to other domains, but depends on a complete and reliable undo mechanism.

6.4.3 Low-level and artificial syntax

User actions recorded in an internal, machine-readable format can be translated directly into a crude but human-readable form. The effect is to make the semantic content of a command, or aggregate effect of a sequence of commands, unclear. Figure 6.1 illustrates this problem in a low-level event trace, though the effect is also visible in the mid-level trace in Figure 6.5, and the high-level trace in Figure 6.3. In each case the meaning of the commands is obscured by unnecessary detail and the way they are presented.

High-level commands and objects are occasionally described in a programming language. The Visual Basic macro in Figure 6.3, for example, was recorded in Microsoft Word and simplified (by removing errors and exploratory actions) in the programming environment provided (Microsoft Corporation, 1992–1999).

This is not always ideal for PBD systems. Non-programmers will find the artificial syntax and control structures difficult to understand.

6.4.4 English-like text and natural language syntax

Natural language syntax is an attractive alternative to artificial programming languages. HyperTalk and AppleScript are examples of programming languages that are designed to read like English text. Well-designed commands read as well-formed imperative sentences and are comprehensible to novice users with little understanding of programming or programming languages (Section 6.5.1; Simone, 1995). Some scripting architectures, including AppleScript (Apple, 1992) and Active Scripting Host (Microsoft Corporation, 1996), offer a choice of program representations, some of which are more readable than others. It follows that other high-level program representations can be transformed into more readable forms.

6.4.5 Graphically representing objects

One of the strengths of PBD is that abstraction, including abstraction from an object to its representation in a program, is minimised. In text-based languages like AppleScript an object is seldom represented in a program as it appears on the screen. More complex visual languages like those in SmallStar (Halbert, 1993) and Pursuit (Modugno, 1997) are able to represent objects in a similar format to the application itself, and are therefore easily recognised by the user. Figure 6.7 shows how a file called *apple* is represented in the style of Familiar, SmallStar, and Pursuit.

Some data does not have a convenient and distinctive graphical representation. The properties of objects, and ubiquitous elements like words in a document or empty cells in a spreadsheet, are examples. In some cases icons and text can be interleaved to create slightly more abstract representations. Figure 6.7 also shows how the *size* property of the a file named *apple* is represented in the three systems. None correspond directly to the user interface because the concept of *file size* is never pictured in the operating system.

Implementing distinctive visual representations in an application-independent manner is difficult unless there is some way to get an image of the object in question. The simplest method is to use images that the developer has provided, like the icon property in the Finder. More complex—but less

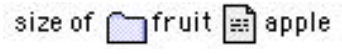
Language	a file object named *apple*	the size of the file object
Familiar (AppleScript)	file "apple" of folder "fruit"	size of file "apple" of folder "fruit"
SmallStar		
Pursuit	 apple	 apple

Figure 6.7 Representations of a *file* object and its size property following the style of Familiar, SmallStar, and Pursuit.

reliable—descriptions can be built from knowledge of the application domain, for example by finding the attributes of an object (such as a spreadsheet row, column, format, and value) and generating a representation using application knowledge; or by finding the location of an object on a screen and taking a snapshot of it.

6.4.6 Graphically representing commands

User actions are more difficult to represent graphically than objects. It is unclear which aspect of the action should be depicted: a symbol representing the semantic meaning of the action (like the scissor icon for the *cut* command), some depiction of the result of the command (such as a tick being added to a checkbox), or the physical actions the user took to perform the command (like moving and clicking the mouse). The latter case is complicated because a high-level command might be invoked in a variety of ways, each of which is represented differently. It is, however, the method of choice in animated help interfaces (Bharat *et al.*, 1995).

Most high-level languages describe commands using succinct keywords like *open*, *move*, and *copy*. Some graphical languages augment these textual descriptions with graphical depictions of the objects as they appeared before or after the command (e.g. Halbert, 1993, the comic strip metaphor); their requirements are discussed in the previous section. Others are more ambitious. Pursuit illustrates the transformation effected by changing depictions and using colour to express continuity, requires an external model of how objects look and how applications transform data (Modugno and Myers, 1997). Every instruction in Cocoa is a graphical rewrite rule consisting of images before and after a

command takes effect (Smith *et al.*, 1994). Mondrian builds an icon depicting the input and output of a transformation (Lieberman, 1993). These systems are embedded in applications, and require a level of integration that is difficult to achieve with existing applications.

6.4.7 Anticipation and highlighting

Anticipation is a user interface technique for describing predicted actions to the user (Cypher, 1993b; Schlimmer and Hermans, 1993). Eager uses it to show the predicted next action by highlighting the relevant interface element, sometimes with additional descriptive information (Cypher, 1993b). Highlighted actions correspond to mid-level events, like clicking on a button, selecting from a menu, or typing a string. This is a very concrete way to describe a prediction: actions are displayed in the standard interface, so little user abstraction is required. However, it is not suitable for describing more than one command at any given time, so cannot be used to represent an entire iteration—or a single high-level event in some cases—in a static form.

Anticipation highlighting is tightly integrated with the application interface, and is often incompatible with existing applications. Eager uses a detailed model of HyperCard, and exploits the fact it is fully scriptable. Most applications do not allow external agents to use their interface so freely, so this approach is not universally applicable. A promising alternative is to add anticipation by altering the display functions in standard user interface toolkits (Edwards *et al.*, 1997; Geier, 1999).

6.4.8 Guide objects and ghost objects

Gamut gives feedback by creating and changing objects in the application interface (McDaniel and Myers, 1999). *Temporal ghosts* are translucent copies of objects as they appeared in the recent past. *Guide objects* are objects the programmer creates to illustrate geometrical relationships, similar to those used in Metamouse (Witten and Maulsby, 1993). Both are used to give the system hints about its inferencing using *hint highlighting*, a special form of selection. In existing applications, these techniques pose two problems: first creating copies of application objects and inserting them into the interface; second colouring and highlighting them. Both require application integration beyond what is currently available, though Olsen *et al.* (1998) consider the highlighting problem.

6.4.9 Forms and dialog boxes

Many PBD systems use forms and dialog boxes to solicit feedback from the user. These dialogs are a familiar interface element and can retrieve specific information in a controlled manner, but can be intrusive and interrupt the user.

Metamouse uses dialog boxes to seek guidance about its learning during a demonstration (Maulsby and Witten, 1993). For example, it might interrupt a demonstration to ask why a user has placed a line in a particular place, and let the user choose a response from a number of likely alternatives. When it detects repetition it first asks the user if it should take over; it then asks for confirmation that each step is predicted correctly. Pursuit generates its own dialog boxes to ask the user how it should automatically respond to system dialog boxes describing errors so that a task can be automated without human input (Modugno and Myers, 1997). Other systems use dialog boxes to get object and command parameters. SmallStar uses editable “data description sheets” to let the user explicitly generalise an object based on its parameters (Halbert, 1993), and other systems use similar dialog boxes to perform searches, thus facilitating set iteration. These can be built by the developer (Piernot and Yvon, 1993; Kurlander, 1993) or generated when the program is run (Myers, 1998).

Dialog boxes, particularly modal dialog boxes, are relatively simple to implement in existing applications because they overlap the standard user interface.

6.4.10 Buttons, menus, and speech

The Gamut interface construction tool allows the user to “nudge” the system when it is not behaving correctly (McDaniel and Myers, 1993). These nudges are implemented through buttons labelled *Do something* and *Stop that*. Adding buttons to an existing applications is difficult when they are embedded in the interface, though buttons on floating windows are straightforward.

Maulsby (1993) observed that users like to give hints—ambiguous partial instructions. His CIMA system allows the user to provide several kinds of instruction (e.g. *I want this* or *Ignore this*) by selecting text and then using a pull-down menu. Like dialog boxes, pull-down menus require little integration with applications and can be implemented by placing an external window over the top of the application or by calling interface routines common to any application.

Familiar, for example, takes advantage of the standard Macintosh menubar to place its menu in every application.

CIMA simulates hints extracted from the user's speech (Maulsby, 1994). Speech recognition software is not in widespread use, so it is difficult to say how much control an agent will have, and how instructions to an agent might interfere with instructions to an application. Spoken input is not built into most user interfaces, and we can speculate that it will be available to a PBD system without interfering with the application. Whether this is true of software that is primarily voice controlled—such as dictation software—remains to be seen.

6.5 Case study: AppleScript

AppleScript is an application-independent high-level scripting language for the Apple Macintosh (Apple Computer, 1993–1999). Familiar uses AppleScript to record the user's actions, and to examine and control target applications. This section discusses the weaknesses—and strengths—of AppleScript and their effect on the development of Familiar.

The Open Scripting Architecture (OSA) is a standard for communication between scripting languages and Macintosh applications: any compatible language can be used to control any compatible application. The default language of the OSA (for English-language versions of the operating system) is the English dialect of AppleScript. Other languages are available, such as UserLand Software's UserTalk (UserLand Software, 1992), but the use of the default is so prevalent that the OSA is generally referred to as AppleScript, and will be here.

Familiar was first used with AppleScript version 1.1.2 for the Macintosh OS versions 7.5–8.1, and has since been used with AppleScript version 1.3.7 for the Macintosh OS version 8.6. All of the problems reported were encountered in earlier versions of the software; some are fixed in the later.

The next section considers the strengths of AppleScript, and explains why it is used in Familiar. The three subsequent sections discuss the weaknesses of AppleScript as a basis for programming by demonstration. The first considers the problems inherent in using a high-level event model (Section 6.5.2); the second considers the language itself (Section 6.5.3); the third considers its various implementations (Section 6.5.4). A range of miscellaneous issues are canvassed in Section 6.5.5.

6.5.1 Why AppleScript is used in Familiar

Familiar is implemented in AppleScript because it satisfies the requirements in Section 6.1 and its high level of abstraction is an attractive programming choice.

Requirements

AppleScript has a large base of existing users and applications (Requirement R1). The language can be used to control scriptable applications (R3) and examine their data (R4). Some (but not all) scriptable applications are also recordable, allowing Familiar to record the user's actions (R2). Because AppleScript is an Apple standard, its syntax is consistent across applications (R6). Finally, AppleScript's natural-language syntax offers a convenient feedback language for the user interface (R5).

English-like feedback language

AppleScript uses a natural language syntax, relieving the PBD system of the need to implement a second program representation (such as anticipation highlighting or a visual language) to describe actions to the user.

Users can usually comprehend the English-like syntax of instructions like *open folder "fruit"* or *select file "apple"*, particularly when the command describes a recently-performed action, even if they are unable to formulate such expressions themselves. Thimbleby et al. (1992) examine a similar language, HyperTalk, and observe that its syntax "makes reading scripts very easy", is "easily and quickly mastered", "encourages experimentation", and "doesn't feel the same as learning one of the 'hard' programming languages like C". They also identify disadvantages. The large number of ways in which an idea can be expressed in English means that the programmer is frequently unsure of the syntax of a particular command, a problem exacerbated by the many exceptions and inconsistencies of the English language (and the HyperTalk implementation). This is seldom an issue in PBD because the system generates code for the user, who need only understand and confirm it. Expressing complex programming concepts in the English language, and therefore in English-like programming languages, can be difficult. Examples include data structures, variable names, object identifiers, iteration and decision constructs, and any detailed operation. Most of these artefacts do not arise in PBD systems, which build straight-line programs, do not use data structures, and give specific examples in place of abstractions like variables.

Application dictionaries

The AppleScript programming language itself has little functionality. It provides program control structures (loops, conditionals, statement blocks), common data types (numbers, strings, lists), interprocess communication, and few other application-independent commands. Scriptable applications extend the language by adding their own commands.

Every scriptable application maintains a description of its commands, object classes, and enumerations. This information is stored in the application dictionary (or, more formally, the AppleEvent Terminology Extension, or AETE resource) and can be read by other applications. Figure 6.8 shows the Finder (version 8.1) dictionary entry for the class *disk*. The plural form (line 2) provides an equivalent name for *disk* that is more readable in some circumstances. Elements (lines 4–20) are classes that a *disk* can contain, such as *folders* and *files*. Properties (lines 21–26) are attributes (or slots) of the objects that can be accessed with AppleScript, and whose values can be basic types like integers, strings and lists, or complex objects like files or windows. The properties shown are all read-only (r/o), and have a brief comment describing their purpose. Inheritance relationships are not shown in the figure, though the *disk* class inherits 23 properties from its superclass *item*, including *name*, *icon*, *id*, and *size*; and others from its superclass *container*. As well as the classes, the Finder dictionary contains commands, like the *set* command in Figure 6.9. Each command has a name, and may have a *direct* parameter (line 2) and a set of named parameters (line 3). Parameters are typed, though many have type *anything* (any data item) or *reference* (any reference to an object), and can be optional or required.

The application dictionary benefits PBD systems in a number of ways. An agent can quickly model an application by retrieving the dictionary, even if that application has never before been encountered. The dictionary information is structured, so it can be treated as data and is easily stored and manipulated. It is consistently structured, so a single parser and compiler can be used with any application, and tasks that span applications are easily automated.

6.5.2 Problems with high-level event architectures

AppleScript uses high-level events. Though this is usually advantageous (Section 6.2.2), it can introduce problems that less abstract architectures do not suffer.

```
 1 Class disk: A disk
 2 Plural form: disks
 4 Elements:
 5     accessory suitcase      by numeric index, by name
 6     alias file              by numeric index, by name
 7     application file        by numeric index, by name
 8     clipping                by numeric index, by name
 9     container               by numeric index, by name
10     control panel           by numeric index, by name
11     desk accessory file     by numeric index, by name
12     document file           by numeric index, by name
13     file                    by numeric index, by name
14     folder                  by numeric index, by ID, by name
15     font file               by numeric index, by name
16     font suitcase           by numeric index, by name
17     item                    by numeric index, by name
18     sharable container      by numeric index, by name
19     sound file              by numeric index, by name
20     suitcase                by numeric index, by name
21 Properties:
22     capacity        integer     [r/o]  –  the total number of bytes
                                          (free or used) on the disk
23     ejectable       boolean     [r/o]  –  Can the media be ejected
                                          (floppies, CD's, and so on)?
24     free space      integer     [r/o]  –  the number of free bytes left on the disk
25     local volume    boolean     [r/o]  –  Is the media a local volume
                                          (as opposed to a file server)?
26     startup         boolean     [r/o]  –  Is this disk the boot disk?
```

Figure 6.8 The Finder (version 8.1) dictionary entry for a *disk* object.

Data description

Data description problems arise because application objects are identified by a description, not accessed directly (by pointers to memory, for example). If an object changes so that it no longer matches its prior description, then it can no longer be accessed by an external system. Further, the description format is chosen by the application developer and may be inappropriate for the purpose of a particular agent. Generating a data description is a well-known problem in the PBD literature (Halbert, 1993) and is by no means specific to AppleScript.

Interaction does not match a high-level command

Some user actions do not correspond directly to high-level events. For example, the Finder allows the user to select and copy part of a file name, but this action is not described by an AppleScript command from the Finder dictionary. This deficiency might be addressed by extending the Finder dictionary to cover selection in text fields, but taken to its logical extreme this solution would add

```
1  set: Set an object's data
2      set reference -- the object to change
3      to anything -- the new value
```

Figure 6.9 The Finder (version 8.1) dictionary entry for the *set* command.

every variation on every command to the dictionary, exploding its size and sacrificing the abstraction of high-level events. A further ambiguity is the role of navigation commands that do not affect data—do they represent significant user actions? Kosbie and Myers (1993) suggest that high-level events correspond to actions that the user might wish to undo.

6.5.3 Problems with the AppleScript language

AppleScript was designed primarily for human users and has several shortcomings as an agent communication language—shortcomings that should not be confused with poor implementations of the language in recordable applications. However, AppleScript is implemented voluntarily by each application developer, so it is difficult to require them to adhere to standards—at best they can be encouraged.

Speed

The Familiar evaluation showed that AppleScript version 1.1.2 was not fast enough for interactive use (Section 7.1). The consequences are more far-reaching than sluggish response: the user operates in real time and if the agent does not react quickly the opportunity for prediction passes. Familiar can and does fall behind the user's demonstration, and though it does not interfere with user interaction, it may offer predictions too late and retrieve data that is stale.

Familiar has recently been tested on a 400 Mhz Macintosh G3, with version 8.6 of the Macintosh operating system and AppleScript version 1.3.7. The performance increase appears sufficient to address the speed problem, though it has not been tested by end users. This configuration differs from the development and evaluation environment, a 200 Mhz Power Macintosh 7300, in several ways. AppleScript version 1.1.2 is written for Motorola 68000 microprocessors, which must be emulated by Power PC-based systems, impeding their performance. Version 1.3.7 is written for the Power PC and run in native mode (though it relies on the AppleEvent library, which is emulated). It is unclear how much of the

speed increase is due to the G3's faster architecture, how much is due to its faster clock speed, and how much is due to the new AppleScript version.

Timing

Applications report the user's actions after they are performed, and agents have no access to data that the actions have overwritten. If the user changes a word to a boldface font, for example, the change will be recorded after it occurs and the agent will have no access to the original style. Various solutions to this problem have been proposed, but most introduce new timing problems.

Apple's Human Interface Guidelines (Apple Computer, 1992) recommend that interaction be structured so the user first selects an object (noun) and then applies some action (verb), a style that is reflected in Familiar's *select-set* cycles. An agent, upon recording the *select* command, can immediately examine the relevant object. In practice, however, the two events are reported almost simultaneously, so the agent cannot examine the selection before the subsequent command. In the Finder, for example, the select event is recorded only if it is followed by some action on the selection, whereupon both events are reported at once.

Cypher (1993b) describes implementations of recording that let the agent examine the HyperCard application before and after each event occurred. This solution is unreleased and incompatible with standard applications and the operating system (Section 6.3.3). It introduces another potential timing problem: an agent could fall behind the demonstration, forcing the application to suspend interaction with the user while it responds to the agent.

Single user assumption

Applications treat commands from agents as though they were user actions, and this causes difficulty when agents activate applications, make selections, or use the clipboard at the same time as the user. Each of these operations involves some kind of global variable—the *frontmost* application, the *selection*, the *clipboard*—that cannot be shared by the user and agent. An agent working in the Finder, for example, might select a file and then delete the selection. If the user was also working in the Finder and selected a different file immediately after the agent, the agent might delete the user's selection without apparent error.

The problem can be solved by letting AppleScript monopolise the machine (or the applications. Another solution is to construct scripts that avoid global variables like the selection and the clipboard. This approach is possible, and

represents good programming style, but is not suitable for PBD because the learned task will not reflect the user's demonstration. Ideally, every agent, whether human or machine, would have its own selection (Olsen *et al.* 1998).

Examinability

Access to the data the that user is working on is one of the strengths of high-level event systems. Familiar regularly uses AppleScript to access data in other applications, but is hampered by its inability to traverse object hierarchies at run-time and its inability to find all the properties of specific objects.

Every application object is stored in a hierarchy (*rectangle "X" of canvas "Y" of project "Z"* for example) but it is often impossible to traverse this hierarchy at run-time. It may be important for an agent to know, for example, what else is contained in *canvas "Y"*. A standard solution is to send the command *get every rectangle in canvas "Y"*, and for the application to return a list of *rectangle* objects. This is unsatisfactory because it is recommended, not required, by the AppleScript specification, and many applications do not implement it. In practice, it is difficult to tell whether or not an application (or class) supports the *get every* syntax. Further, it says nothing about objects that are not *rectangles* that are contained by *canvas "Y"*. A related problem is finding the class and properties of an object. Some object instances do not belong to a specific class, poorly match their class definitions, or have unclear class definitions. In these cases it is difficult to find the properties of an object (Section 6.5.4).

There are general solutions to these problems. The simplest is to enforce the use of the *get every* syntax and appropriate inheritance information or of more general *get contents* and *get properties* commands. These commands are generally omitted because they are an extra expense, technically difficult, and unlikely to benefit many users.

An additional complication is that, unlike every other command, *get* is built into the AppleScript parser and its definition cannot be overridden by the application.[5] This is a problem for application developers who wish to extend it in unique ways, but may be an advantage for agents because it ensures that access to an application's data is through *get* commands with a consistent syntax (the command templates in Figure 4.5 exploit this consistency).

[5] Chris Espinosa, AppleScript Implementors mailing list, December 2, 1999.

Spatial representation in English text

Textual languages are often inadequate for describing graphical data. For example, a command like *set position of selection to {65,0}* describes a position on the screen exactly, but is difficult to translate into a pixel location. Given application knowledge it is possible to represent this information graphically, but consistency with the rest of the language is sacrificed. This is one of several shortcomings of English-like programming languages (Thimbleby *et al.*, 1992).

Persistence of objects and references

AppleScript objects used in commands are not required to exist after the command is executed. If you store a reference, it may be invalid or identify a different object when it is reused. For example, when recording scripts in the Scriptable Text Editor, documents are identified by index number, where the frontmost is document 1, the next document 2, and so on. The AppleScript command to bring the rearmost of two open documents to the front is *select document 2*, but as soon as this command is executed the indexes of the two documents are exchanged and any future references to *document 2* in fact affect the former *document 1*.

Undo

AppleScript has inconsistent support for *Undo* and *Redo* commands. Although many applications support these functions, they do so in an inconsistent manner and cannot be relied upon. As a result, agents are unable to undo their actions.

6.5.4 Problems with AppleScript implementations

Many problems with AppleScript implementations can be traced to the developer's assumption that recording will be used by humans rather than agents. Others are the inevitable result of ignoring basic design principles (Simone, 1995).

Syntax

AppleScript syntax is not always well-chosen. One of our example tasks uses Microsoft Excel to enter the formula *=AVERAGE(B1:B50)* in cell C1 (Appendix A.1). The relevant AppleScript recorded is:

```
Select Range "R1C3"
set FormulaR1C1 of ActiveCell to "=AVERAGE(RC[-1]:R[49]C[-1])"
```

This exemplifies several problems. First, the user selects a single *Cell*, but the recording describes it initially as a *Range*, then as *ActiveCell*. It is not clear why the *FormulaR1C1* property is used to describe the contents of the selected cells, or precisely what *FormulaR1C1* means.[6] The formula itself is the aspect of this trace most likely to confound the user. Every *Cell* has a *FormulaR1C1* property, used in the trace, and a *Formula* property, which contains the formula as the user sees it. The latter representation is simpler and more closely reflects the user's actions.

An agent can use domain knowledge to make specific and consistent cosmetic changes when displaying commands to the user; for example, the string *ActiveCell* might be replaced with *selected cell*, and the *FormulaR1C1* label and value might be replaced with those of *Formula*. These solution requires an external model of the application and—more seriously—might impede the user's education if they learned to program AppleScript by traditional techniques.

Recordings don't match actions

The single largest problem with AppleScript recording is that the recording does not always reflect the actions the user has performed. This confuses agents that rely on AppleScript recording to monitor the user's actions, and misleads the user about the syntax of the language and the effect of commands. Two specific problems occur in the applications used with Familiar: recording extraneous commands, and failing to record commands. A third problem, discussed in Section 6.5.2, is that some user actions do not correspond to high-level events. This is a problem with high-level event architectures that is not specific to AppleScript, but it may indicate that the application's dictionary is poorly designed.

The formatting commands in Microsoft Excel (version 5) misrepresent the user's actions by adding commands. When the user sets the alignment of a cell to *center*, four commands are recorded:

set HorizontalAlignment of Selection to xlCenter
set VerticalAlignment of Selection to xlBottom
set WrapText of Selection to false
set Orientation of Selection to xlHorizontal

6 *FormulaR1C1* is the formula in an internal notation that makes relative references clear. The suffix *R1C1* indicates the formula is relative to row 1 and column 1 of the spreadsheet.

There are a number of possible explanations for the extra commands. They may be a side-effect of reusing code, or may reflect what actually happens in Excel when a *center* command is given. AppleScript recording was originally designed to let users see code that worked, so they may be intended to expose the user to more of the syntax.

Some applications do not record all the user's actions. Examples include Netscape Navigator (version 3), which does not record the user clicking on a hyperlink; text editors that do not record text manipulation actions; and many minor Finder functions. The developers might have felt that these actions were unimportant, would not be used frequently, were unlikely to be required in scripts, or were too difficult to report.

In some cases commands are not recorded as a matter of policy. An example is selection in the Finder. In Microsoft Excel, a *select* event is recorded every time the user selects a cell, even if they then select another cell without altering the first one. This can result in a series of consecutive *select* commands as the user navigates. In the Finder a select command is only recorded if an action is subsequently performed on the selected object. Although the Excel style is probably better for an agent—the agent can ignore the extra commands it does not use, but it cannot restore those it never receives—the Finder style uses less system resources and is easier for a human reader to comprehend. Lieberman (1998) suggests that the agent and application negotiate an appropriate protocol in situations like these; in theory an agent could request different levels of verbosity from a recordable application.

Application behaviour changes during recording

One of the main advantages of PBD over other forms of programming is that programs are demonstrated in the familiar user interface. Unfortunately, some applications behave differently when recording than they do normally. This impairs the user's ability to demonstrate the program as they would in normal circumstances. An example is Microsoft Word (version 6, 98), which disables the mouse when recording is turned on.

Incompletely specified objects

Often objects are incorrectly described in the dictionary. This occurs most frequently, and most problematically, with object inheritance. Although AppleScript provides a facility for defining inheritance, it is often ignored, even

when inheritance is in fact used. An example is the Finder (Version 7–8.1), where, for example, *alias files* are subclasses of *files*, and *files* are subclasses of *items*, but no inheritance relationships are specified. Though they may be intuitively obvious to a person, agents have great difficulties with these omissions. A similar problem occurs when only some instances of a class inherit properties of a superclass. In the Finder (version 7.6), some instances of the *disk* class inherit the properties of the *sharable container* class and some do not.

There is a historical reason for these problems. In early versions of AppleScript the application dictionary had a size limit, and the inheritance notation was introduced as a space-saving shorthand—it did not (and does not) necessarily reflect the actual internal structure of the application. As the dictionary was intended to be read by users, not agents, developers could depend on human intuition to infer the inheritance relationships.

Errors (often fatal)

AppleScript recording is often used as a learning aid, rather than a basis for generating and executing code. As a result, many application developers neglect to test it extensively, and many AppleScript implementations contain serious errors. Some, like the missing *set* command in the Fetch (version 3.0.3) dictionary, can be worked around. Others, like the *Resize* command in GIFConverter (version 2.4d18), which hangs the machine when recorded and played back, cannot be repaired, only avoided.

Errors such as this can be fixed in subsequent releases. Developers would be much more conscious of them if recording were more widely used.

6.5.5 Other issues

This section discusses miscellaneous issues like the purpose of AppleScript implementations, the lack of recordable applications, and the use of the “suite” model as a concrete specification of “domain knowledge”.

Purpose of recording

AppleScript recording was intended to let end-user programmers generate code to edit and reuse, not for the purposes of user monitoring, and many implementations remain true to the original purpose. Their primary motivation is to display working code, not to describe the user’s actions; consequently the recorded actions do not always reflect what the user has done. This explains

some of the poor design choices (from a user monitoring perspective) in Section 6.5.4.

Lack of recordable applications

A significant problem with AppleScript as a platform for PBD is the shortage of recordable applications. Those that are available are often unsuitable because of the quality or style of the scripting implementation. This lack can be attributed to a combination of technical, economic, and political factors.

The technical difficulty is simply that building scriptable applications is hard, and there have been few tools and examples, a combination that forces every developer to implement their own scripting. Some applications, particularly those written before AppleScript was available, are implemented with programming languages or paradigms that are incompatible with scriptability. Both of these problems increase the cost of writing scriptable and recordable applications, an economic problem compounded by the relatively small population of users who find scriptability and recordability useful, though niche markets—notably desktop publishing—do exist. Finally, Apple released AppleScript in, 1992, but did not release a significantly revised version of the product until, 1998. Many developers perceived this neglect as evidence that AppleScript was not an important technology and might not be supported in later releases of the operating system.

Domain knowledge through suites

AppleScript dictionaries are arranged into suites of related commands and objects (Apple Computer Inc., 1992). There are a number of standard suites, each of which contain standard commands and objects. For example, the *standard* suite contains commands like *set*, *get*, *make*, and *open*; and objects like *window*, *text*, and *file*. Other suites include the *text* and *table* suites. Developers can use these suites as the basis for their own dictionaries by implementing the commands as they appear in the suites, or by extending and overriding their parameters.

Suites might be viewed as application-independent domain knowledge (Section 2.1). The information in a particular dictionary represents application knowledge, so suite information is potential domain information that is shared by every application in the domain.

6.5.6 Guidelines for PBD-aware scriptable application

This section contributes practical advice on how to design a scripting implementation that supports PBD systems and other agents.

The first step is to design the scripting implementation well for people. An implementation will be no good to a PBD system if it is no good to a human user. Simone's (1995) "human scriptability guidelines" codify his advice to application developers on how to implement scriptable applications for human users. They are summarised below. More detail can be found in Simone's *According to script* column in *Develop* magazine (Apple Computer Inc., 1990–1997).

Some implementations support human needs admirably, but agents poorly. The main problem areas are recordability and data description. A set of "agent scriptability guidelines" are included below; following these guidelines will make a scriptable application compatible with Familiar and with other agents.

Human scriptability guidelines (from Simone)

Designing the object model

- Decide which objects to include in the model. They should represent the objects the user thinks about when working with the application.
- Think from actions to objects: objects should be designed to support the actions the user is likely to want to perform.
- Start with menu commands. Menu commands should be scriptable, but the scripting implementation should not be limited to menu commands.
- Make early blueprints. Write down the user's commands as real sentences, and build a prototype AETE resource (application dictionary) to parse them.
- Make the containment hierarchy obvious. Make it easy for the user to determine it, and ensure that every object is connected.

Assembling the application dictionary

- Use standard terms whenever possible, but do not use them with non-standard meanings, and do not vary the terms in standard suites (i.e. command and object groupings).
- Use extended terms to express concepts unique to your applications, but keep in mind the style of what has been done before. Creating new object classes

or properties is usually preferable to creating new commands. If you are adding a lot of vocabulary, place it in a separate suite.

Stylistic conventions

- Begin terms with lowercase letters.
- Separate multiple-word terms with spaces. For example, *transfer protocol* is preferable to *TransferProtocol.*
- Use familiar terms, but avoid reserved words.

Enumerations, lists, records, and type definitions

- Use lots of enumerations. (Simone's other guidelines in this category are very detailed and only occasionally applicable.)

Direct objects

- Explicitly identify the direct object. Do not create commands that operate on some default target.
- Make the target of the command the direct parameter.
- Help users identify the objects used with each command.

Other tips and tricks

- Some classes might best be implemented as properties, and *vice versa*. Many objects can be represented as properties of other objects more naturally than as classes in their own right. However, turning too many classes into properties can confuse the object model hierarchy.
- Use inheritance to shrink your AETE.
- Be cautious when reusing type codes.
- Avoid using *is* in property and parameter names. For example, a property called *is selected* can cause confusion when used with the *is* operator.
- Control the number of parameters. Long lists of command parameters lead to long dictionary entries and unwieldy sentences. Instead, the command might accept a list of enumerators, or be split into two or more commands.
- Make sure that commands that return values give meaningful results.

Agent scriptability guidelines

First, design the system well for users, as is described above. Syntax that is good for a user is good for an agent, because agents aim to understand and emulate the user. The following guidelines should be followed in addition to Simone's.

AppleScript recording

- Implement recording. Ensure the application's behaviour does not change when recording is activated.
- Ensure that recorded actions describe user actions. Do not record "extra" commands, and do not use internal notation. Recorded commands should both read well and parse easily, so use simple terms and handle each parameter separately. Avoid property lists—if you have too many parameters, make heavier use of defaults or create new commands. Do not record internal information—if you want to instruct programmers, then write documentation.
- Ensure that every significant user action is recorded. Unfortunately, it is difficult to judge what might be a significant action. PBD is especially appropriate for tasks that developers fail to anticipate, so commands that seem obscure may prove important to users. Navigation commands are problematic; it is usually a good idea to merge consecutive navigation commands unless the application is some form of browser or the commands have side-effects on data.

Data definitions and access

- Specify every class completely. Do not omit inheritance relationships. Agents have little intuition, so include every detail in the dictionary.
- Use persistent references. Avoid references that become stale or expire.
- Make every property of an object accessible through the *get* command, including inherited properties. If an property does not apply or exist for some object, then return a null object or list. Do not return an error—errors should be used to signal a mistake by the user or agent, not to compensate for the shortcomings of your design.
- Support the *get every* syntax universally. Every containee element of a class should share a superclass so that it is possible to retrieve the contents of an object with a single command.

6.6 Summary

This chapter lists the platform requirements of PBD systems and explains the extent to which they are met. Two domain-independent systems make PBD available in existing applications: macro recorders, which work through low-level events; and Familiar, which exploits a high-level scripting architecture. Related research is used to demonstrate that mid-level events meet the requirements of PBD in several existing architectures, which are consequently viable platforms for PBD.

The quality of a PBD system is affected by the application knowledge and interface services available. These are limited by the platform, which is why PBD systems are so often implemented on research environments, not end-user environments. Generally, higher-level architectures meet the detailed requirements best, but are uncommon, while lower-level architectures are more accessible but often lack access to application information.

AppleScript is an adequate platform for PBD, but has a number of deficiencies. Most of these can be attributed to the fact that the language was designed to be attractive to end users, not for agent communication.

7 Evaluation

The thesis statement in Section 1.2 asserts that PBD can be made available in existing applications and used to automate iterative tasks by end users. This chapter establishes the second part of this claim: that given an appropriate PBD system, end users can use it to automate iteration. It then considers the circumstances in which users will choose to use PBD in preference to alternative tools, and the types of task that Familiar and other PBD systems can automate.

This chapter describes two experiments. The first, a user evaluation, establishes that end users are capable of using PBD to automate iteration, but finds that they often choose alternative techniques when they are available, and occasionally attempt tasks that are beyond the abilities of the Familiar system. The second experiment explores the extent to which PBD can be useful by compiling a set of iterative tasks and examining how each can be solved with Familiar, with an improved PBD system, and with improved applications.

The user evaluation had several objectives. The first, and most important, was to test the hypothesis that end users are capable of automating iterative tasks with PBD. The second was to find out whether end users will choose to use Familiar in two sets of circumstances: when the alternative is performing a task by hand, and when other automation tools are available. A final goal was to gather feedback from end users about the Familiar interface.

The subjects in the evaluation were asked to solve two iterative tasks, then introduced to Familiar, and finally asked to complete two much larger tasks. Ability to use PBD was tested by observing the subjects ability to use Familiar. All were able to do so with little training. Tool preference was assessed by observing whether the subjects chose to invoke Familiar or existing automation tools based on multiple selection. Most subjects elected to use the existing tools when they were available.

The subjects' comments on Familiar contributed to the design in Chapter 3. The interface had previously only been evaluated informally and by few users, who were predominantly computer science researchers. Less experienced users had fewer preconceptions and were able to proffer novel advice.

An interesting observation made during the experiment was that some subjects were not able to gauge the extent of Familiar's abilities, and often attempted to use it to solve subtasks that were beyond its abilities. For example, one spreadsheet user taught Familiar to copy a block of seven cells containing the days of the week, and expected it to know how to split the block into two groups and paste them in different locations when it reached the end of a month. Given the typical user's high expectations of agents (Section 3.2.3), attempting impossible tasks is a potentially significant problem. One way to measure it is to estimate the range of tasks that can and cannot be automated with PBD.

The task evaluation explores Familiar's limitations with reference to a set of twenty repetitive tasks collected from interviews with users, the PBD literature, and other sources (Appendix A). The evaluation, a Gedanken (thought) experiment, asks which of these tasks Familiar can automate. Seventeen of the examples are iterative tasks. Familiar, as described in Chapters 3–5, was implemented far enough for use in the user evaluation, and can automate some but not all of the seventeen examples. Two problems prevent the remainder from being completed: a lack of recordable applications, and a lack of inferencing capabilities. These issues are considered separately. First, we ask what would need to be added to the applications available on the Macintosh before a hypothetical perfect PBD system could automate the example tasks. Next, we assume that these improvements have been made, and ask which tasks Familiar is capable of learning. The tasks are divided into those that can be learned by the current version of Familiar, those that might be learned with a better implementation of the inferencing model described in Chapter 4, and those that might be learned by some other hypothetical system. This exercise reveals that less than half of the tasks can be learned by the current version of the software, but almost all could be if Familiar received a small set of modifications.

Section 7.1 describes the user evaluation, and Section 7.2 the task evaluation. In each case the experimental procedure is first described, then an overview of the observations and conclusions, and finally the detailed results.

7.1 User evaluation

Familiar was evaluated by a group of end users in August 1998. The evaluation considered two hypotheses, the second of which is conditional on the first:

- End users are capable of using PBD to automate iteration
- End users will choose to use PBD to automate iteration

The first hypothesis was tested by asking users to complete an extensive iterative task that could not be automated with conventional tools. There were two variations of this scenario. In the first, the experimenter artificially prevented the use of multiple selection, and thus of conventional automation techniques. In the second, the user was asked to complete a task that could not be automated with conventional tools. In each case, subjects faced the choice of performing the task manually or using Familiar in circumstances where the former is too time-consuming to be a realistic solution. If the subjects consistently used Familiar and successfully automated the task, we conclude that end users are capable of automating iteration with PBD.

The second hypothesis contends that end users will choose to use PBD. It was tested by asking the subjects to perform a large task with no restrictions on their actions: they had been introduced to Familiar, but other tools (based on multiple selection) were also available. If the subjects consistently chose Familiar over the alternatives, we conclude that they will choose to use PBD to automate iteration.

7.1.1 Procedure

Table 7.1 summarises the experimental procedure. The body of the experiment asked each subject to attempt four variations of two iterative tasks, the *calendar* task and the *image conversion* task. Before the tasks, the subjects were asked to complete a questionnaire (Appendix D.1) describing their experience with computers and applications; afterwards they were interviewed about their tool choice, their opinion of Familiar, and their experience with macro recorders (Appendix D.6).

Variants 1 and 2 introduced the subjects to the applications and to the iterative nature of the tasks. The first variant asked the subjects to perform each task as if they were doing it for themselves, revealing their natural behaviour in iterative situations. The second asked them to repeat the task without using multiple

Step	Task	Variant	Size	Summary of Instructions
1	Calendar	1	225 cells 449 chars	Create calendar as if doing it for yourself
2	Calendar	2	225 cells 449 chars	Create calendar without using multiple selection
3	Image	1	12 images	Convert images as if doing it for yourself
4	Image	2	12 images	Convert images without using multiple selection
5				Work through the Familiar tutorial
6	Calendar	3	221 cells 645 chars	Create calendar without multiple selection, but with Familiar available
7	Calendar	4	221 cells 645 chars	Create calendar with both multiple selection and Familiar available
8	Image	3	63 images	Convert images without multiple selection, but with Familiar available
9	Image	4	63 images	Convert images with both multiple selection and Familiar available

Table 7.1 Experimental procedure for the user evaluation.

selection, forcing them to confront the iteration problem. The subjects had little choice but to perform variant 2 manually; the tasks were designed so that it was possible, though tedious, to complete them in the time available.

After a short break, the subjects were asked to work through the Familiar tutorial reproduced in Appendix D, then to repeat each task twice more. The third variant, like the second, prohibited the use of multiple selection, but allowed the use of Familiar. The size of the data sets was increased so that manual execution was extremely tedious. The third variant was a *de facto* test of whether a user is able to use Familiar to automate iteration, and is used to test the first hypothesis.

The fourth variant, like the first, placed no restrictions on the subject, but in it the subject knew about Familiar. This variant measured the extent to which users will choose PBD over other alternatives, thus testing the second hypothesis.

The tasks were deliberately ordered so that the subject was introduced to the applications (if they do not know them already) before they encountered Familiar (Section 7.1.6). The tasks are performed in mutually exclusive domains, so application functionality learned one is unlikely to be applicable to the other.

Figure 7.1 A subject completing the *image conversion* task with Familiar.

The iterative tasks

In the *calendar* task, the subject was asked to duplicate a printed calendar as a Microsoft Excel spreadsheet. The task is described in Appendix A.2, and the instructions are reproduced in Appendix D.4.

The calendar used in the first two variants comprised 449 characters in 225 cells. In the third and fourth variations the calendar was altered to counter learning effects and to make the task longer; the number of cells in this version decreased to 221, but the number of characters increased to 645. In its simplest form, the *calendar* task involved iterations of two high-level events, repeated on 225 (or 221) cells; in practice the subjects performed many noise actions and used tools to automate the task.

The *image conversion* task spanned three programs: the Finder, the Fetch FTP client, and the GIFConverter image manipulation program. In the first two variants, the subject was asked to use a graphical FTP client to download twelve files from an FTP site onto the hard drive; to use an image manipulation program to resize them; to use the FTP client to return the modified files to the FTP site; and to use the operating system to clean up any copies left on the local machine. The last two variants used 63 images instead of 12, and instead of resizing each, the subject converted them from one graphic format to another. The task is described in Section 1.3.1 and more formally in Appendix A.6; the instructions given to the subjects are reproduced in Appendix D.5.

Approximately 15 high-level events must be performed to convert each of the images. Figure 7.1 shows a how one subject taught Familiar to automate variant three of the *image conversion* task with Familiar; Figure 3.11 shows another subject completing a subtask of variant 3.

Domain experience		Application experience	
Spreadsheet	10	Excel	10
Graphical user interface	10	Finder	6
FTP client	4	Fetch	3
Image manipulation	5	GIFConverter	5

Table 7.2 The number of subjects with experience in each of the domains and applications in the user evaluation.

Subjects

End users, as discussed in Section 2.1, include people of all levels of experience who use computers as tools. For this evaluation we assembled ten volunteers between 20 and 30 years of age (seven male, and three female). All were university students or recent graduates: three were from computer-related disciplines, two from other sciences, four from the arts, and one from management. Two had used computers professionally, one an office worker and the other a computer support person. Four subjects were novices who had used spreadsheet and word processor applications to type documents but had no other training or experience.

Table 7.2 summarises the subjects' experience with the four domains represented in the two tasks. All had used the Microsoft Excel spreadsheet, though half claimed less than one year's experience. All had used a graphical user interface, six of these had used the Macintosh Finder. Fewer subjects had experience with an FTP client or image manipulation program. Each of the five subjects who had used an image manipulation program claimed to have used GIFConverter; this seems high, and suggests the subjects thought it a domain, not a specific program.

Participation

Table 7.3 shows the number of subjects that attempted the four variants of each task. The ten subjects were asked to attempt most variants (row A). Some subjects did not have time to complete the fourth variant of each task; they were allowed to read the problem description, then asked to explain how they would solve it (row B). The fourth variant tests the subjects' tool choice, which was included in their explanations, and their responses are valuable when considering the second hypothesis. Some of the subjects did not use multiple selection in variant 1; they were not asked to complete variants 2 and 4, which differ from variants 1 and 3 only by allowing or disallowing multiple selection. Three other subjects used multiple selection very rarely in variant 1 of the *image conversion* task and were not asked to perform variant 2, but were asked to attempt variant

		Calendar				Image			
		1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
A	Subjects asked to attempt task	10	9	9	4	10	6	10	7
B	Subjects asked to explain task	–	–	–	4	–	–	–	2
C	Subjects who used no multiple selection	–	1	–	1	–	4	–	1
D	Subjects who had technical problems	–	–	1	1	–	–	–	–
	Total	10	10	10	10	10	10	10	10

Table 7.3 Participation rates in the user evaluation.

4 because it is a longer task with greater incentive to use automation techniques (row C). Finally, one subject was unable to participate in the last two variants of the *calendar* task due to technical difficulties with Microsoft Excel (row D).

To save time, subjects were sometimes asked to halt before they completed all of a task. Each subject participated in a single two-hour session, and each was told that the time they took to complete the tasks was not important. However, the tasks are not trivial, and potential time overruns were anticipated. When the order of their operations became clear, subjects were asked to explain how they intended to complete the remainder of the task (confirming the observed order) and then to proceed to the next stage of the experiment. This methodology did not appear to affect the subjects' strategy, which was formed in the expectation of completing the entire task. Despite this precaution, one subject came under significant time pressure at the end of the experiment (see Section 7.1.4).

Equipment

The PBD system used in this evaluation was an early version of Familiar. Section 3.3.4 shows how the current version has evolved with respect to feedback gathered during the evaluation. The principal differences between the two versions are that the evaluation interface predicted one step at a time until the user requested a full iteration, did not explain predictions, and did not accept direct feedback about its predictions (instead the user gave feedback by performing a new demonstration). The two systems have equivalent learning power, but the improvements made to the newer system reduce user effort and increase user control. References to Familiar in this section pertain to the evaluation version.

The evaluation was carried out on a 200Mhz Power Macintosh 7300/200 computer with 32 megabytes of RAM.

7.1.2 Observations

The subjects used a variety of strategies to complete the tasks; the most common are reported here.

Nine of the ten subjects identified three subtasks in all variants of the *calendar* task: entering the topmost row of month names, then the leftmost column, then the body of the spreadsheet. In the first variant of the task, they completed the subtasks in left-to-right or top-to-bottom order, taking advantage of regularities in the data to use automation tools. In the second, when they had no automation tools, three attempted alternative strategies: two filled all the cells containing 1, then all the cells containing 2, and so on; the other copied and pasted a simple formula. The tenth subject, who did not use multiple selection at all, entered the entire spreadsheet one row at a time.

The *image conversion* task has several key steps that must be performed in a specific order. Figure 7.2 shows the strategies each subject used to perform variant 4 of the *image conversion* task. The requisite steps are listed left to right: the image files must first be downloaded (Figure 7.2, column 1), then opened (column 2), then saved (column 3), then uploaded (column 4), then deleted (column 5,6). The behaviour of each of the nine subjects, labelled A–I, is represented by a row of solid bars, showing the tools they used and how they grouped the steps. Subject C, for example, used multiple selection (MS) to download the PICT files (Figure 7.2, column 1); then taught Familiar to open each PICT, save it as a JPEG file, close its window in the image manipulation program, and upload the JPEG file to the FTP site (columns 2–5); then used multiple selection to delete all the files with one operation (columns 5,6). Subjects H and I were not asked to attempt variant 4, but to explain how they would perform it (Table 7.3, row 2). Some details were missing from their descriptions.

Figure 7.2 shows that every subject elected to use Familiar during the task. The *Save as* operation (column 3) cannot be automated with multiple selection, and all the subjects taught Familiar a subtask involving *Save as* and its adjacent steps. Two subjects (E and G) took this strategy to its extreme, and used Familiar to automate the entire task with a single cycle like the one in Figure 7.1. Subject D discovered a way to automate subtask 4, *Close window*, without using multiple selection or Familiar: he did not perform this operation until he had saved all the files, he then quit the application, causing it to close all the windows.

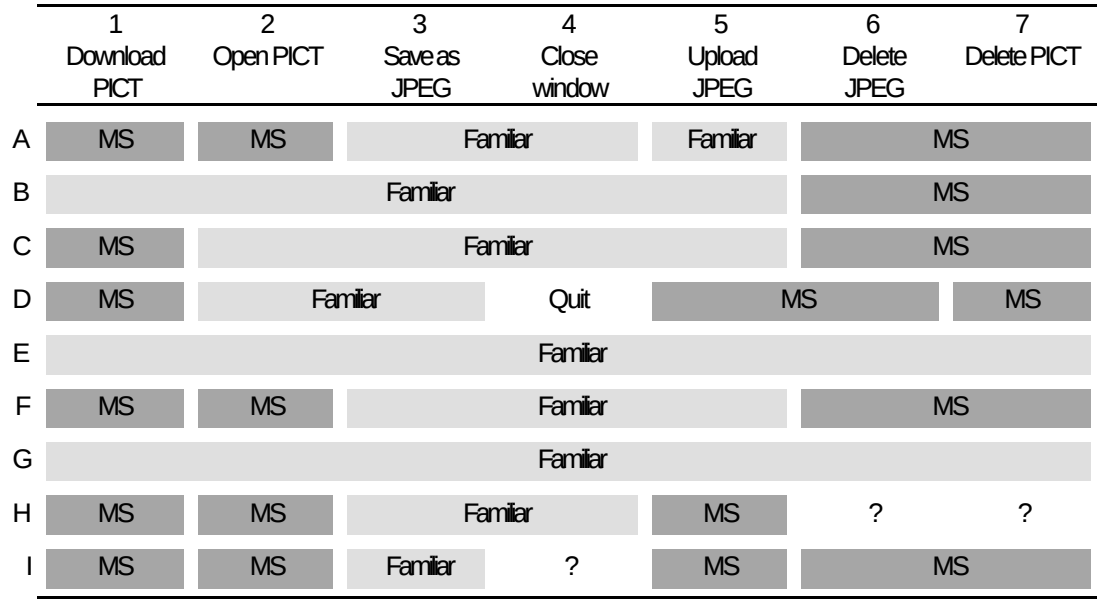


Figure 7.2 The tools used by the nine subjects (A–I) completing the seven steps (1–7) of variant 4 of the *image conversion* task.

7.1.3 Results

The first hypothesis is that end users are capable of using PBD to automate iteration. During the experiment, each subject was asked to complete large iterative tasks in situations where they must perform them manually, a tedious and time-consuming chore, or use Familiar. All of the subjects successfully used Familiar in these circumstances. Thus the evidence of the evaluation, reported in Section 7.1.4, supports the first hypothesis.

The second hypothesis is that end users will choose to use PBD tools over the available alternatives. This hypothesis was tested when the subjects (who had used both Familiar and aggregation-based tools) were asked to complete the two tasks using the tools they thought most appropriate. Most subjects chose not to use Familiar when alternatives were available, as is described in Section 7.1.5. The evidence of the evaluation does not support the second hypothesis.

The third goal of the user evaluation was to test the Familiar interface. The user's reaction to the interface, and their subsequent influence on its design, are described in Section 7.1.6.

The remainder of Section 7.1 discusses these results in detail. First the observations that support the first hypothesis are reported (Section 7.1.4), then those that contradict the second (Section 7.1.5). Finally, the feedback gathered from the subjects is discussed (Section 7.1.6).

7.1.4 Testing the end user's ability to use PBD

This section describes in detail the observations that test the first hypothesis, that end users are capable of using PBD to automate iteration.

In the third variant of each task, the subjects were asked to complete a large iterative task, but prevented from using multiple selection. They had to either use Familiar or complete the task manually, a tedious and time-consuming chore. Nine of the subjects attempted the third variant of the *calendar* task, and ten the third variant of the *image conversion* task; all elected to use Familiar. One had problems with an aspect of the task (described below), but all were able to automate iteration. The fourth variant of the *image conversion* task is also relevant because Familiar is the only tool that can automate the *Save as* command. Every subject who attempted the task successfully automated this step with Familiar, as shown in Figure 7.2, column 3.

In the final two variants of the *image conversion* task Subject A did not teach Familiar the full task correctly. Figure 7.2 shows the tools he used to perform each of the subtasks in variant 4. Having successfully saved the files in JPEG format with Familiar (columns 3–4), he restarted the agent and began teaching it to upload the JPEG files (column 5). However, he accepted an incorrect prediction (in both variants) and created a program to upload the same file sixty times, rather than every file once. The subject was in a hurry, and the pressure to finish quickly both affected his performance and prevented him from checking his results. His mistake was compounded by three factors: he was an inexperienced computer user, he poorly understood the AppleScript feedback, and he naively trusted Familiar. He was not told of the error after variant 3, so did not check for it in variant 4 (he remains unaware of it). Despite this error—serious though it was—his ability to complete the other subtasks and to automate the *calendar* task is evidence that, like the others, this subject is able to use Familiar to automate iteration.

7.1.5 Testing the end user's tool preference

This section reports in detail the observations that test the hypothesis that end users, given the choice, will choose to use PBD tools. In variant 4 of the two tasks, the subjects (who had used both Familiar and aggregation-based tools) were asked to complete iterative tasks that were too onerous to perform by hand, and allowed to choose the tools they thought most appropriate.

The analysis is complicated by two factors, both of which bias the subject towards solutions using Familiar. First, variant 4 of each task was attempted immediately after variant 3, in which the subjects used Familiar to complete an identical task, and some time after variant 1 of the corresponding task, in which multiple selection was used to complete a smaller and slightly different task. Some subjects repeated the strategy they used successfully in variant 3, rather than finding the best strategy. Second, the *image conversion* task contained a step that could not be automated with multiple selection; it had to be automated with Familiar or not at all. Consequently the two tasks are considered separately below.

Four subjects were asked to attempt the *calendar* task; four others were instead asked to explain how they planned to complete it. Table 7.4 shows that of these eight, all elected to use multiple selection, and three also used Familiar. Two used Familiar to generate a column of day names, then used multiple selection to copy the complete column, which they pasted into the remaining columns. The third used Familiar to repeatedly paste seven cells (containing the days of the week) but found that it was unsuited to this task (because some months end with an incomplete week) and abandoned this strategy in favour of one involving only multiple selection. Ultimately, two subjects used Familiar to automate a subtask, and the remainder chose not to use it; so the *calendar* task offers little evidence that end users will choose PBD over the alternatives.

Seven subjects were asked to attempt the final variant of the *image conversion* task, and two others were asked to explain how they planned to perform it. Table 7.4 shows that all nine elected to use Familiar at some point in the task, and that two used only Familiar and not multiple selection. Their tool choice is illustrated in Figure 7.2. The task is biased towards Familiar because the *Save as* command (Figure 7.2, column 3) cannot be automated with multiple selection. Consequently, every subject taught Familiar subtasks that included the *Save as* and adjacent steps. If these are excluded from Figure 7.2, Familiar was used only in only one of the remaining subtasks, and multiple selection in 19. This suggests the subjects prefer multiple selection to Familiar if either tool can be applied.

In conclusion, neither task supported the hypothesis that the end user will choose to use PBD when alternatives are available. Although Familiar was often applicable, and was occasionally the only tool that could automate a subtask, it was routinely overlooked in favour of other techniques. The subjects appeared to

	Calendar variant 4	Image variant 4
Total number of subjects	8	9
Number who used multiple selection	8	7
Number who used Familiar	3	9

Table 7.4 Tool preference in variant 4 of the user evaluation.

prefer multiple selection because it is fast, simple, familiar, and able to set termination conditions based on task data rather than a number of iterations.

7.1.6 User Feedback

This section summarises the subjects' feedback about Familiar, control issues, and macro recorders. It was gathered by observation and from interviews conducted immediately after the evaluation which included, but were not limited to, the questions in Appendix D.6.

Problems with Familiar

The subject's comments on Familiar, and their observed behaviour, led to several improvements in the interface.

The most common complaint was that Familiar is slow. Its speed is restricted by AppleScript, which is not a fast language. This problem is largely beyond Familiar's control, but is ameliorated by faster hardware and later versions of the operating system (Section 6.5.3).

The evaluation interface did not explain how predictions are made, and sometimes made predictions based on generalisations that did not match the subject's mental model, causing confusion when the inconsistency was revealed. Familiar's explanations address this problem.

Familiar occasionally made poor predictions in the face of multiple counter-examples, or took too long to make predictions. In these cases, the correct behaviour was obvious to the subject, and Familiar was able to predict it, but chose an incorrect pattern analysis scheme. Current versions of Familiar resolve this problem by allowing the user to explicitly reject Familiar's predictions.

Two experienced subjects identified problems with Familiar's handling of termination conditions. They said that multiple selection and the *Autofill* tool were superior because they knew exactly what data would be entered, and because the termination conditions are set explicitly and visually. This is a

revealing conclusion because the *AutoFill* interface does not show what data will be entered. *AutoFill* lets the user select two or more cells, then expand the selection to cover new cells, which are subsequently filled by extrapolating the values in the original selection. Three features are visible in the *AutoFill* interface but not in Familiar: the originally selected values from which the predictions are extrapolated, the position on the screen where the last value is entered, and the final extrapolated value. These features let the subjects reliably estimate what the filled values will be, and set termination conditions accurately. Section 8.4.1 describes how Familiar can be extended to set termination conditions based on extrapolated values rather than the number of iterations.

Some subjects commented that Familiar was unsuitable for the short iterations in the *calendar* task because it was too slow to execute predictions and learn changes in patterns (particularly when one column ended and the next began). This problem is addressed in part by faster hardware, and in part by a new implementation of the *PAS-extrapolation* scheme, but is not fully resolved.

The subjects were all asked whether they felt they knew what the AppleScript explanations meant. Eight claimed that they did. The other two were relatively inexperienced computer users, and one said he thought he would learn given time. The other was the subject described in Section 7.1.4 who accepted Familiar's erroneous predictions automating the *image conversion* task. Four of the subjects who said they understood had queried the terminology used by Microsoft Excel, and later acknowledged it had initially caused them difficulties.

Perceived loss of control

In previous user evaluations of PBD, subjects reported that they felt they had lost control of the user interface when the agent took over (Cypher, 1993b; Maulsby, 1994). Subjects in the Familiar evaluation did not report this problem.

The issue of control was considered in Familiar's design (Section 3.2.3), and again when the evaluation was proposed. The experimental design assumed that subjects would be more comfortable with PBD in familiar settings, when they felt that the system was helping them in a significant way, and when the new feature was entirely optional.

Familiar adds PBD to existing applications. Inexperienced end users are likely to be uncomfortable with new applications, and the experimental procedure was designed to avoid overwhelming users with the need to learn to use a new

application and a new interface technique at the same time. The subjects were asked to attempt the *calendar* task before the *image conversion* task because they were all expected to have at least some experience with Microsoft Excel spreadsheets. Some had no experience with some or all of the other applications (Table 7.2), but the first two variants of the task provided a gentle introduction. By the time Familiar was introduced, the subjects had used the applications to complete two variants of each task using the relevant application functionality.

The examples used in many PBD evaluations are obviously toy problems, which the subject is asked to solve with a particular system. The problem may be so contrived that that the user feels the evaluation (and hence the tool) is trivial, or so small that the user would perform it by hand in the normal course of events. In contrast, the Familiar evaluation used large and potentially tedious tasks, so that some kind of automation technique was desirable, or even necessary.

The Familiar evaluation placed the subject in a position where they could choose to use the tool, but were not required to use it. The subjects were told that they *may* use it if they want to, not that they *must* use it. In practice, the evaluation involved large and potentially tedious tasks, providing a strong incentive to use Familiar when other automation tools were unavailable. Thus the subjects used Familiar, but only when they actively chose to do so. Having taken the initiative, they were more likely to feel in control.

In summary, it is crucial that the user does not feel that the system has taken over, but that they have delegated a menial task—one that they do not want to perform themselves—to the system. This philosophy guided the Familiar evaluation, and appears to have minimised the subjects' perceived loss of control.

Macro recorders

Macro recorders are a form of PBD, and a potential solution to the tasks in the user evaluation. One subject inquired about using a macro recorder during the experiment. He wished to aggregate the *Resize*, *Save as*, and *Close window* steps of variant 2 of the *image conversion* task.

The post-experiment interviews revealed that five of the ten subjects had previously used a macro recorder. These five were asked if they had considered using macros in the *image conversion* task, and all but the one mentioned above said no. This subject had run macros on a daily basis in the workplace. Two of

the others had “forgotten” about macro recorders, the other two did not realise they were applicable to the task. None of the subjects thought a macro was appropriate for the *calendar* task because it was too “simple”. One subject explained that “I felt the time I would have taken to refresh myself with the macro would have been longer than just doing the table.” Another considered writing (programming) a macro, but not recording (demonstrating) one, and stated that his employer would probably discourage him from taking the time to learn (to program) macros in the workplace. It was unclear whether he understood the difference between recording a macro and writing a script.

The subjects’ lack of experience with macro recorders is instructive. Only one of the subjects seriously considered using a macro; others thought them inappropriate or too complex, and half did not know of them at all. These results reinforce the design choices in Chapter 3: if PBD is ever to attain a wider user base, it must be easy to discover and access, and easy to learn and use.

7.2 Task evaluation

The user evaluation demonstrates that end users are capable of using Familiar to automate iteration, but this ability will not help them unless they have the opportunity to use it. This section explores the situations where we can expect PBD to be useful. A set of iterative tasks is assembled, and circumstances under which they can be automated is elicited.

One of the motivations for a general-purpose system like Familiar is that it is impossible to anticipate exactly what tasks users will want to perform. Indeed, if we could anticipate all the user’s needs, we could build a specialised tool to satisfy each, and dispense with end-user programming completely. As a consequence, any evaluation that focusses on tasks is necessarily limited. Nevertheless, this evaluation strives to examine a broad range of tasks, and offers a unique insight into the challenges facing PBD by identifying the iteration problems that it can and cannot solve.

7.2.1 Procedure

The task evaluation takes the form of a Gedanken experiment that asks which of a set of example tasks can be automated with Familiar. The seventeen iterative tasks that we will consider are listed briefly in Table 7.5, with their source and

domain of application (rows 1–17). Three additional tasks are described; these were not used in the evaluation (rows 18–20). The tasks are all described in detail in Appendix A. They are numbered consistently in this Chapter and in Appendix A. The *mail merge* task, for example, is task 9, and appears on row 9 of Table 7.5 and in Appendix A.9.

The tasks were gathered prior to Familiar's implementation. The original goal was to assemble *repetitive* tasks, but most of the tasks were iterative, and this became Familiar's focus. The preponderance of iterative tasks does not necessarily mean that they form the bulk of repetition problems in everyday computer use. The tasks were gathered by informal methods and admit several alternative explanations: users do not think of tasks repeated at long intervals as repetitive; tasks not attempted recently are less likely to be recalled; and purely iterative tasks involve a single significant investment of time and are consequently more memorable. Ultimately, twenty tasks were assembled. Seventeen are examined in this analysis and the remaining three were discarded: the *fractal snowflake* task because the user is unlikely to attempt it through a graphical user interface (Appendix A.18.1), the *intelligent image filtering* task because it is not iterative (Appendix A.18.2), and the *manipulating spreadsheets* task because it takes place in a unique environment (Appendix A.18.3).

Table 7.6 summarises the source and domain of the seventeen iterative tasks. They are drawn from interviews with end users, related research including the test suite in *Watch What I Do* (Potter and Maulsby, 1993), tools in existing applications, and the author's experience (Table 7.6a). They take place in a number of domains, including drawing editors, email clients, FTP clients, image manipulation programs, operating systems (OS), spreadsheets, text editors (or word processors) and web browsers (Table 7.6b). Five span two or more domains.

In practice, relatively few can be automated by the implementation described in Chapters 3–5 using the existing Macintosh operating system and application software, either because no suitable application exists or because Familiar is incapable of inferring the tasks—or both.

Even a perfect PBD system, capable of learning any of the example tasks from a demonstration, could not automate all of the tasks on the Macintosh platform because of the lack of application support. Some domains simply have no recordable applications; others have recordable applications that do not provide

	Task	Source	Domain
1	Averaging column data *Calculate the average of ten blocks of numbers in adjacent cells.*	user (from literature)	spread-sheet
2	Calendar *Duplicate a printed calendar in a spreadsheet.*	author	spread-sheet
3	Copying files *Copy a set of files from one folder into another.*	literature	OS
4	Copying files to floppy *Copy a set of files onto a set of floppy disks.*	literature	OS
5	Copying mail headers *Create a numbered list of the subjects of email messages.*	literature	email, text
6	Image conversion *Download JPEG files, convert to PICT, return to FTP site.*	user	OS, FTP, graphics
7	Indexing document files *Create a catalogue of the files in a computer system.*	user	database, OS, others
8	Joining document sections *Compile a single text document from a set of web pages.*	author	text , web browser
9	Mail merge *Print a form letter for a list of people.*	application, literature	text
10	Network diagram *Copy a set of files onto a set of floppy disks.*	user	drawing, text editor
11	Numbering table of contents *Add section numbers to a table of contents.*	user	text
12	Printing odd and even pages *Print the odd pages of a document, then the even pages.*	literature	text
13	Program editing *Create a set of macros in a script editor.*	user	spread-sheet
14	Saving search results *Search internet for PostScript files and save the results.*	user	web browser
15	Sorting rectangles *Position a set of rectangles in order of increasing height.*	literature	drawing
16	Subtotal *Insert subtotals to a column of data values.*	application	spread-sheet
17	Truncate lines *Truncate a lines where they intersect another object.*	literature	drawing
18	Fractal snowflake *Draw a fractal snowflake (a triadic Koch curve).*	literature	drawing
19	Intelligent image filtering *Load selected images on a web page.*	author	web browser
20	Manipulating checklists *Promote every entry in a nested checklist.*	author	Apple Newton

Table 7.5 A summary of the tasks in Appendix A, showing the source, the domain, and a description of each (18–20 are not used in the task evaluation).

features necessary to the tasks. The first part of the task evaluation identifies the features necessary to automate the tasks through AppleScript that are missing from currently available applications.

Having identified the shortcomings of the applications, we consider which tasks could be automated if these shortcomings were rectified. Given a set of perfect

(a)

Source	Number of tasks
User	6
Literature	6
Application	2
Invention	2
Total	17

(b)

Domain	Number of tasks
drawing	2
operating system (OS)	2
spreadsheet	4
text	3
web browser	1
drawing, text editor	1
email, text	1
OS, database, others	1
OS, FTP, graphics	1
web browser, text	1
Total	17

Table 7.6 Source and domain of example tasks.

applications, the current Familiar implementation is capable of learning some but not all of the example tasks. The remainder can be divided into those that could be learned by an improved version of Familiar using the same inferencing model, and those that require a PBD system with greater abilities.

7.2.2 Results

In order for the user to be confident that a PBD system is useful, it is important that it be able to automate many tasks. The current version of Familiar, in conjunction with the current set of recordable applications, can be used to automate five of the seventeen iterative tasks gathered in Appendix A. Familiar's abilities can be extended by increasing the number of applications available or the sophistication of its inferencing.

Table 7.7 shows which tasks are currently practical. The recordable applications currently available for the Macintosh are sufficient to perform the ten tasks named in the top row of the table, but not the seven in the bottom row. The inferencing implemented in Familiar can learn the seven tasks in the leftmost column, but not the ten in the rightmost. Of the seventeen tasks, the five named in the top left cell of the table can be automated with the implementation of Familiar described in this thesis and the applications currently available.

The example tasks are drawn from nine domains, five of which boast effective recordable applications. These would allow a hypothetical perfect PBD system to automate ten of the seventeen tasks. If better applications were available, two more tasks could be automated with the current version of Familiar (Table 7.7, bottom left cell), and all might be automated by a perfect system.

	Familiar inferencing currently adequate	Familiar's inferencing not currently adequate
Current applications sufficient	1 Averaging column data 2 Calendar 4 Copy files to floppy 6 Image conversion 12 Printing odd and even pages	3 Copying files 9 Mail merge 11 Numbering table of contents 13 Program editing 16 Subtotal
Requires new or modified applications	5 Copying mail headers 8 Joining document sections	7 Indexing document files 14 Saving search results 15 Sorting rectangles 17 Truncate lines 10 Network diagram

Table 7.7 Requirements for automating the example tasks with Familiar.

In practice, the applications written for the Macintosh computer are beyond our control, but significant gains can be made by improving Familiar's inferencing. Even if a perfect recordable application existed in every imaginable domain, Familiar would be incapable of automating ten of the example tasks because they are beyond its inferencing abilities (Table 7.7, rightmost column).

Table 7.8 divides the set of iterative tasks into three categories. First are those that the current version of Familiar is capable of learning (left column). Second are those that can in principle be learned by the Familiar inferencing model (second column). Familiar will be able to automate them once the modifications described in Section 8.4.1 are complete. (These modifications comprise pattern analysis schemes that were not required in the user evaluation, and interface features that were not accorded a high priority.) Third are those tasks that Familiar cannot automate, but which might be solved by other PBD systems (third column). The high-level model described in Chapter 4 cannot learn nested iteration nor detect patterns containing conditionals. A sophisticated user can sometimes transform these problems into pure iteration problems, but this strategy may be beyond a typical end user.

The seven tasks that the current implementation cannot automate, but which lie within the capabilities of the Familiar model (Table 7.8, centre column), are useful for prioritising the functionality to add to Familiar. The evaluation in Section 7.2.4 suggests that the most worthwhile improvements are set iteration in sorted order, filtered set iteration, better mining of contextual data, and text inferencing. Improvements to Familiar based on these features are discussed in Section 8.4.1.

Only a few tasks are completely beyond Familiar's abilities. The tasks in question involve nested iteration, different sequences of steps in each iteration, and

Tasks that can be learned by the current Familiar	Tasks that can be learned by Familiar's inferencing model	Tasks that are beyond Familiar's inferencing model
1 Averaging column data	3 Copying files	7 Indexing document files
2 Calendar	9 Mail merge	14 Saving search results
4 Copying files to floppy	10 Network diagram	16 Subtotal
5 Copying mail headers	11 Numbering table of contents	
6 Image conversion	13 Program editing	
8 Joining document sections	15 Sorting rectangles	
12 Printing odd and even pages	17 Truncate lines	

Table 7.8 Tasks that Familiar is able to learn, that the inferencing model is theoretically able to learn, and that are beyond the inferencing model.

intuition. These abilities—except intuition!—have been demonstrated in other PBD systems (e.g. Halbert, 1993; Modugno and Myers, 1997), though they are usually explicitly controlled by the user in a programming environment, not inferred from their demonstration. Tasks that require human intuition have repetitive elements that could be learned by a suitable PBD system, and thus partially automated. However, the Familiar user interface is not suited to this style of interaction because it interprets additional user actions as noise. A modification that allows the user to force Familiar to retain a particular pattern is described in Section 8.4.1.

The remainder of Section 7.2 comprises the detail of the Gedanken experiment. Section 7.2.3 describes the shortcomings of existing Macintosh application programs, and Section 7.2.4 the shortcomings of Familiar.

7.2.3 Shortcomings of application programs

Seven of the example tasks cannot be automated because they occur in domains where there are no recordable applications, or where the recordable applications lack some feature (Table 7.7, bottom row). Two of these tasks could otherwise be solved with Familiar. This section examines each domain represented in the example tasks, and asks what functionality their applications lack. One caveat is that there may be recordable applications addressing the other domains that were not encountered during this research or that were released subsequently.

Ten of the tasks take place in domains where an adequate application exists (Table 7.7, top row). These include the FTP client, image manipulation program, operating system, spreadsheet, and text editor. Table 7.9 lists the set of domains, and examples of adequate and inadequate applications in each. This section

	Domain	Adequate applications	Inadequate applications
1	database		
2	drawing		
3	email client		MailSmith, Eudora
4	FTP client	Fetch	
5	image manipulation	GIFConverter	JPEGview
6	operating system	Finder	
7	spreadsheet	Microsoft Excel	
8	text editing	Scriptable text editor	Microsoft Word, Style
9	web browsing		Netscape Navigator

Table 7.9 Applications used in the example tasks.

examines the domains with no applications (drawing editor), those with inadequate functionality (email client, web browser), and those which are adequate but poor (spreadsheet, text editor).

The *network diagram*, *sorting rectangles* and *truncate lines* tasks take place in drawing applications. To automate them, Familiar requires a drawing application that is recordable and allows an agent to examine its data and extract the objects in a drawing. The third task also requires that each drawing object has an attribute that describes the objects that it intersects. No recordable drawing program currently exists.

The *joining document sections* and *saving search results* tasks both ask the user to iterate over a set of web pages using a web browser (Appendix A.8, A.14). Familiar cannot be used to automate them because current web browsers do not record navigation actions. Netscape Navigator (versions 3–4.5) records navigation actions that are initiated by the user typing a URL into a dialog box, as opposed to following a link.[1] This facility can occasionally be exploited to automate specific cases of the tasks, but requires that users change their behaviour. The discussion of the *joining document sections* task in Section 7.2.3 considers this solution in more detail.

The *copying mail headers* task requires a recordable email client (Appendix A.5). The task asks the user to copy the subject line of an email message. Several email clients are scriptable (e.g. Eudora Light, version 3; Netscape Messenger, version 4), and MailSmith (version 1) is partially recordable, but none satisfy the requirements of the tasks.

[1] It is possible to monitor all the pages the user has visited, but not through the standard AppleScript recording protocol.

Microsoft Excel, perhaps the most widely used Macintosh spreadsheet application, has an adequate implementation of AppleScript. All the functionality required in the tasks is available, but Excel's display of formulae and formatting are very poor. It is therefore debatable whether Microsoft Excel is adequate for the *subtotal* task (Appendix A.16), as users are likely to have difficulty understanding the AppleScript it generates (Section 6.5.4).

Many of the tasks involve repetitive text editing operations. The Scriptable Text Editor is an adequate application for most such tasks. However, its AppleScript implementation causes some confusion in the example tasks because it refers to documents by changeable index numbers (Section 6.5.3). Its reliability is also questionable, because recordings of long tasks do not always recreate the user's actions exactly. Despite these concerns, the Scriptable Text Editor is superior to both Microsoft Excel (which changes its behaviour when recorded and records in programming language syntax) and other recordable editors which omit text manipulation commands (e.g. Style version 1.5, Tex Edit Plus version 2.2). Two professional text editing tools, Word Perfect (version 3.1) and MacWrite Pro (version 1.5), are reported to have flawed implementations of AppleScript recording (Fenner *et al.*, 1995–1996), but were not examined for this evaluation.

The *indexing documents* task leaves the full range of applications unspecified (Appendix A.7). Of those cited by the user, only one (Microsoft Excel) has a useful recording function. A recordable database program (InfoDepot version 2.0) has been reported (Fenner *et al.*, 1995–1996), but has not been examined for this evaluation. Recordable applications are available in a range of other domains, including chemical modelling, compression, statistical modelling, terminal emulation, and scheduling (Fenner *et al.*, 1995–1996; Apple Computer Inc, 2000). These were not examined, as they are not relevant to the evaluation.

In summary, adequate AppleScript recordable applications exist for five of the nine domains represented by the example tasks. Ten of the seventeen tasks can be automated with existing applications, but the remaining seven can not. This evaluation suggests that AppleScript currently lacks the recordable applications that it needs to be an effective platform for end-user PBD. However, there are numerous Macintosh applications, and this examination may have overlooked some that are suitable for automating tasks in the outstanding domains. Further, new applications are continuously being created, and old applications updated, so the situation may improve over time.

7.2.4 Shortcomings of Familiar

This section considers each task in turn, and divides them into the three classes of Table 7.8: those that Familiar is currently capable of learning (left column); those that the Familiar inferencing model is capable of learning (centre column); and those that are beyond Familiar's inferencing ability (right column).

Some of the tasks in Appendix A are so small that it is not clear that automating them will benefit the user in any practical sense. They could be performed as quickly by hand. This evaluation is not concerned with this distinction, and explores only the system's ability to learn the example problems, not the ultimate utility to the user. However, it is important to realise that any of the tasks could be made to seem more important by supplying more data. For example, the practical benefits of automating the *copying mail headers* task, which involves iterating over seven messages, initially appears trivial (Appendix A.5). However, if we consider the same task in a mail folder containing seventy messages, or seven hundred, or seven thousand, then the benefits of automation quickly become obvious. Similarly, the other iterative tasks can be made to appear more important by the expedient of increasing the requisite number of iterations.

Automation with the Familiar implementation

Seven tasks can be learned by the version of Familiar described in Chapters 3–5 (Table 7.8, leftmost column). The *calendar* and *image conversion* tasks were performed by end users in the evaluation of Section 7.1, and require no further discussion. The *averaging column data* task, *copy files to floppy*, and *printing odd and even pages* tasks are also straightforward. The *copying mail headers* and *joining document sections* tasks are learnable, but require suitable applications.

The *averaging column data* task asks the user to calculate the average of ten blocks of numbers (Section 1.3.2, Appendix A.1). It is easily learned by Familiar, and was used to generate training data for Familiar's machine learning schemes (Appendix B.3). One difficulty is that Microsoft Excel's AppleScript feedback is difficult to understand (Section 6.5.4). Although the syntax is problematic for users, Familiar extrapolates it easily. Indeed, the problem with the syntax is that it is more suited to programs than to people.

The *copying files to floppy* task asks the user to copy a large set of files onto a set of floppy disks (Appendix A.4). In principle, this task involves nested iteration (over disks, and over files) and complex termination conditions that are

conceptually beyond Familiar's inferencing, but in practice the task is automated fairly simply because Familiar stops if it encounters a system error. Consequently, Familiar does not need to calculate how many files to copy to each floppy, it simply copies files until a *disk full* error occurs, then waits while the user inserts a new floppy.

Figure 7.3 shows a user teaching Familiar to copy files onto standard floppy disks. They start by demonstrating the first two iterations (Figure 7.3a). The Familiar prediction window appears, correctly predicting the next operation, and the user tells Familiar to iterate 1000 times (Figure 7.3b). Figure 7.3c shows the prediction window during the task, when 28 of these iterations have been performed. Eventually, the floppy disk becomes full, and an error occurs when Familiar attempts to copy another file onto it (Figure 7.3d). The user presses the *Stop* button in the error dialog, ejects the floppy disk, and inserts a new one.[2] At this point, there are 77 commands in the event trace, culminating in the most recent failed *copy* command (Figure 7.3e). The most recent file was not copied, so the user selects it in the Finder interface and drags it onto the floppy, and Familiar records their actions (Figure 7.3f). Familiar then correctly predicts that it should copy the remaining files onto the new floppy disk (Figure 7.3g).[3] The user can then instruct Familiar to copy the next set of files.

The *printing odd and even pages* task asks the user to print first the odd pages of a document, and then the even pages in reverse order (Appendix A.12). It is easily learned by Familiar, though it is complicated by poor implementations of AppleScript in existing applications. The task is begun by printing pages one, three, and five of the document. Figure 7.4 shows the AppleScript recorded when this is attempted in three different applications. Microsoft Excel records poorly formed but effective instructions (Figure 7.4a), the Scriptable Text Editor records simple statements but omits necessary information about the pages printed (Figure 7.4b), and Microsoft Word records nested Visual Basic commands (Figure 7.4c). Familiar can be used to automate the task in Excel and Word (though AppleScript from the latter is difficult to display), and in other applications.

---

[2] Conveniently, the Finder does not record the *Eject disk* command, though if it did the *SRS-noisy* sequence recognition scheme would identify it as a noise event and ignore it.

[3] In this case the new floppy is named *untitled*, the same as the first floppy. If the new name were different, Familiar could learn it from the new example in Figure 7.3f.

Figure 7.3 The *copying files to floppy* task.

Two other tasks can be learned by Familiar, but are not practical because no suitable recordable applications exist in the task domain. The first of these, the *copying mail headers* task, could be automated if an existing email client were extended to record all the user's actions as they perform the task. The second, *joining document sections*, can be automated in the special case where the data is in a regular format and the user is willing to change their behaviour. The full task can be automated with suitable adaptations to the web browser software.

(a)
```
tell application "Microsoft Excel"
    activate
    Print SelectedSheets of ActiveWindow FromPage 1 ToPage 1 Copies 1
    Print SelectedSheets of ActiveWindow FromPage 3 ToPage 3 Copies 1
    Print SelectedSheets of ActiveWindow FromPage 5 ToPage 5 Copies 1
end tell
```

(b)
```
tell application "Scriptable Text Editor"
    activate
    print document 1
    print document 1
    print document 1
end tell
```

(c)
```
tell application "Microsoft Word"
    activate
    do Visual Basic "Application.PrintOut FileName:=\"\",
        Range:=wdPrintRangeOfPages, Item:= _
        wdPrintDocumentContent, Copies:=1, Pages:=\"1-1\", PageType:= _
        wdPrintAllPages, Collate:=True, Background:=False"
    do Visual Basic "Application.PrintOut FileName:=\"\",
        Range:=wdPrintRangeOfPages, Item:= _
        wdPrintDocumentContent, Copies:=1, Pages:=\"3-3\", PageType:= _
        wdPrintAllPages, Collate:=True, Background:=False"
    do Visual Basic "Application.PrintOut FileName:=\"\",
        Range:=wdPrintRangeOfPages, Item:= _
        wdPrintDocumentContent, Copies:=1, Pages:=\"5-5\", PageType:= _
        wdPrintAllPages, Collate:=True, Background:=False"
end tell
```

Figure 7.4 Event traces recorded while printing pages 1, 3, and 5 of a document in (a) Microsoft Excel, (b) the Scriptable Text Editor, and (c) Microsoft Word.

The *copying mail headers* task asks the user to copy the subject line of each email message in a folder, and to paste it into a text document (Appendix A.5). Familiar can learn this task, but it requires a recordable email client. Figure 7.5 shows a hypothetical event trace that might be used to teach Familiar the task.[4] They record a user selecting and opening a message (events 2,3), selecting and copying its subject line (events 4,5), and closing the message window (event 6). The user then selects a text editor document (event 8), types the first list number (event 8), pastes the subject line (event 9), and types a return character (event 10). Each of these steps is easily learned by Familiar.

The *joining document sections* task asks the user to visit every page on a web site, copying the text of each into a text editor (Appendix A.8). The shortcomings of

```
 1  activate – Hypothetical MailSmith extension
 2  select message id 161 of incoming mail
 3  open selection
 4  select subject of message window 1
 5  copy
 6  close message window 1
 7  activate – Scriptable Text Editor
 8  set selection to "1. "
 9  paste
10  set selection to "  "
```

Figure 7.5 Hypothetical event trace for the *copying mail headers* task based on the dictionary in the MailSmith email client.

Macintosh web browsers affect the way the task is automated (Section 7.2.3). A hypothetical web browser that records the user following hyperlinks could be used to automate this task. Figure 7.6a shows one iteration from an event trace in such a web browser. Familiar has only to follow the link labelled *Next* in each iteration to generalise event 2. Netscape Navigator (versions 3–4.5) does not record navigation actions when the user follows a link, but does record those that are initiated by the user typing a URL into a dialog box. This facility can be used to join sectioned documents whose pages have regular URLs. Event 2 of Figure 7.6b was recorded when a user retrieved the first page of such a document. Familiar can automate the task if the sections of the document are consistently named (e.g. *page1.html*, *page2.html*, *page3.html*, etc). This is not usually the case.

Automation with the Familiar model

Seven of the example tasks can be learned with the Familiar inferencing model, but are beyond the abilities of the current implementation (Table 7.8, centre column). They could be learned if the implementation were extended as described in Section 8.4.1.

The *copying files* task can be automated with relatively simple extensions to Familiar. The description in Appendix A.3 has two variants. The first asks the user to copy every file whose name contains the substring *canyon*. In the second, the user is asked to copy a set of files based on two file attributes (type and extension). Familiar requires two modifications before it can automate these tasks. First, it must be able to iterate over some but not all of a set of containee objects. Second, it must be able to examine substrings of the properties of objects.

[4] The hypothetical email client commands (Figure 7.5, lines 1–6) are modelled on the existing MailSmith application, which is partially recordable.

(a)
1 Activate – Hypothetical web browser
2 Follow link "Next"
3 Select all
4 Copy
5 Activate – Scriptable Text Editor
6 Paste
7 Type enter

(b)
1 Activate – Netscape Navigator
2 Open URL "http://www.site.com/books/bookname/page1.html"
3 Select all
4 Copy
5 Activate – scriptable text editor
6 Paste
7 Type enter

Figure 7.6 Automating the *joining document sections* task (a) with a hypothetical web browser and (b) with regular URLs in Netscape Navigator.

In practical terms, this means examining contextual information in greater detail than the property level—a complicated search task, but a common one that Familiar should address.

The *mail merge* task asks the user to copy each of a list of names and addresses into a form letter one at a time, printing a copy of each letter (Appendix A.9). Figure 7.7 shows Familiar after the user has shown it how to automate a simplified version of the task where every address is four lines long. In each iteration, Familiar brings the *address list* document to the front (Figure 7.7, event 29), selects and copies the next address (events 30, 31), brings the *form letter* document to the front (event 32), selects the old address and copies the new one over it (events 33, 34), and finally prints the altered document (event 35). The AppleScript is unnecessarily complicated because the Scriptable Text Editor identifies the rearmost document as *document 2*, but changes that description to *document 1* when it is brought to the front (events 29, 32). This example works because every address is exactly four lines long and is separated from the previous one by a single blank line, so Familiar does not need to generalise about the text; it simply increments the beginning and end of the text to be copied from the *address book* document by five paragraphs (i.e. five lines) in each iteration (event 30). The addresses in the full formulation of this task vary from three to five lines, and Familiar would need to learn to generalise the syntactic features of text to automate it properly.

The *network diagram* task asks the user to draw a network diagram based on a written description of the network (Appendix A.10). Although the user did not

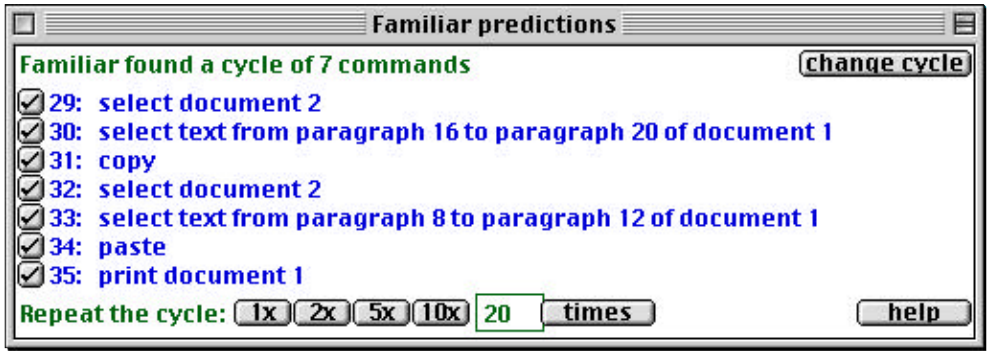


Figure 7.7 Completing a simplified *mail merge* task with Familiar.

describe the parts of the task they found repetitive, we can easily identify several iterative subtasks. The first is to draw the three servers and 24 client computers. The user must copy a drawing of a computer to appropriate positions on the screen, a subtask that Familiar can learn because each is offset by a regular distance. A second and more difficult subtask is to add a label naming the computer to each of the pictures. This can be achieved by opening the text file containing the task description in a text editor, selecting and copying the first *computer ID*, pasting the text string into the drawing editor, and (if necessary) moving it to the appropriate position. These actions are theoretically within the abilities of Familiar's inferencing, but the current version would be unable to generalise the features of the *computer ID* in the text file, and it is unclear whether it could predict coordinates in the drawing program. The other attributes of each computer (*administrator*, *domain name*, *computer type*) can be selected from the text file and pasted into the diagram in the same way as the *computer ID*.

The *numbering table of contents* task asks the user to reformat several pages of text (Appendix A.11). This task cannot be automated with Familiar because it requires the ability to generalise text strings, just as the full *mail merge* task does.

The *program editing* task asks the user to duplicate a macro several times and perform several small edits on each copy (Appendix A.13). Its domain is difficult to define: it is a text-editing task but originally took place in a spreadsheet macro editor. Like the two previous tasks, Familiar is unable to automate it because it cannot infer text patterns. However, some of the subtasks can be learned if the user decomposes the task appropriately.

The task can be divided up into four subtasks: copying the macro an appropriate number of times, changing each macro name, changing each macro button operation, and changing each copy argument. Figure 7.8a shows the history

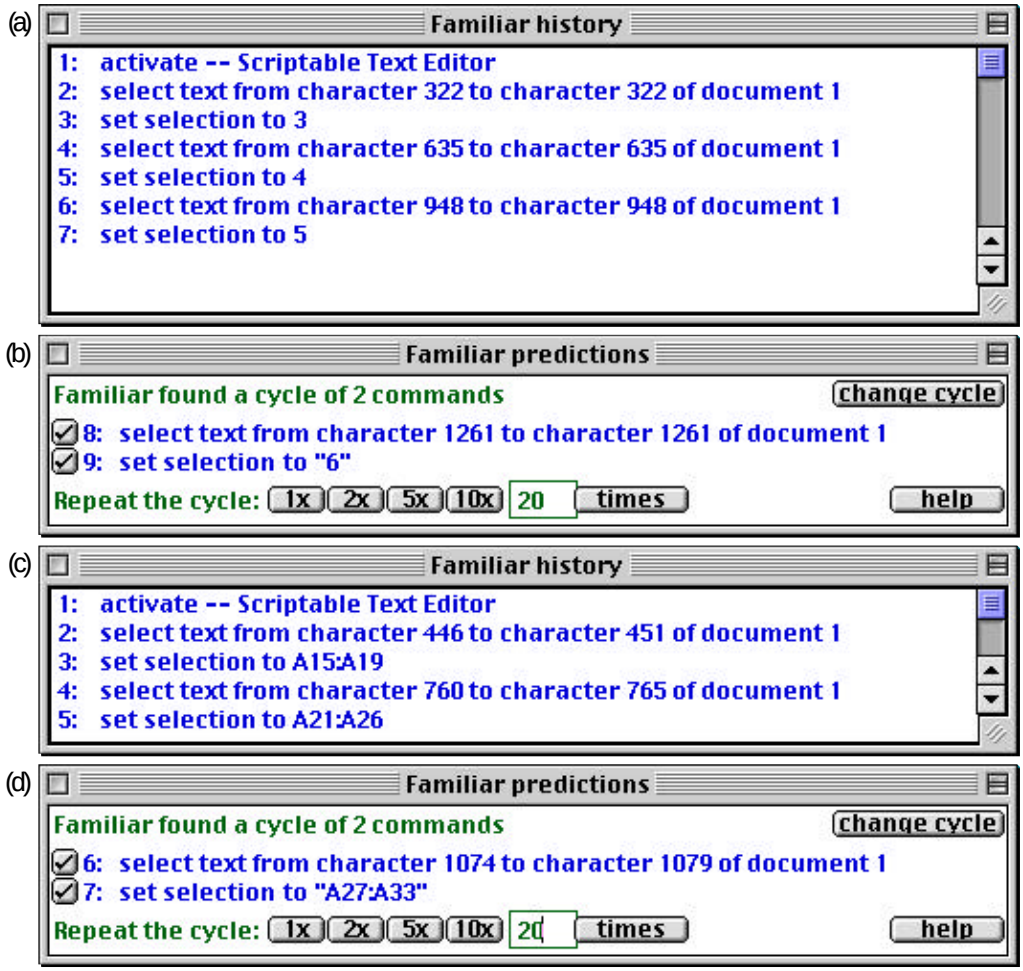


Figure 7.8 Using Familiar to (a,b) successfully change macro names, and (c,d) unsuccessfully attempt to infer irregular spreadsheet regions.

window as the second subtask is demonstrated. The user selects the single character in the macro name that needs to be changed (lines 2,4,6), then types in a correction (lines 3,5,7). Figure 7.8b shows the prediction window after Familiar has learned the task. This strategy is possible because each copy of the macro is the same length (in lines and characters), but has to be relearned after the ninth iteration because two characters (10, 11, 12…) will then be inserted into the text of each macro, causing the subsequent text offsets to increase.

Figure 7.8c,d shows a more difficult problem. The user demonstrates the fourth subtask by selecting the text range (events 2,4) and replacing it (events 3,5). However, Familiar's prediction of the next iteration, shown in Figure 7.8d, is incorrect: the cell range should be "A27:A40", not "A27:A33", because the ranges in the spreadsheet vary in size unpredictably from one iteration to the next. Familiar must either learn to automate only part of the task, leaving the user to enter the cell ranges, or infer the range by examining the spreadsheet. This is

perhaps too difficult a task for a general PBD system, though specialised systems have been taught to find similar relationships (Myers *et al.*, 1994; Gaxiola, 1995; Sugiura and Koseki, 1996).

The *sorting rectangles* task asks the user to arrange a set of rectangles in sorted order (Appendix A.15). To automate this task, Familiar must examine the drawing data, extract the objects, and iterate over them in sorted order. It is currently able to perform the first two of these steps, but not the third.

The *truncate lines* task asks the user to iterate over a set of parallel lines, shortening each so that it intersects with some other object (Appendix A.17). Even if suitable applications were available, Familiar would need two modifications to learn this task. First, it must be able to iterate over the parallel lines in order of location. Second, it would then need to learn to set the new endpoint of each line to its intersection with the target object. This task might be possible with the current *PAS-previous* pattern analysis scheme if the drawing package was suitably implemented; if not, a more general scheme for searching the contextual information would be required. Interestingly, Potter (1993a) shows how the *truncate lines* task can be performed using only low-level information.

Automation using a more general inferencing model

Three of the example tasks cannot be learned by the Familiar inferencing model (Table 7.8, right column). The *indexing document files* task requires different steps in each iteration, and the *saving search results* and *subtotal* tasks require nested iteration. These might be automated, or partially automated, by restructuring the task so that Familiar can learn it. Similar "circumvention and delegation" strategies have been observed in other studies (Bhavnani and John, 1998).

The *indexing document files* task asks the user to create a database of all the files stored on a computer system (Sections 1.3.3, Appendix A.7). It presents four obstacles that a PBD system must overcome. First, the task has little structure, and may require a different pattern of commands in each iteration. The Familiar inferencing system must perform the same tasks in each iteration, so cannot automate this task without extending the learning model. Second, the task occasionally calls for intuitive guesses on the user's part. PBD systems are ill-suited to intuition, though in an ideal case they might detect some underlying rule based on contextual information that the user is unable to articulate. Third, the files are stored in many locations on a computer, so a recursive traverse of the containees of an object (the hard disk) will be necessary, and only some of the

```
 1  Select the first cell in column A
 2  Repeat
 3      let X = the row number of the selected cell
 4      move selection down until a cell with a new value is selected
 5      let Y = the row number of the selected cell - 1
 6      insert a row
 7      select cell in column A of the current row
 8      set contents of selected cell to "Subtotal:"
 9      set style of selected cell to Bold
10      select cell in column B of the current row
11      set formula of selected cell to "=SUM(BX:BY)"
12      select cell in column A of the current row
13      move selection down to the next row
```

Figure 7.9 Pseudocode for automating the *subtotal* task.

containees need be examined. This is beyond the current implementation, but might be implemented within the bounds of the inferencing model. Fourth, Section 1.3.3 suggests that this task might be partially automated by Familiar. The interface, which allows the user to intersperse commands from the prediction window with new demonstrations, makes this possible, but it is unwieldy in practice because noise events that interfere with the pattern analysis are common. Enhancements that make this style of interaction more feasible are described in Section 8.4.1.

The *saving search results* task asks the user to iterate over a set of web pages, downloading all the PostScript files linked to by each (Appendix A.14). Familiar cannot learn this task because it involves nested iteration, first over pages of results, then over the links on each page. Additionally, the inner iteration (over links) is selective: only links to PostScript files are relevant.

The *subtotal* task asks the user to insert subtotals into a column of numbers in imitation of Microsoft Excel's subtotal tool (Appendix A.16). Familiar is unable to automate it. Figure 7.9 shows an algorithm for automating the task. The most complex part of the task occurs on line 4, where a downward search is performed for the next row that requires a subtotal. This is effectively a second iterative task nested inside the first, and beyond Familiar's abilities. It might instead be viewed as a search operation, which Familiar is also unable to perform, though AIDE shows that complex searches can be automated with explicit demonstrations (Piernot and Yvon, 1993). The formula in line 11 appears complex, as it depends on two variables (X, Y) that are set previously (lines 3, 5). However, both variables will appear in the event trace as properties of selected cells and can be inferred from the contextual information.

8 Conclusions

This thesis extends our knowledge of PBD into the world of the end user. It focusses on the tasks real users perform, the tools they use, and the skills they are able to apply. It shows that PBD can be used in real-world situations to solve real-world problems. The central claim (Section 1.2) is that

> *domain independent PBD can be made available in existing applications, and used to automate iterative tasks by end users.*

This thesis considers this claim in two parts. First, it shows that it is possible to make PBD available in existing applications, and second, it shows that end users are capable of using it to automate iteration. Both of these arguments are revisited in Section 8.3.

This chapter summarises the main findings of the thesis. Section 8.1 gives an overview of the argument and reviews the role of each chapter. Section 8.2 describes the contributions this research has made to the study of demonstrational interfaces, and Section 8.3 returns to and expands on the thesis claims. Finally, Section 8.4 outlines future work.

8.1 Overview

We began by introducing PBD, iterative tasks, and end users, the three most important themes of this work (Chapter 1). Three end users are described as they contend with iterative tasks. Although PBD has the potential to solve these problems, it cannot be applied in the real world.

Chapter 2 surveys research into PBD and automating iteration problems. There are several existing techniques and strategies for automating iteration, including manual execution, command and element aggregation, and end-user programming. Existing PBD systems that address iteration problems are specific to particular applications and consequently solve only a small fraction of the

tasks that a user might encounter. Other PBD systems solve repetitive (not iterative) tasks, or specialised problems, and are usually inapplicable to iteration problems. Chapter 2 concludes by describing the problems with PBD that prevent its uptake in the real world.

The third chapter continues where the second left off, discussing the design of PBD systems for real-world problems based on the needs and abilities of end users. It advocates an application-independent approach using existing applications and environments with a focus on simplicity, minimising the user's role in program generation, and attempting to educate the end user. These guidelines are used to design Familiar, a PBD system for automating iteration problems. Familiar aims to support and educate the end user, but is limited by AppleScript, the platform on which it is implemented.

Familiar has several unique features that differentiate it from other systems (Chapter 4). It is domain independent, and able to work with completely new applications that it has never before encountered. At the same time, it operates at a high level, exploiting detailed knowledge of each application to improve its performance. Familiar's inferencing algorithm is applicable to any high-level event system. It works in two phases, first detecting iterative patterns in command sequences, then extrapolating command parameters. Its modular design is extensible, tolerates noise, incorporates feedback, generates explanations, and incorporates standard machine learning techniques.

Familiar uses machine learning techniques in two ways: to guide prediction and to build conditional rules from the user's demonstration (Chapter 5). First, the inferencing system executes several prediction engines in parallel, then uses two sets of learned rules to choose the best prediction to present to the user. These rules are generated by a machine learning algorithm from training data gathered when a user performed a set of iterative tasks. An evaluation shows that this method is better than three obvious heuristics, but can be improved by incrementally updating the classifier at run time to reflect unique aspects of the current user's interaction style. The second application of machine learning is to learn conditional relationships between command parameters and attributes found in the event trace. Several problems with simple rules are solved by using a permutation test statistic to identify pertinent attributes. Permutation tests are useful solutions to PBD problems because they work with small quantities of data.

Chapter 6 addresses the first claim of this thesis: that PBD can be made available in existing applications. The platform requirements of such a PBD system are users and applications, recordability, controllability, examinability, user interface, and consistency. A range of software architectures using different levels of abstraction meet these requirements, but the level of access they allow an agent limits the capabilities of a PBD system. This point is reinforced by a detailed examination of the AppleScript architecture, which has proved to be adequate for PBD, but which has a number of significant deficiencies. Most of these can be attributed to the fact that the language was designed to be attractive to end users, not for agent communication. The chapter concludes with a set of guidelines for implementing PBD-aware recordable applications.

The second claim of the thesis, that end users are capable of using PBD to solve iterative tasks, is examined in Chapter 7. A user evaluation shows that end users are able to use PBD, and that in some circumstances they will choose to do so. However, they generally preferred simpler techniques based on multiple selection when they were available. A task evaluation ascertains the scope of the iterative tasks that can be solved with PBD. It examines a set of example tasks, and finds that some can be automated by the current implementation of Familiar in existing applications, and that most of the remainder could be solved given appropriate applications and a set of practical extensions to the Familiar implementation.

8.2 Summary of contributions

This thesis makes contributions in three areas. The first two are based around making PBD available in existing applications and showing that end users can use it to automate iterative tasks. The remainder arise from the design and implementation of Familiar.

8.2.1 Making PBD available in existing applications

Chapter 6 shows that PBD can be made available in existing applications with three analyses. First, the general requirements of PBD systems are identified, and a range of platforms that support them are described. Second, the specialised requirements of existing PBD systems and the possibility of integrating them into existing applications are discussed. Third, the AppleScript scripting language is

examined, and its strengths and weaknesses are explored. The major contributions are

- a set of basic platform requirements for PBD systems, comprising users, applications, recordability, examinability, controllability, interface, and consistency;
- a description of high-level, low-level, and mid-level architectures that satisfy these requirements;
- a survey of the information requirements of existing inferencing systems, and how they can be satisfied by existing applications;
- a survey of user interface and feedback techniques used in existing systems, many of which require cooperation from the application, and how they can be integrated with the interfaces of existing applications;
- an analysis of AppleScript, a high-level scripting language that is an adequate platform for PBD, and the problems with its high-level event architecture, language design, and application implementations; and
- a set of guidelines for designing AppleScript implementations that support PBD systems and other agents.

8.2.2 End users can automate iteration with PBD

The user evaluation in Chapter 7 shows that end users are capable of using PBD to automate iterative tasks. After a short tutorial exercise, all subjects in the evaluation were able to use Familiar to automate iterative tasks. As expected, Familiar was not the subjects' first choice as an automation tool. With some exceptions, they tended to use it when alternatives based on multiple selection were unavailable or could not be applied to the task. An obvious criticism of the experiment is that it was a small study: ten subjects participated, and their performance was observed as they performed two tasks. Although a larger study would impart more statistical validity, the insights offered by the present evaluation are worthwhile and unlikely to be contradicted. The main contributions are

- evidence that end users are capable of using PBD to automate iterative tasks in existing applications;

- evidence that end users will not necessarily *choose* to use PBD when other techniques are more appropriate, and that users are comfortable using different techniques to automate different steps of a larger task; and
- a task evaluation that examines a wide-ranging set of iterative tasks and classifies each on the basis of the conditions necessary to solve it using the Familiar PBD system and Macintosh applications.

8.2.3 Familiar

The Familiar PBD system was designed, implemented, and evaluated to support the argument of the thesis, but makes several contributions to the state of the art in its own right. These contributions are in three distinct areas: Familiar's design, its architecture, and its use of machine learning.

Familiar's design is user focussed (Chapter 3), and contributes

- an analysis of the end user's motivation, skills, and attitudes;
- a set of example tasks against which a PBD system's performance can be measured;
- design guidelines including the use of existing applications, simplicity, minimising user effort, and educating the end user; and
- a user interface based on these guidelines and an explanation of the practical compromises they entail.

Familiar's inferencing system (Chapter 4) contributes

- a domain-independent system for modelling applications that gathers class information about objects and commands, and uses it to gather contextual information by examining the applications during a demonstration;
- a modular algorithm for detecting and extrapolating repetition in a high-level event architecture using a two-level model that first detects repetition and then extrapolates it;
- modules for detecting patterns in perfect and noisy demonstrations; and
- modules for extrapolating patterns by extending simple sequences, by searching the event trace, and by retrieving and examining application data.

Familiar uses machine learning to guide prediction and to construct conditional rules (Chapter 5), contributing

- a method for using standard machine learning techniques to guide prediction;
- an extension to this technique that improves predictive performance by using an incremental learner to adapt to individual users;
- a system for inferring conditional rules by training a machine learning scheme on data gathered from the event trace and associated contextual information;
- an explanation of the obstacles to inferring conditional rules, including the abundance of attributes and the scarcity of instances; and
- an explanation of how permutation tests overcome these obstacles by calculating the statistical significance of inferred rules.

8.3 Claims revisited

This thesis claims that domain-independent PBD can be made available in existing applications and used to automate iteration by end users. Both parts of the claim are supported by this thesis, with some caveats.

The general requirements of domain-independent PBD systems that exploit existing applications are users and applications, recordability, controllability, examinability, interface, and consistency. Any architecture that meets these requirements is a viable platform for PBD, but the potential of PBD systems are limited by the level of access an architecture permits, and hence the information that it makes available. Generally, high-level architectures meet the requirements well and allow complex inferencing, but few real systems allow high-level access. Conversely, low-level architectures are common and easily accessed, but do not provide application information, restricting PBD to the simple mimicry seen in macro recorders. Mid-level access can be added to many architectures through standard user interface toolkits. Mid-level events are accessible and allow access to the application's interface components, though they do not provide direct examinablity. At each level of abstraction, this thesis presents a range of literature that shows that current computer architectures can meet the requirements of PBD, and that they are sufficient to implement PBD systems.

The user evaluation of Familiar showed that end users are able to use PBD to automate iterative tasks. The subjects used the interface to automate iteration in

existing applications, and exploited its domain independence by automating tasks that spanned as many as three different commercial applications. Although every subject was able to use the Familiar interface, most preferred alternative techniques when they were available and could be applied to the problem. Sometimes they misunderstood Familiar's intentions and capabilities. The evaluation version of Familiar was not polished—it was slow, it did not accept feedback, it offered no explanations, and its English-language sentences were often poorly formed. The current version resolves these problems, and we can suppose that it would fare even better in a similar evaluation.

8.4 Future work

This section identifies four avenues for future research, ranging from the practical improvements to Familiar that arose from the task evaluation in Chapter 7, to machine learning and user studies, to the application of the results for improving software architectures.

8.4.1 Augmenting Familiar

Familiar is a complete and working system, but its modular design has yet to be fully exploited. This section describes several areas where Familiar might be extended. The suggestions are arranged in order of likely practical benefit. The first area includes the inferencing requirements identified in the task evaluation of Section 7.2. These extensions to Familiar are specific and constitute practical enhancements that will lead to measurable improvements in the task evaluation. Second are enhancements to the user interface that make automating partial tasks more practical. This recommendation also stems from the task evaluation. The remaining avenues are more speculative, they involve termination conditions, new inferencing algorithms, and generating AppleScript programs.

Inferencing requirements from task evaluation

The task analysis in Section 7.2 identifies seven tasks that cannot be learned by the current implementation of Familiar because they require additional pattern analysis schemes. These were omitted from the current version because they were not needed in the user evaluation. The new features are iteration over some but not all containee objects, iteration over containee objects in sorted order, rules

based on contextual information occurring as substrings of object properties, and text inferencing.

The *PAS-set* pattern analysis scheme handles iteration over the containees of an object, such as the *files* in a *folder* (Section 4.4.3). It iterates over every object in alphabetical order of name, and can neither filter the objects nor sort them based on other attributes. The ability to filter a set would allow the user to iterate over some, but not all, of a set of objects. For example, the user might want to iterate over the *files* in a *folder* that are GIF images. Filtered set iteration could be implemented using the learning techniques from *PAS-ML* (Section 4.4.3). *PAS-set* currently sorts objects in ascending alphabetical order of the *name* attribute. In order to sort on another attribute—*size* or *creation date*, for example—Familiar would need to ask the application for the attribute value of every object, then use it as the sort key. Neither filtering nor sorting are difficult to implement, though retrieving the requisite contextual information may prove time-consuming.

The *PAS-ML* and *PAS-previous* pattern analysis schemes both search the event trace and its associated contextual information for relationships between the values of a target parameter and the values of previously occurring parameters and properties. Both search for relationships between complete parameters and properties, and ignore partial matches. This prevents Familiar from learning descriptions like *every file whose name contains "canyon"* (Section 7.2.3). In practice, the ability to learn partial matches will be useful for many tasks (Section 7.2; Modugno and Myers, 1997). A related problem is Familiar's inability to generalise and instantiate free text. It cannot learn regular expressions to describe positions and selections in text like those demonstrated in CIMA (Maulsby, 1994), which restricts Familiar to automating only the simplest text processing tasks.

Interface enhancements from task evaluation

Many iterative tasks are irregular, require intuition, or are too difficult for Familiar to learn completely. These may never be fully automated with PBD, but might be partially automated. Unfortunately, the current interface is unsuited to partial automation because Familiar interprets noise events that occur after a pattern has been learned as signals that the pattern is wrong and should be abandoned. There is no way to tell Familiar that the correct pattern has been learned but to expect noise events and temporary deviations.

Partial tasks could be supported with some changes to the Familiar *prediction* window. In addition to the *change cycle* button, which lets the user explicitly reject a pattern, a *confirm cycle* button could explicitly confirm it. This would prevent Familiar from abandoning what it has learned when the user performs noise actions. Some additional controls may be necessary to prevent Familiar and the user performing the same actions. This would give the user greater control of the commands executed in each iteration, at the expense of increased interface complexity.

Termination conditions

The examples in Chapter 3 show that Familiar's handling of termination conditions is unsophisticated. In Figure 3.5b, to choose an arbitrary example, the user must type in the number of iterations to perform even though Familiar can deduce this information from the fact that it is iterating over nine files in the folder and has performed three iterations. Alternatively, Figure 3.5c shows that there are too many files to fit within the visible window area, so the user may want to stop iterating when the edge of the window is reached. Familiar can be extended to calculate and suggest likely termination conditions at the same time (and in the same way) that it predicts parameters and gives explanations. Each pattern analysis scheme could, in addition to predicting the next iteration, be asked to predict the number of iterations the user will perform, and to provide a reason for the prediction. These estimates could be added to the *predictions* window with the standard buttons for executing a fixed number of cycles.

New sequence recognition and pattern analysis schemes

Familiar is designed to be extended, and can have an arbitrary number of sequence recognition and pattern analysis schemes. Several practical pattern analysis techniques are described above. Another potential scheme is the "noisy integer" extrapolation (linear regression) used in Eager (Cypher, 1993b). Only two sequence recognition algorithms have been implemented because they have detected all the patterns in Familiar's evaluation thus far. However, both have weaknesses, as described in Section 4.3.2. Many other algorithms might be used, including those from the PBD literature (Crow and Smith, 1992; Lau and Weld, 1999; Ruvini and Doni, 1999), and general-purpose sequence detection schemes like Sequitur (Nevill-Manning, 1996). Probabilistic (compression) models predict the likelihood of the next event and are suitable only if they provide pattern and history information; some, like PPM, retain this as context (Cleary *et al.*, 1995).

Generating code

Familiar makes predictions through a purely interactive interface, and discards the tasks it has learned when they are finished. Users cannot ask Familiar to remember a task—if they want to perform the task again they have to teach it to Familiar again. This problem could be addressed by having Familiar generate an AppleScript program to complete the entire iterative task. Users could execute this program to repeat the task, or modify it by conventional programming techniques. The AppleScript language is sufficiently powerful for this purpose—the complexity of a PBD system lies in inferring the patterns, extrapolating them is comparatively simple. Appendix E describes an algorithm for generating AppleScript code from Familiar's predictions.

8.4.2 Machine learning

The applications of machine learning described in Chapter 5 are both novel and effective, but barely scratch the surface of a very fertile research area. Familiar uses machine learning in two ways. Each makes significant contributions to the inferencing system, but neither has been subjected to rigorous investigation and both warrant further investigation.

Section 5.2 explains how machine learning is used to guide prediction, and notes several issues that deserve attention. First, the training tasks were limited by the time and resources available. It is interesting to consider how the mixture of training tasks could affect the results of an evaluation. A second factor is the way these tasks are performed. Bias may be introduced in the training data if the user who trained the system has an idiosyncratic demonstration style, if their demonstration contains an unusually high or low number of errors, or if they are overly familiar or unfamiliar with the training tasks. A third issue is the composition of the attributes that Familiar uses to estimate the probability that a prediction is correct. The attributes used may not be the best ones; they were chosen for their convenience and their similarity to a set of heuristic measures that, ultimately, were arbitrarily chosen.

Familiar's second use of machine learning is to infer conditional rules from the user's demonstration. Section 5.3 explains the problems encountered in the implementation, and the measures taken to solve them. Two areas invite further investigation. First, Familiar's rules are based on a single attribute, and the number of tasks it could learn might be increased if they were more complex.

Second, the permutation test statistic is theoretically sound, and appears to perform well in the experiments described, but has not been properly evaluated. It has yet to be formally compared to other machine learning algorithms and tested on rules that are based on more than one attribute.

8.4.3 User studies

Familiar has only been formally evaluated by end users in one experiment, and that evaluation used an early version of the Familiar interface. Additional evaluations will serve two purposes. First, the interface can be further refined based on the users' feedback. The explanations and the ability to change cycles and reject predictions have yet to be formally evaluated. Second, a weakness of the first evaluation is that it is small: it considered only ten users and two tasks. Further evaluation will reinforce the results described in Section 7.1, and test whether the improvements to the user interface and the increased responsiveness offered by faster hardware make the interface more useable.

8.4.4 Improved computer architectures

The description in Chapter 6 of the requirements that PBD systems make of computer architectures can be directly applied to software architectures and application programs. Commercial architectures have reached the point where they are just adequate to support PBD systems, and will progress still further in the future if the demand for "intelligent agents" is sustained. Platform designers must incorporate the requirements of agents, including those identified in this thesis, into software architectures if the state of the art is to advance and PBD to appear on every desktop.

References

Apple Computer Inc (1990–1997) *Develop Magazine.* Apple Computer Inc, Cupertino, CA.

Apple Computer Inc (1992) *Macintosh Human Interface Guidelines.* Addison-Wesley, Reading, MA.

Apple Computer Inc (1992-1999) *Apple Event Registry: Standard Suites.* Apple Computer Inc, Cupertino, CA.

Apple Computer Inc (1993–1999) *AppleScript Language Guide: Engl;ish Dialect*. Apple Computer Inc, Cupertino, CA.
http://developer.apple.com/techpubs/macos8/InterproCom/AppleScript Scripters/AppleScriptLangGuide

Apple Computer Inc (1996) *Inside Macintosh: Interapplication communication*. Apple Computer Inc, Cupertino, CA.
http://developer.apple.com/techpubs/mac/IAC/IAC-2.html

Apple Computer Inc (2000) *Scriptable applications*. Web page, Apple Computer Inc, Cupertino, CA.
http://www.apple.com/applescript/enabled.00.html

Arnold K. and Gosling J. (1996) *The Java Programming Language*. SunSoft Press, Mountain View, CA; Addison-Wesley, Reading, MA.

Bergadano F., Matwin S., Michalski R., and Zhang J. (1988) “Measuring Quality of Concept Descriptions.” *Proc. of the European Working Session on Learning.* Glasgow, UK: Pitman, London, UK.

Bharat K. and Sukaviriya P. (1993) “Animating User Interfaces Using Animation Servers.” *Proc. ACM Symposium on User Interface Software and Technology*, pp. 69–80. Atlanta, GA: ACM Press, New York, NY.

Bharat K., Hudson S. and Sukaviriya P. (1995) “Synthesized Interaction in the X Window System.” *Technical Report 95-07*, Graphics, Visualization and Useability Centre, College of Computing, Georgia Institute of Technology. http://www.cc.gatech.edu/gvu/reports/1995/abstracts/95-07.html

Bhavnani, S. K. and John, B. E. (1998). “Delegation and Circumvention: Two Faces of Efficiency.” *Proc. Conference on Human Factors in Computing Systems (CHI)*, pp. 273–280. Los Angeles, CA: ACM Press, New York, NY.

Box, D. (1997) “Say Goodbye to Macro Envy with Active Scripting.” *Microsoft Interactive Developer*, February. Microsoft Corporation, Redmond, WA. (Note: this periodical has been renamed *MSDN Magazine*.) http://www.microsoft.com/mind

Cheikes B. A., Geier M., Hyland R., Linton F., Rodi L., Schaefer H. (1998) “Embedded training for complex information systems.” *Proc. Intelligent Tutoring Systems*, pp. 36-45. San Antonio, TX: Springer-Verlag, Berlin.

Cleary J G., Teahan W. J., and Witten I. H. (1995) “Unbounded Length Contexts for PPM.” *Proc. Data Compression Conference*, pp. 52–61. Snowbird, Utah: IEEE Computing Society, Los Alamitos, CA.

Cockburn A. and Bryant A. (1997) “Leogo: An equal opportunity user interface for programming.” *Journal of Visual Languages and Computing, 8*: 601–619. Academic Press, New York, NY.

Crow D. N. and Smith B. M. (1992) “DB_Habits: Comparing minimal knowledge and knowledge-based approaches to pattern recognition in the domain of user-computer interaction. In *Pattern recognition and nueral networks in human-computer interaction*, edited by R. Beale and J. Finlay, pp. 39-63. Ellis Horwood, Chichester, UK.

Cuff R. (1980) “On casual users.” *International Journal of Man-Machine Studies, 12 (2)*: 163–202. Academic Press, New York, NY.

Cypher A. (1990) “Managing the Mundane.” In *The Art of Human-Computer Interface Design*, edited by B. Laurel, pp. 155–160. Apple Computer Inc, Cupertino, CA.

Cypher A. (1991) Eager: Programming Repetitive Tasks by Example. *Proc. Conference on Human Factors in Computing Systems (CHI)*, pp. 33–39. New Orleans, LA: ACM Press, New York, NY.

Cypher A. (Ed) (1993a) *Watch what I do: Programming by Demonstration*. MIT Press, Cambridge, MA.

Cypher A. (1993b) "Eager: Programming repetitive tasks by demonstration." In Cypher 1993a, pp. 205–217.

Digitool (1995-1999) *Macintosh Common Lisp.* Macintosh Software. http://www.digitool.com/

Edwards W. K., Hudson S. E., Marinacci J., Rodenstein R., Rodriguez T., and Smith I. (1997) "Systematic Output Modification in a 2D User Interface Toolkit." *Proc. ACM Symposium on User Interface Software and Technology*, pp. 151–158. Banff, Canada: ACM Press, New York, NY.

Erickson T. (1997) "Designing Agents as if People Mattered." In *Software Agents*, edited by J. M. Bradshaw, pp. 79–96. AAAI Press, Menlo Park, CA; MIT Press, Cambridge, MA.

Fenner M., Terry F. and PreFab Software Inc. (1995–1996) *Scriptable Apps.* Web page. http://www.tiac.net/prefab/scriptweb/scriptableapps.html

Fisher R. A. (1936) "The use of multiple measurements in taxonomic problems." *Annual Eugenics 7* (II): 179–188. (Reprinted in *Contributions to Mathematical Statistics*, 1950. John Wiley, New York, NY.)

Fizner W. F. and Gould L. (1993) "Rehearsal world: Programming by rehearsal." In Cypher 1993a, pp. 79–100.

Foner L. N. (1993) "What's an agent anyway? A sociological case study." *Agents Memo 93-01*, Agents Group, MIT Media Lab, Cambridge, MA.

Frank E. (2000) *Pruning decision trees and lists.* PhD Thesis, Department of Computer Science, University of Waikato, Hamilton, New Zealand.

Frank E. and Witten I. H. (1998) "Using a Permutation Test for Attribute Selection in Decision Trees." In *Proc International Conference on Machine Learning*, pp. 152–160. Madison, WI: Morgan Kaufmann, San Francisco, CA.

Frank E., Paynter G. W., Witten I. H., Gutwin C., and Nevill-Manning C. G. (1999) "Domain-specific keyphrase extraction." *Proc. International Joint Conference on Artificial Intelligence*, pp. 668–673. Stockholm, Sweden: Morgan Kaufmann, San Francisco, CA.

Gaxiola D. (1995) “Tatlin: Integrating commercial applications into programming by demonstration.” *Advanced Undergraduate Project*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Geier M. (1999) *WOSIT User Manual: Version 1.1*. The Mitre Corporation, Bedford, MA.
http://www.mitre.org/centers/cafc3/wosit/

Geier M. (forthcoming) *JOSIT User Manual*. The Mitre Corporation, Bedford, MA.

Good P. (1994) *Permutation tests: A practical guide to resampling methods for testing hypotheses*. Springer-Verlag, New York, NY.

Gutwin, C., Paynter, G., Witten, I.H., Nevill-Manning, C., and Frank, E. (2000) “Improving browsing in digital libraries with keyphrase indexes.” *Journal of Decision Support Systems* 27 (1–2): 81–104; November.

Halbert D. (1993) “SmallStar: Programming by Demonstration in the Desktop Metaphor.” In Cypher 1993a, pp 103–124.

Hendry D. G. and Green T. R. G. (1994) “Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model.” *Int. Journal of Human-Computer Studies* 40: 1033–1065.

Hix D. and Hartson H. R. (1994) *Designing user interfaces: Ensuring usability through product and process*. J. Wiley, New York.

Holte R. C. (1993) Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. *Machine Learning 11 (1)*: 63–91.

Jones S. and Paynter G. W. (1999) "Topic-based browsing within a digital library using keyphrases." *Proc. ACM Conference on Digital Libraries*, pp. 114–121. Berkeley, CA: ACM Press, New York, NY.

Kay J. and Thomas R. C. (1995) “Studying Long-Term System Use.” *Communications of the ACM* 38 (7): 61–69; July.

Kosbie D. and Myers B. (1993a) “PBD Invocation Techniques: A review and proposal.” In Cypher 1993a, pp 423–431.

Kosbie D. and Myers B. (1993b) “A system-wide macro facility based on aggregate events: A proposal.” In Cypher 1993a, pp. 433–444.

Kurlander D. (1993) “Chimera: Example-based graphical editing”. In Cypher 1993a, pp. 271–292.

Lau T. A. and Weld D. S. (1999) Programming by demonstration: An Inductive Learning Formulation. *Proc. International Conference on Intelligent User Interfaces*, pp. 61–69. Redondo Beach, CA: ACM Press, New York, NY.

Lieberman, H. (1993a) “Tinker: A Programming by Demonstration System for Beginning Programmers.” In Cypher 1993a, pp. 49–68.

Lieberman, H. (1993b) “Mondrian: A teachable graphical editor.” In Cypher 1993a, pp. 340–358.

Lieberman, H. (1998) “Integrating user interface agents with conventional applications.” *Proc. International Conference on Intelligent User Interfaces*, pp. 39–46. San Francisco, CA: ACM Press, New York, NY.

Lieberman H., Nardi B. A. and Wright D. (1999) “Training agents to recognize text by example.” *Proc. International Conference on Autonomous Agents*, pp. 116–122. Seattle, WA: ACM Press, New York, NY.

Linton F., Charron A. and Joy D. (1998) “OWL: A Recommender System for Organisation-Wide Learning.” *Technical Report*, The MITRE Corporation, Bedford, MA.

Linton F., Joy D, and Schaefer H. (1999) “Building User and Expert Models by Long-Term Observation of Application Usage.” *Proc. International Conference on User Modeling*. Banff, Canada: Springer-Wien, New York, NY.

Mason D. C. and Wheeler D. A. (1999) *GNOME User’s Guide*. http://www.gnome.org/

Masui T. and Nakayama K. (1994) “Repeat and Predict: Two Keys to Efficient Text Editing.” *Proc. Conference on Human Factors in Computing Systems (CHI)*, pp. 118–123. Boston, MA: ACM Press, New York, NY.

Maulsby D. (1994) *Instructible agents*. PhD Thesis, Department of Computer Science, University of Calgary, Calgary, Canada.

Maulsby D., Kittlitz K. A., and Witten I. H. (1989a) “Constraint-solving in interactive graphics: A user-friendly approach.” Proc. Computer Graphics International, pp. 305–318. Leeds, UK: Springer-Verlag, Tokyo.

Maulsby D., Witten I. H., and Kittlitz K. A. (1989b) “Metamouse: Specifying graphical constraints by example.” *Computer Graphics, 23* (3): 127–136.

McCloud, S. (1994) *Understanding Comics*. Kitchen Sink Press.

McDaniel R. G. and Myers B. A. (1998) “Building applications using only demonstration.” *Proc. International Conference on Intelligent User Interfaces,* pp. 109–118. San Francisco, CA: ACM Press, New York, NY.

McDaniel R. G. and Myers B. A. (1999) “Getting more out of programming by demonstration.” *Proc. Conference on Human Factors in Computing Systems (CHI),* pp. 442–449. Pittsburgh, PA: ACM Press, New York, NY.

Michail A. (1998) “Imitation: An alternative to generalization in programming by demonstration systems.” *Technical Report UW-CSE-98-08-06,* Department of Computer Science, University of Washington.

Microsoft Corporation (1992–1998) *Microsoft Word 98 Macintosh Edition.* Macintosh Software. Microsoft Corporation, Redmond, WA.

Microsoft Corporation (1996) “Windows Scripting Host: A Universal Scripting Host for Scripting Languages.” *White Paper*, Microsoft Corporation, Redmond, WA.

Mitchell T., Caruana R., Freitag D., McDermott J., and Zabowski D. (1994) “Experience with a Learning Personal Assistant.” *Communications of the ACM* 37 (7): 81–91; July.

Miura M. and Tanaka J. (1998a) “A Framework for event-driven demonstration based on the Java toolkit.” *Proc. Asia Pacific Computer Human Interaction Conference,* pp. 331–336. Kanagawa, Japan: IEEE Computing Society, Los Alamitos, CA.

Miura M. and Tanaka J. (1998b) “Jedemo: The Environment of Event-driven Demonstration for Java Toolkit.” *Proc. International Symposium on Future Software Technology (ISFST),* pp. 215–218. Hangzhou, China.

MJT Net Ltd. (1998) *Macro Scheduler.* Microsoft Windows Software. http://www.mjtnet.com/

Modugno F., Corbett A. T., and Myers B. A. (1996) “Evaluating program representation in a visual shell.” *Proc. Experimental Studies of Programmers Sixth Workshop,* pp. 131–146. Alexandria, VA: Ablex Publishing corporation, Norwood, NJ.

Modugno F. and Myers B. A. (1997) “Visual programming in a visual shell.” *Journal of Visual Languages and Computing, 8*: 491–522.

Myers B. A. (1992) “Demonstrational interfaces: A step beyond direct manipulation.” *Computer* 25 (8): 61–73; August.

Myers B. A. (1993a) “Peridot: Creating user interfaces by demonstration.” In Cypher 1993a, pp. 125–153.

Myers B. A. (1993b) “Garnet: Uses of demonstrational techniques.” In Cypher 1993a, pp. 219–236.

Myers B. A. (1993c) “Tourmaline: Text formatting by demonstration.” In Cypher 1993a, pp. 309–322.

Myers B. A. (1998) “Scripting Graphical Applications by Demonstration.” *Proc. Conference on Human Factors in Computing Systems (CHI),* pp. 534–541. Los Angeles, CA: ACM Press, New York, NY.

Myers B. A., Goldstein J., and Goldberg M. A. (1994) “Creating Charts by demonstration.” *Proc. Conference on Human Factors in Computing Systems (CHI)* pp. 106–111. Boston, MA: ACM Press, New York, NY.

Myers B. A. and Kosbie D. S. (1996) “Reusable hierarchical command objects.” *Proc. Conference on Human Factors in Computing Systems (CHI),* pp. 260–267. Vancouver, Canada: ACM Press, New York, NY.

Myers B. A. and Maulsby D. (1993) Glossary. In Cypher 1993a, pp 593–604.

Nardi B. A. (1993) *A Small Matter of Programming*. MIT Press, Cambridge, MA.

Nardi B. and Miller J. (1990) “The spreadsheet interface: A basis for end user programming.” *Proc. 3rd IFIP. Conf. Human-Computer Interaction (INTERACT),* pp. 27–31. Cambridge, UK: Elsevier Science Publishers (North-Holland), Amsterdam, The Netherlands.

Nevill-Manning C. G. (1996) *Inference of Sequential Structure.* PhD Thesis, The Department of Computer Science, The University of Waikato, Hamilton, New Zealand.

Nevill-Manning C. G., Witten I. H. Witten, and Paynter G. W. (1997) “Browsing in Digital Libraries: A Phrase-Based Approach.” *Proc. ACM Conference on Digital Libraries*, pp. 230–236. Philadelphia, PA: ACM Press, New York, NY.

Nevill-Manning C. G., Witten I. H. Witten, and Paynter G. W. (1999) “Lexically-generated subject hierarchies for browsing large collections.” *International Journal of Digital Libraries* 2 (2 / 3): 111–123, September.

Nielsen J. (1993) *Usability engineering.* Academic Press, New York, NY.

Norman D. A. (1994) “How might people interact with agents.” *Communications of the ACM* 37 (7): 68–71; July.

Olsen D. R. Jr., Boyarski D., Verratti T., Phelps M., Moffett J. L. and Lo E. L. (1998) Generalising Pointing: Enabling Multiagent Interaction. *Proc. Conference on Human Factors in Computing Systems (CHI),* pp. 526–533. Los Angeles, CA: ACM Press, New York, NY.

Olsen D. R. Jr., Hudson S. E., Verratti T., Heiner J. M. and Phelps, M. (1999) “Implementing interface attachments based on surface representations.” *Proc. Conference on Human Factors in Computing Systems (CHI),* pp. 191–198. Pittsburgh, PA: ACM Press, New York, NY.

Paynter G. W. and Witten I. H. (1999) “Automating Iteration with Programming by Demonstration: Learning the User’s Task.” *Proc. IJCAI Workshop on Learning About Users,* pp. 51–56. Stockholm, Sweden: Morgan Kaufmann, San Francisco, CA.

Paynter G. W., Witten I. H., Cunningham S. J. and Buchanan G. (2000) “Scalable browsing for large collections: A case study.” *Proc. ACM Conference on Digital Libraries.* San Antonio, TX: ACM Press, New York, NY.

Piernot P. P., and Yvon M. P. (1993) “The AIDE project: An application-independent demonstrational environment.” In Cypher 1993a, pp. 383–401.

Piernot P. P., and Yvon M. P. (1995) “A Model for Incremental Construction of Command Trees.” *Proc. Conference on Human-Computer Interaction (HCI),* pp. 169-179. Huddersfield, UK: Cambridge University Press, Cambridge, UK.

Potter R. (1993a) “Triggers: Guiding Automation with Pixels to Achieve Data Access.” In Cypher 1993a, pp. 361–380.

Potter R. (1993b) “Just-in-time programming.” In Cypher 1993a, pp. 513–526.

Potter R. and Maulsby D. (1993) “A Test Suite for Programming by Demonstration.” In Cypher 1993a, pp. 539–592.

Preece J. (1994) *Human-computer interaction.* Addison-Wesley, Reading, MA.

Quinlan J. R. (1986) “Induction of decision trees.” *Machine learning* 1 (1): 81–106.

Quinlan J. R. (1993) *C4.5: Programs for machine learning.* Morgan Kaufman, San Francisco, CA.

Rader C., Brand C. and Lewis C. (1997) “Degrees of comprehension: Children’s understanding of a visual programming environment.” *Proc. Conference on Human Factors in Computing Systems (CHI),* pp. 351–358. Atlanta, GA: ACM Press, New York, NY.

Ritter S. and Koedinger K. R. (1995) “Towards lightweight tutoring agents.” *Proc. AI-ED 95 World Conference on Artificial Intelligence in Education,* pp. 91-98. Washington DC.

Ruvini J. and Dony C. (1999) “Learning user habits: The APE project.” *Proc. IJCAI Workshop on Learning About Users,* pp. 65–73. Stockholm, Sweden: Morgan Kaufmann, San Francisco, CA.

Schiefler R. W. and Gettys J. (1990) *X Window System*. Digital Press, Boston, MA.

Schlimmer J. C. and Hermens L. A. (1993) “Software Agents: Completing Patterns and Constructing User Interfaces.” *Journal of Artificial Intelligence Research* 1: 61-89, November.

Sengers P. (1999) “Designing comprehensible agents.” *Proc. IJCAI ’99,* pp. 1227–1232. Stockholm, Sweden: Morgan Kaufmann, San Francisco, CA.

Shneiderman B. (1997) “Direct manipulation verses agents: Paths to predictable, controllable, and comprehensible interfaces.” In *Software Agents*. Edited by J. M. Bradshaw, pp. 97–106. AAAI Press, Menlo Park, CA; MIT Press, Cambridge, MA.

Simone C. (1995) “Designing a scripting implementation.” *Develop* 21: 48–72; March. Apple Computer Inc, Cupertino, CA.

Smith D. C. (1975) *Pygmalion: A computer program to model and simulate creative thought.*

Smith D. C. (1993) “Pygmalion: An executable electronic blackboard.” In Cypher 1993a, pp 19–47.

Smith, D. C., Cypher A, and Spohrer J. (1994) “KidSim: Programming agents without a programming language.” *Communications of the ACM* 37 (7): 54–67; July.

Sugiura, A. and Koseki Y. (1996) “Simplifying Macro Definition in Programming by Demonstration.” *Proc. ACM Symposium on User Interface Software and Technology,* pp. 173–182, Seattle, Washington: ACM Press, New York, NY.

Thimbleby H., Cockburn A., and Jones S. (1992) “HyperCard: An object-oriented disappointment.” In *Building Interactive Systems: Architectures and Tools.* Edited by P.D. Gray and R. Took, pp. 35–55. Springer, Berlin.

UserLand Software (1992–1999) *Frontier.* Version 5. Macintosh Software. http://frontier.userland.com/

Waern A. (1997) *What is an Intelligent Interface?* Seminar Notes, Swedish Institute of Computer Science, Kista, Sweden. http://www.sics.se/~annika/

Widemann B. (1992–1999) *AliasMenu*. Version 2. Macintosh Software.

Witten I. H. (1993) “A predictive calculator.” In Cypher 1993a, pp. 67–76.

Witten I. H. and Frank E. (2000) *Data Mining.* Morgan Kaufmann, San Francisco, CA.

Witten I. H. and Maulsby D. (1993) “Metamouse: An instructible agent for programming by demonstration.” In Cypher 1993a, pp. 155–181.

Witten I. H. and Mo D. (1993) “TELS: Learning text editing tasks from examples.” In Cypher 1993a, pp. 183–203..

Witten I. H., Paynter G. W., Frank E., Gutwin C., and Nevill-Manning C. G. (1999) “Kea: Practical automatic keyphrase extraction.” *Proc. ACM Conference on Digital Libraries,* pp. 254–256. Berkeley, CA: ACM Press, New York, NY.

Wolber D. (1996) “Pavlov: Programming by stimulus-response demonstration.” *Proc. Conference on Human Factors in Computing Systems (CHI),* pp. 252–259. Vancouver Canada: ACM Press, New York, NY.

Yvon M. P., Piernot P. P. and Cot N. (1995) “Programming by Demonstration: Detect Repetitive Tasks in Telecom Services.” *Proc. Australian Conference on Computer-Human Interaction (OzCHI),* pp. 68–74. Wollongong, Australia.

Zeiliger R. and Kosbie D. (1997) “Automating Tasks for Groups of Users.” *Proc. IFIP International Conference on Human-Computer Interaction (INTERACT),* Sydney, Australia: Kluwer Academic Publishers, Boston, MA.

A Iterative tasks

This Appendix contains a set of iterative tasks that are used in the evaluation in Chapter 7. The tasks were gathered in 1996 and 1997, before Familiar was implemented Section 7.2 explains how and why they were assembled.

The PBD literature contains several references to iterative tasks that involve formatting semi-structured text. Maulsby (1994) in particular examines a number of “Find and do” problems, involving phone numbers, bibliography entries, names, and addresses. These were omitted because they are restricted to the text editing domain, which is already well represented in the task list.

A.1 Averaging column data

Domain: Spreadsheet

Source: User (reported in Hendry and Green, 1994, p. 1041)

User task: Calculate the average of ten blocks of numbers in adjacent cells.

Background: Two lists of 500 numbers into the first two columns of a spreadsheet (Figure 1.2, columns A and B). These represent data in ten blocks of 50 records each. The user is asked to calculate the average of column B for each of the blocks, and store the results in the first ten cells of column C. The correct formulae are shown in Figure 1.2, column C.

This task is superficially simple but difficult to solve with a conventional spreadsheet. The user cannot simply enter the first formula in the topmost cell of column C copy and paste it because the formulae are offset by one cell and the blocks by 50 cells. If the first formula is pasted into the second cell, it will be copied as *=AVG(B2:B51)*, not *=AVG(B51:B100)*.

This task is also described in Section 1.3.2.

A.2 Calendar

Domain: Spreadsheet

Source: Author

User task: Duplicate a printed calendar in a spreadsheet.

Background: The original calendar that the user is asked to duplicate is partially shown in Figure A.1. The names of the months are in the top row. The leftmost column contains the dates of the month (1 to 31). Each body cell contains the day of the week on that month and date.

The *calendar* task appears in the user evaluation in Chapter 7. The instructions the user is given and a complete depictions of the calendar are reproduced in Appendix C.4. Variants 3 and 4 of the calendar task in the user evaluation are identical to this task, while variants 1 and 2 use a slightly simpler calendar.

A.3 Copying files

Domain: Operating system

Source: Literature (Potter and Maulsby, 1993, p. 551; Maulsby, 1994)

User task: Copy a set of files from one folder into another.

Background: The user is given a folder of files and asked to copy those that match some criteria into another folder. Two variations are suggested in the literature. Halbert suggests files whose names contain the substring *canyon*. Maulsby suggests files whose name ends in *.ps* and whose type is anything other than *Write* (Maulsby, 1994, p. 132).

A.4 Copying files to floppy

Domain: Operating system

Source: Literature (Potter and Maulsby, 1993, p. 550)

User task: Copy a set of files onto a floppy disk.

Background: The user is given an ordered list of files on a hard disk, and asked to copy them onto a number of floppy disks. As many files as possible must be stored on each disk, and no file is to be split across two disks.

	January	February	March	. . .
1	Wed	Sat	Sat	
2	Thu	Sun	Sun	
3	Fri	Mon	Mon	
4	Sat	Tue	Tue	
5	Sun	Wed	Wed	
. . .				

Figure A.1 The calendar to be duplicated in the *calendar* task (reproduced in full in Appendix C.4).

Potter supplies sample data for this problem that are unlikely to appear on a modern computer. Instead, we use a set of 196 image files, ranging in size from 16 kilobytes to 48 kilobytes, that take up 6.6 megabytes of disk space. These are to be copied onto five standard 1.4 megabyte floppy disks.

A.5 Copying mail headers

Domain: Email client, text editor

Source: Literature (Cypher, 1993b, also Potter and Maulsby, 1993, p. 559)

User task: Create a numbered list of the subjects of email messages.

Background: Every email message has a number of fields, including a subject line. The user is given a set of email messages, and asked to create a numbered list of the subjects of the messages. The list of Subject lines are shown in Figure A.2a, and the list in Figure A.2b.

This task (including the data in Figure A.2) was used to illustrate the Eager PBD system (Cypher, 1993b). In its original form, the email messages and text list were implemented in HyperCard.

A.6 Image conversion

Domain: Operating system, FTP client, text editor, graphic editor

Source: User (July 22, 1997)

User task: Download JPEG files, convert to PICT, and return to FTP site.

Background: A set of Macintosh PICT image files are stored on a Unix system in Hamilton, and must be converted to JPEG format and stored on the Unix system in Calgary. The user is asked to download each file, convert it to JPEG format,

(a)
Subject: Trial Info
Subject: Some more good ideas
Subject: a necessary evil
Subject: Fitness centre re-opens
Subject: Meeting this week
Subject: Lost data, can't find
Subject: Where were you

(b)
1. Trial Info
2. Some more good ideas
3. a necessary evil
4. Fitness centre re-opens
5. Meeting this week
6. Lost data, can't find
7. Where were you

Figure A.2 The *copying mail headers* task, showing (a) the data and (b) the finished list (adapted from Cypher, 1993b).

upload the JPEG file to the second Unix system, and delete the local JPEG and PICT files.

The user has a similar problem with Microsoft Word for the Macintosh files, which must be converted to a Windows-compatible format on the Macintosh.

This task is described in more detail in Section 1.3.1, and appears in the evaluation in Chapter 6 (variants 3 and 4 of the *image conversion* task).

A.7 Indexing document files

Domain: Operating system, database, and other applications

Source: User (March 4th, 1997)

User task: Create a catalogue of the files in a computer system.

Background: Many large collections of documents that are poorly described by their name and appearance in the operating system, particularly in operating systems with limited filenames. One solution is to build a database of document metadata to help find obscure files. The user is asked to create a simple database associating the name and path of every document on an office computer with information describing it.

The database is to have fields for each document's reference number, name, path, author, date, application, type (letter, minutes, etc) and subject. The documents are in a range of formats, though most are in Microsoft Word, Microsoft Excel, and Microsoft Publisher format. They are in located in many places on the computer in a haphazard fashion, and though they are usually grouped in directories, some are interspersed with files she is not interested in (such as executable program files).

The user must iterate over the document files, examine each, create a database record for it, and fill in the database fields with the appropriate information. The

task involves many irregularities and exceptions: many files are in a consistent format, but others are not, and some should be ignored altogether. The users judgement will be required to describe a document's subject and guess who might have been the author. These inferences are all but impossible for a computer to make, but the task nonetheless has iterative elements that can be automated: finding and opening documents, creating records, entering the name, path, date, and application.

This task is described in Section 1.3.3.

A.8 Joining document sections

Domain: Web browser, text editor

Source: Author

User task: Compile a single text document from a set of web pages.

Background: Many technical documents on the world wide web are split into several pages, with a section on each page. They are not provided in a complete form, making it difficult to print or search the entire document. In this task, the user is asked to visit every section of a large document in order, and to copy each page into a single text editor document.

Sectioned documents usually have a table of contents with links to every page. Each page can have a section number, section title, text describing the page, links to sub-sections, links to related external pages, and navigation links (to the next, previous, above, index, and author pages). Examples (in 1997) include:

- *The Common Lisp Hyperspec.* This document contains a number of sections that have no text, only sub-section titles.
  http://www.harlequin.com/books/HyperSpec/FrontMatter/Chapter-Index.html
- *GNU Make.*
  http://www.debian.org/Documentation/texi/make/make_toc.html
- *HTML by Example.* This is a 500 page book; other books have a similar format.
  http://www.mcp.com/que/bookshelf/
- *The AppleScript Language Guide: English Dialect.* This is the second chapter of a larger document.

http://developer.apple.com/techpubs/macos8/InterproCom/AppleScriptScripters/applescriptscripters.html

A.9 Mail merge

Domain: Text Editing, text editing

Source: Application (Potter and Maulsby, 1993, p. 560 is identical)

User task: Print a form letter for a list of people.

Background: The user is given two word processor files. The first, partially shown in Figure A.3a, contains a list of names and addresses. The data for this list was generated by searching the *Internet Address Finder*, a publicly available address database on the world wide web, for "John Smith" and retrieving the results with a physical address. In retrospect, this data is very regular, and a more diverse set of names would be more realistic. The second file contains a letter template, with a space in which the recipient's name and address can be inserted (Figure A.3b). The user's task is to select each name and address in turn, copy them into the form letter, and print the letter.

The "mail merge" tool in many modern word processors can be used to send form letters, but is difficult to learn and use.

A.10 Network diagram

Domain: Drawing

Source: User (March 4th, 1997)

User task: Draw a network diagram.

Background: A user has been asked to draw a network diagram for an engineering firm. He has been given a text file describing the three servers and 24 client computers on the network, shown in Figure A.4. The finished diagram will look like the one in Figure A.5.

(a)

John Smith
Westholme Partners, L.P.
39th Floor, Tower 45
120 West 45th Street
New York, NY 10036

John Smith
Evolving Systems, Inc.
8000 E Maplewood Ave
Englewood, CO 80111

John J Smith
7921 Woodruff Court
Woodbridge, VA 22151

John J Smith
Tracor Aerospace, Inc.
6500 Tracor Lane
Austin, TX 78725-2050

…

(b)

Gordon Paynter
The University of Waikato
Hamilton,
New Zealand

February 26, 2000

[insert name and address here]

Dear Sir or Madam,
Text of polite letter. Text of polite letter.
Text of polite letter… Text of polite letter.

Yours sincerely

Gordon Paynter

Figure A.3 Data for the *mail merge* task, showing (a) part of the address list, and (b) the form letter.

A.11 Numbering table of contents

Domain: Text editing

Source: User (February 26, 1997; also see Potter and Maulsby, 1993, p. 563)

User task: Add section numbers to a table of contents.

Background: Figure A.6a shows a section of the table of contents of a Masters thesis. Each section starts with a (numbered) chapter heading, and has a hierarchical set of subheadings, which may be up to four levels deep. Each subheading is preceded by a "+" then a tab character for each level of depth. The user is asked to reformat the chapter heading and then the subheadings, removing the "+" symbol form each, and replacing it with the section number. Part of the finished table of contents is shown in Figure A.6b (page numbers and tabulation were added later and are not part of the task).

The task is complicated by several factors.

- All the chapters except the references section are numbered
- Only lines starting with a "+" are headings, the rest are comments and should be deleted.
- Some lines are too long and should be wrapped and tabbed attractively.

Server ID	Net ID	Administrator	Domain Name	Computer Type
Accounting	0	Ivan Cassidy	ACC	COMPAQ DX2/50 160M
Drawing	0	Karen Summers	DRA	HP VECTRA
Engineering	0	Mary Clark	ENG	HOUSTON PENTIUM

Computer ID	Net ID	Name of User	Username	Computer Type
ACC_00	1	Ivan Cassidy	ICASSIDY	HOUSTON PENTIUM
ACC_01	2	Peter Hayes	PHAYES	HP VECTRA
ACC_02	3	Elaine Kerr	EKERR	HOUSTON PENTIUM
ACC_03	4	Janet Stevens	JSTEVENS	IBM PS2 260M
ACC_04	5	Elvis Murray	EMURRAY	COMPAQ PRESARIO
DRA_00	1	Norm Jones	NJONES	IBM PS2 260M
DRA_01	2	Deanne Wright	DWRIGHT	IBM PS2 260M
DRA_02	3	Jon Nelson	JNELSON	HP VECTRA
DRA_03	4	Karen Summers	KSUMMERS	IBM PS2 260M
ENG_00	1	Mary Clark	MCLARK	HOUSTON PENTIUM
ENG_01	2	Burt Masters	BMASTERS	HP VECTRA
ENG_02	3	Leslie Bain	LBAIN	COMPAQ DX2/50 160M
ENG_03	4	Alfie Rodgers	ARODGERS	COMPAQ PRESARIO
ENG_04	5	Patrick Collins	PCOLLINS	HOUSTON PENTIUM
ENG_05	6	Susan Towers	STOWERS	HP VECTRA
ENG_06	7	Gary Peterson	GPETERSON	HP VECTRA
ENG_07	8	Liam Fowles	LFOWLES	COMPAQ PRESARIO
ENG_08	9	Katrina Wright	KWRIGHT	IBM PS2 260M
ENG_09	10	Quintin Williams	QWILLIAMS	COMPAQ PRESARIO
ENG_10	11	Frank Dean	FDEAN	COMPAQ PRESARIO
ENG_11	12	Norm Clark	NCLARK	IBM PS2 260M
ENG_12	13	Herbert French	HFRENCH	IBM PS2 260M
ENG_13	14	Reece Duggan	RDUGGAN	HP VECTRA
ENG_14	15	Veronica Moore	VMOORE	COMPAQ PRESARIO

Figure A.4 Data for the *network diagram* task.

- In practice, some titles are arbitrarily edited when they are reformatted (the user explained that he made the changes “just for the hell of it”).

This task has numerous exceptions and special cases. However, it is still largely repetitive and the original user repeatedly expressed his dissatisfaction at having to perform it manually.

A.12 Printing odd and even pages

Domain: Text editor, or any other application with multi-page documents.

Source: Literature (Potter and Maulsby, 1993, p. 562)

User task: Print the odd pages of a document, then the even pages in reverse order.

Background: It is possible to simulate double-sided printing on a single-sided laser printer by first printing the odd pages, then turning the paper upside down

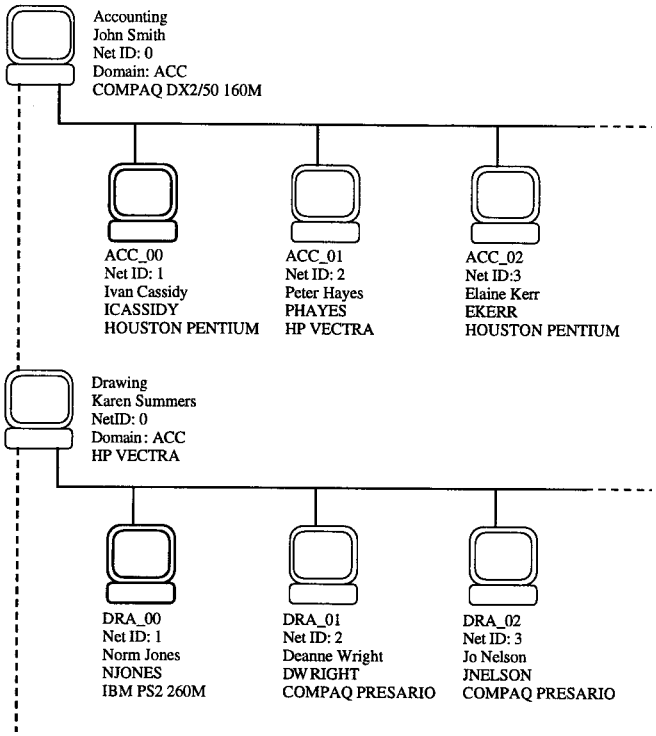


Figure A.5 Part of the finished *network diagram*..

and returning it to the printer, then printing the even pages in reverse order. The user is asked to print a document in this manner.

A.13 Program editing

Domain: Text editing, spreadsheet

Source: User (30 November, 1996)

User task: Create a set of macros in a script editor.

Background: The user is asked to convert an introductory spreadsheet tutorial from one format to another, rewriting the existing macros as scripts in the new format. The task uses the spreadsheet application's script editor.

The tutorial screen consists of a small financial spreadsheet, a button, and some explanatory text. The student is expected to read the text, follow its instructions, then press the button to indicate they have finished. Pressing the button invokes a script that updates the instructions.

The tutorial has ten parts, with corresponding scripts. The scripts are nearly identical, so the user recorded the first and named it *Step2()*. It is shown in Figure

(a)
```
+1. Introduction
    +Goals and Objectives of the Research
    +Research Approach
    +Contributions of the Research
    +Related Research
        +The COMIC Project
    +Structure of this Thesis
        insert diagram showing general structure (inc. models that don't
        really fit in) mention that since this is largely a lit. rev. that it was
        deemed unnecessary to ref every little thing, and that refs are given
        in general
```

(b)

Chapter 1	Introduction		1
1.1	Goals and Objectives of this Research		1
1.2	Research Approach		1
1.3	Contributions of this Research		4
1.4	Related Research		4
	1.4.1	The COMIC Project	4
1.5	Structure of the Thesis		6

Figure A.6 One chapter from the *numbering table of contents* task, (a) before and (b) after completion.

A.7. He realised it was easier to repeatedly copy and paste the macro than to record it for each step. He therefore copied the macro, moved to the end of his document, pasted it, changed the title from *Step2()* to *Step3()*, changed the button action from *Step3* to *Step4*, and changed the *Range* from *A8:A13* to point to the new text. He repeated these actions nine times to get his basic spreadsheet. On the last repetition, the button action was changed to an empty string.

A.14 Saving search results

Domain: Web browser

Source: User (November, 1996)

User task: Search the internet for postscript files and save the results.

Background: A researcher wants to download a large number of postscript files from the internet as the basis of a digital library. He uses the Alta Vista search engine to search for pages with links to postscript files (specifically, perform an advanced search for link:ps.Z and request compact output). The search, performed in 1996, returns approximately 3000 result pages like the one in Figure A.8. Each contains ten links to web pages; and each of these contains one or more links to postscript files. The user is asked to save all these postscript files.

The files must be saved in postscript format, but their filenames are unimportant.

```
Sub Step2()
        ActiveSheet.DrawingObjects("Button 2").OnAction = "Step3"
        Range("A14:A22").ClearContents
        Sheets("B").Range("A8:A13").Copy
        Sheets("A").Range("A14").PasteSpecial Paste:=xlAll, Operation:=xlNone,
        SkipBlanks _
                :=False, Transpose:=False
        Application.CutCopyMode = False
End Sub
```

Figure A.7 The Step2() macro for the *program editing* task.

A.15 Sorting rectangles

Domain: Drawing

Source: Literature (Witten and Maulsby, 1993)

User task: Position a set of rectangles in order of increasing height.

Background: A set of rectangles are created in a drawing editor. The user is asked to arrange them horizontally, evenly spaced and sorted by height.

This task is used to evaluate Metamouse with four rectangles of equal width (Mauslby and Witten, 1993). Maulsby describes a more complex variation, where the ten boxes can have the same height but different widths (Potter and Maulsby, 1993, p. 577).

A.16 Subtotal

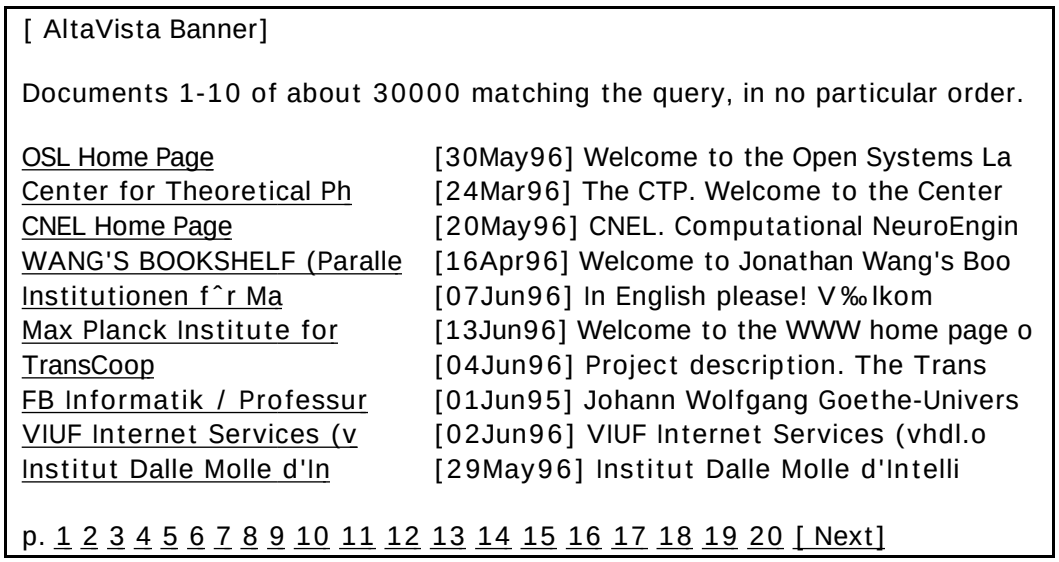
Domain: Spreadsheet

Source: Application (Microsoft Excel's *subtotal* tool)

User task: Insert subtotals to a column of data values.

Background: A spreadsheet consists of two columns of data: the first contains variable sized groups of key values (strings or numbers), and the second a set of values. The user is asked to create a subtotal for each group of values, exactly reproducing the behaviour of Microsoft Excel's *subtotal* tool.

Figure A.9 shows the spreadsheet before (Figure A.9a) and after (Figure A.9b) the task.


[AltaVista Banner]

Documents 1-10 of about 30000 matching the query, in no particular order.

OSL Home Page	[30May96] Welcome to the Open Systems La
Center for Theoretical Ph	[24Mar96] The CTP. Welcome to the Center
CNEL Home Page	[20May96] CNEL. Computational NeuroEngin
WANG'S BOOKSHELF (Paralle	[16Apr96] Welcome to Jonathan Wang's Boo
Institutionen fˆr Ma	[07Jun96] In English please! V‰lkom
Max Planck Institute for	[13Jun96] Welcome to the WWW home page o
TransCoop	[04Jun96] Project description. The Trans
FB Informatik / Professur	[01Jun95] Johann Wolfgang Goethe-Univers
VIUF Internet Services (v	[02Jun96] VIUF Internet Services (vhdl.o
Institut Dalle Molle d'In	[29May96] Institut Dalle Molle d'Intelli

p. 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 [Next]

Figure A.8 AltaVista search results for “link:ps.Z” in text mode used in the *saving search results* task.

A.17 Truncate lines

Domain: Drawing

Source: Literature (Potter and Maulsby, 1993, p. 576)

User task: Truncate a set of lines where they intersect another object.

Background: The user is given a drawing containing a set of parallel lines that all intersect another object. They are asked to iterate over the parallel lines truncating each at the point where they intersect the other object.

In this example, the parallel lines represent cables in a picture of a bridge, and are truncated where they intersect its arch. Bhavnani and John (1998) discuss a similar task, performed by an actual user, where the lines delimit ceiling tiles, and are cut where they overlap rectangles representing ceiling vents. Potter (1993a, p. 374) describes another, more abstract, example.

A.18 Discarded tasks

Some of the tasks are not iterative tasks, or are otherwise unsuitable for the Gedanken experiment described in Chapter 7.

a

A	B
1	4
1	5
1	2
1	7
2	8
2	1
2	9
3	8
3	2
3	3
3	5
3	3
4	1
4	8
4	7

b

A	B
1	4
1	5
1	2
1	7
Subtotal	=SUM(B1:B4)
2	8
2	1
2	9
Subtotal	=SUM(B6:B8)
3	8
3	2
3	3
3	5
3	3
Subtotal	=SUM(B10:B14)
4	1
4	8
4	7
Subtotal	=SUM(B16:B18)

Figure A.9 The *subtotal* task, (a) before, and (b) after (with formulas displayed).

A.18.1 Fractal snowflake

Domain: Drawing

Source: Literature (Potter and Maulsby, 1993, p. 556)

User task: Draw a fractal snowflake (a triadic Koch curve).

Background: A fractal snowflake can be created by drawing a picture of an equilateral triangle, and then transforming it by replacing every line with 4 connected line segments. Figure A.10 illustrates this transformation on a single line: the segment in Figure A.10a is replaced by the four smaller segments in Figure A.10b. The transformation is repeated until every line segment is shorter than a predetermined minimum length.

This task can be framed as a nested iterative task. In each pass the transformation is applied to the entire picture, and in each transformation every line is replaced by four shorter lines.

This task is not included in the task evaluation because the user would not normally attempt to draw it by hand, they would instead use a specialised tool or write a program. As Kurlander notes, “it is difficult to draw good approximations of this shape by hand, because of the vast number of lines that

Figure A.10 The transformation applied in the *Fractal snowflake* task.

need to be positioned precisely. This task has been cited as an example of why programming interfaces are useful in graphical editors" (Potter and Maulsby, 1993, p. 556).

A.18.2 Intelligent image filtering

Domain: Web Browser

Source: Author (application)

User task: Load selected images on a web page.

Background: Web browsers can be instructed not to automatically load graphics into a web page. The user can later command the browser to load all the images, or load the images individually. This was a common strategy in the early days of the world wide web, when network connections tended to be slower (particularly in countries geographically distant from America and Europe).

This task assumes the user is browsing with images disabled, and asks the web browser to anticipate which images on a web page the user will want to view, and load them without instruction.

For example, the browser might learn to load images that

- have a URL that matches "*bullet*.gif",
- are used more than twice on the page,
- are used more than three times on the last 5 pages, or
- have a URL that matches "http:/www.unitedmedia.com/dilbert/dilbert*.gif"

This task was not included in the Gedanken experiment of Chapter 7 because it is not iterative; it is executed in response to an event.

A.18.3 Manipulating checklists

Domain: Apple Newton 2.0 Operating System

Source: Author

User task: Promote every entry in a nested checklist.

x	Books to buy		
	x	Alfred Bester	
		x	The stars my destination
	x	Phillip K. Dick	
		x	A handful of darkness
		x	A scanner darkly
		x	Collected Letters
		√	Flow My Tears...
		x	Gather yourselves together
		x	I hope I shall arrive soon
		x	The Divine Invasion
		√	Ubik
	x	John Fowles	
		x	The Collector
		x	The Aristos
		√	French Lieutenant's Woman
		√	A Maggot
	x	Tim Powers	
		x	The Annubis Gates
			…

Figure A.11 Part of the data for the *manipulating checklists* task.

Background: The Apple Newton handheld computer provides several types of stationary in its notepad application. These include a checklist, a hierarchically structured list where every item may have a number of sub-items, and can be optionally be marked with a tick (√).

The checklist interface does not allow the user to promote or demote objects in the hierarchy. In fact, the interface is extremely minimal, and an entry cannot be copied, though the label of an entry can be copied as a simple text fragment.

The task is to take the Newton checklist in Figure A.11, and to promote every entry (except the topmost) by one level without losing the hierarchical structure.

This task has been omitted from the Gedanken experiment in Chapter 7 because it is too specific to the Newton operating system to be realistically duplicated on another platform. Although manipulating hierarchies is a universal problem, and can potentially be framed as an iterative problem, this particular task is tedious because of the peculiarities of the Newton implementation, whose interface forces the user to perform the task in a difficult and unintuitive manner.

B Training tasks

This Appendix describes the set of iterative tasks used to train the classifiers for guiding prediction described in Section 5.2.

Seven iterative tasks were used to train the classifiers. They are different tasks from those used to generate the dataset for testing the classifier (Section 5.2.3). The tasks were chosen to cover a range of applications and test all the pattern analysis schemes.

B.1 Label by size

The *letters* folder contains 26 files named for the letters of the alphabet. The task is to set the label of each file according to its size. Files 4KB or less will be labelled orange, 8KB or less green, 96KB or less yellow, 200KB or less blue, and the remainder red.

B.2 Label by kind

The *extra files* folder contains 100 irregularly named files. There are six different kinds of file: *Alias*, *BBEdit text file*, *Internet Explorer document*, *MCL document*, *Microsoft Word document*, and *Text document*. The task is to label files with a different label for each *kind* of file.

B.3 Hendry & Green

The spreadsheet document *Hendry & Green* contains a list of 500 records. The records are stored as numbers in column A and B. The task is to compute the average of column B for each of the ten blocks in the first ten cells of column C.

This task is drawn from Hendry and Green (1994) and described in Appendix A.1 and Section 1.3.2.

B.4 Hendry & Green extended

This task is very similar to the *Hendry and Green* task (Appendix B.3). The spreadsheet file *Hendry & Green extended* contains a list of 3000 records in blocks of 50, similar to the smaller task described above. The task is to compute the average of column B for each of the 60 blocks in the first 60 cells of column C.

B.5 Resize tiles

The folder *tiles to resize* contains a list of image files. These files are all JPEG files, and all the pictures are approximately 60 pixels square. The task is to open each of the files, use GIFConverter to resize each image to 100 pixels square, and save the modified version over the old version.

B.6 Move and rename files

The folder *new tiles* contains a list of image files. The files are named for the days on which they were created, but not in a regular manner. The folder *new files numbered* is empty. The task is to copy each file from the *new files* folder into the *new files numbered* folder and rename it. The files should be named *tile63.jpeg*, *tile64.jpeg*, *tile65.jpeg*, and so on.

B.7 Position files

The folder *tutorial example* contains a group of files named for the letters of the alphabet. They are not sorted in any particular order. The task is to arrange the files into a line across the window in name order. This task is the same as the tutorial example in Appendix D.

C Attributes for ***PAS-ML***

This Appendix contains the full list of attributes the *PAS-ML* pattern analysis scheme examines during the task described in Section 5.3. The attributes are generated dynamically from the pattern history and the event trace, so different tasks have different attribute lists.

In Table 5.4, the attribute with the highest significance is called *size of selection*. Internally, it is known as

("now" "prev-step" "event" "select" "physical size" "evaluation")

and appears on line 64 of the list below.

The internal representation describes the path that must be followed through the instance data from the current event to find the attribute's value. The attribute described above is not, strictly speaking, the *size* of the *selection*, it is the evaluation of the *physical size* of the *select* parameter value of the event in the previous step of the cycle. The internal representation of the size of the current selection is

("now" "data" "application \"Finder\"" "selection" "physical size" "evaluation")

and does not appear on the list below.

Section 5.3.1 explains that Familiar adds the most recent attributes from the instance information to the training dataset in batches of 20. This is not strictly true, for two reasons. First, *PAS-ML* ignores attributes that it cannot evaluate, so some operations add fewer than 20 attributes. The size of the current selection (above) is not included in the list below for this reason. Second, there are occasional ties for the "most recent" attribute which are broken by including more than one attribute at a time. Consequently, Familiar sometimes adds more than 20 attributes in a single batch. In this example, three batches of attributes are added to the training data, comprising those on lines 1–7, lines 8–42, and lines 43–71 respectively.

The list of attributes in the training dataset, in the order they were added, is shown below. Familiar is implemented in Lisp, so the list is in Lisp syntax.

1. (('now' 'prev-iter' 'event')
2. ('now' 'prev-step' 'event')
3. ('now' 'prev-iter' 'event' 'set')
4. ('now' 'prev-iter' 'data' 'application \"Finder\"')
5. ('now' 'prev-step' 'event' 'select')
6. ('now' 'prev-step' 'data' 'application \"Finder\"')
7. ('now' 'prev-iter' 'event' 'to')
8. ('now' 'prev-iter' 'data' 'application \"Finder\"' 'startup disk')
9. ('now' 'prev-iter' 'data' 'application \"Finder\"' 'folder \"letters\"')
10. ('now' 'prev-iter' 'data' 'application \"Finder\"' 'selection')
11. ('now' 'prev-step' 'data' 'application \"Finder\"' 'startup disk')
12. ('now' 'prev-step' 'data' 'application \"Finder\"' 'folder \"letters\"')
13. ('now' 'prev-iter' 'data' 'application \"Finder\"' 'date \"Thursday, 23 April 1998 2:00:20 PM\"')
14. ('now' 'prev-iter' 'data' 'application \"Finder\"' 'date \"Tuesday, 20 January 1998 9:36:23 AM\"')
15. ('now' 'prev-step' 'event' 'select' 'information window')
16. ('now' 'prev-step' 'event' 'select' 'window')
17. ('now' 'prev-step' 'data' 'application \"Finder\"' 'date \"Saturday, 2 May 1998 1:12:57 PM\"')
18. ('now' 'prev-step' 'data' 'application \"Finder\"' 'date \"Tuesday, 20 January 1998 9:36:39 AM\"')
19. ('now' 'prev-step' 'event' 'select' 'container')
20. ('now' 'prev-step' 'event' 'select' 'id')
21. ('now' 'prev-step' 'event' 'select' 'index')
22. ('now' 'prev-step' 'event' 'select' 'name')
23. ('now' 'prev-step' 'event' 'select' 'bounds')
24. ('now' 'prev-step' 'event' 'select' 'position')
25. ('now' 'prev-step' 'event' 'select' 'folder')
26. ('now' 'prev-step' 'event' 'select' 'disk')
27. ('now' 'prev-step' 'event' 'select' 'comment')
28. ('now' 'prev-step' 'event' 'select' 'description')
29. ('now' 'prev-step' 'event' 'select' 'kind')
30. ('now' 'prev-step' 'event' 'select' 'label index')
31. ('now' 'prev-step' 'event' 'select' 'modification date')
32. ('now' 'prev-step' 'event' 'select' 'creation date')
33. ('now' 'prev-step' 'event' 'select' 'physical size')
34. ('now' 'prev-step' 'event' 'select' 'size')
35. ('now' 'prev-step' 'event' 'select' 'creator type')
36. ('now' 'prev-step' 'event' 'select' 'file type')
37. ('now' 'prev-step' 'event' 'select' 'content space')
38. ('now' 'prev-step' 'event' 'select' 'selected')
39. ('now' 'prev-step' 'event' 'select' 'version')
40. ('now' 'prev-step' 'event' 'select' 'product version')
41. ('now' 'prev-step' 'event' 'select' 'stationery')
42. ('now' 'prev-step' 'event' 'select' 'locked')
43. ('now' 'prev-iter' 'data' 'application \"Finder\"' 'folder \"letters\"' 'file \"a\"')
44. ('now' 'prev-iter' 'data' 'application \"Finder\"' 'selection' 'label index')

45. ('now' 'prev-step' 'event' 'select' 'name' 'evaluation')
46. ('now' 'prev-step' 'event' 'select' 'information window' 'evaluation')
47. ('now' 'prev-step' 'event' 'select' 'window' 'evaluation')
48. ('now' 'prev-step' 'data' 'application \"Finder\"' 'folder \"letters\"' 'file \"b\"')
49. ('now' 'prev-step' 'event' 'select' 'disk' 'evaluation')
50. ('now' 'prev-step' 'event' 'select' 'container' 'evaluation')
51. ('now' 'prev-step' 'event' 'select' 'id' 'evaluation')
52. ('now' 'prev-step' 'event' 'select' 'index' 'evaluation')
53. ('now' 'prev-step' 'event' 'select' 'label index' 'evaluation')
54. ('now' 'prev-step' 'event' 'select' 'bounds' 'evaluation')
55. ('now' 'prev-step' 'event' 'select' 'position' 'evaluation')
56. ('now' 'prev-step' 'event' 'select' 'folder' 'evaluation')
57. ('now' 'prev-step' 'event' 'select' 'size' 'evaluation')
58. ('now' 'prev-step' 'event' 'select' 'comment' 'evaluation')
59. ('now' 'prev-step' 'event' 'select' 'description' 'evaluation')
60. ('now' 'prev-step' 'event' 'select' 'kind' 'evaluation')
61. ('now' 'prev-step' 'event' 'select' 'selected' 'evaluation')
62. ('now' 'prev-step' 'event' 'select' 'modification date' 'evaluation')
63. ('now' 'prev-step' 'event' 'select' 'creation date' 'evaluation')
64. ('now' 'prev-step' 'event' 'select' 'physical size' 'evaluation')
65. ('now' 'prev-step' 'event' 'select' 'locked' 'evaluation')
66. ('now' 'prev-step' 'event' 'select' 'creator type' 'evaluation')
67. ('now' 'prev-step' 'event' 'select' 'file type' 'evaluation')
68. ('now' 'prev-step' 'event' 'select' 'content space' 'evaluation')
69. ('now' 'prev-step' 'event' 'select' 'version' 'evaluation')
70. ('now' 'prev-step' 'event' 'select' 'product version' 'evaluation')
71. ('now' 'prev-step' 'event' 'select' 'stationery' 'evaluation'))

D User evaluation instructions

This Appendix contains the instructions given to the participants in the user evaluation conducted during the month of August in 1998.

Chapter 7 describes the experimental procedure. The instructions consist of:

- a pre-experiment questionnaire about the participants experience with computers and the applications in the evaluation;
- the Familiar tutorial;
- the Familiar detailed information sheet;
- the task description of the four variants of the calendar task;
- the task description of the four variants of the image task; and
- an initial set of questions for the post-experiment interview.

The user saw each document except the questions for the post-experiment interview. These were read out to the participant, and were amended and extended as the interviewer deemed appropriate.

D.1 The pre-experiment questionnaire

Participants in the evaluation were asked to fill in the questionnaire below before attempting the tasks.

I would like to know how much experience you have with the following products. Please estimate how long you have been using each, and how many hours per week you spend using each.

Name (optional): _____

Product	**Experience**	**Weekly use**
Computers	*years/months*	*hours*
The Apple Macintosh "Finder"	*years/months*	*hours*
Other graphical interfaces (like Windows)	*years/months*	*hours*
The Microsoft "Excel" spreadsheet	*years/months*	*hours*
Other spreadsheets	*years/months*	*hours*
Apple SimpleText	*years/months*	*hours*
Other Text Editors and Word processors	*years/months*	*hours*
The "Fetch" FTP client	*years/months*	*hours*
Other FTP programs	*years/months*	*hours*
GIFConverter	*years/months*	*hours*
Other image manipulation programs	*years/months*	*hours*

D.2 The Familiar tutorial

Each participant was asked to complete the following tutorial during the user evaluation.

What is Familiar?

Familiar is a program that helps you perform repetitive tasks on the Macintosh computer.

Familiar is designed to help you with “cyclical” tasks. These are tasks where you find yourself carrying out the same series of actions again and again.

Familiar doesn’t work in every situation and application, and even when it does there may be alternatives to Familiar that are more suitable for your particular task.

What does Familiar do?

Familiar lets you solve problems *by demonstration*. When you encounter a cyclical task, you can demonstrate it to Familiar, and Familiar will attempt to learn the task and help you finish it.

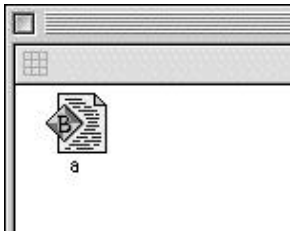
To automate a task, activate Familiar and then start performing the task. Familiar records your actions, remembers them, and looks for patterns in them. When it finds a pattern it tries to extend that pattern and predict what you will do next. If the predictions are correct you can ask Familiar to carry them out.

How do I use Familiar?

The best way to see how Familiar is used is to work through a simple cyclical task: lining up all the files in a folder. Open the “tutorial example” folder on the desktop in the Finder.

Next to the *Help* menu on your menubar you will find a *Familiar* menu. To start repetitive task, go to the Familiar menu and choose *Start recording*. The Familiar program will display the *Familiar status* window. This window tells you what Familiar is doing.

We have to demonstrate the repetitive task to Familiar. Go back to the “tutorial example” folder in the Finder and put file “a” in the top left hand corner of the window like this:



Now return to Familiar. You will see that just below the *status* window is the *Familiar history* window. The *history* window shows every command that Familiar has seen you perform. It will look something like this:

1 : activate -- Finder

2: select file “a” of folder “tutorial example”

3: set position of selection to {1 ,0}

These statements reflect what you have done. First you activated the Finder, then you selected file “a,” and then you set the position of that file (the selection). The numbers show you what order the commands were performed in.

Familiar is sometimes quite slow and has trouble keeping up with you if you work too fast. The *status* window will tell you what Familiar is doing. The last few lines will look something like this:

Recorded command 3

parsing "set position of selection to {1 ,0}"...

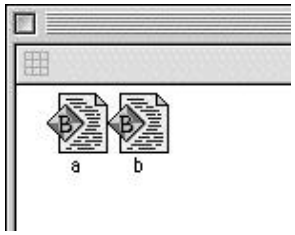
adding "set" to history...

predicting future events...

Waiting for the next command.

Familiar reports each of these steps as it does them. When Familiar says it is waiting for the next command you can carry on with your demonstration. If familiar has not finished processing the command, you should wait for Familiar to catch up. You could just go on performing new commands and let Familiar catch up later, but Familiar works better if you don’t let it fall too far behind.

Return to the Finder and move the file “b” so that it is next to file “a” like this:

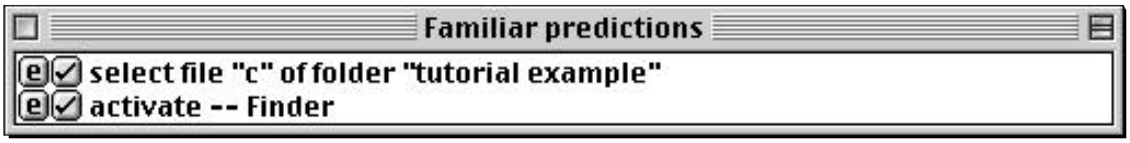


If you return to Familiar you will see something like this added to the *history* window:

4: select file “b” of folder “tutorial example”

5: set position of selection to {33,0}

Below the *history* window will be a new window, titled *Familiar predictions*. The *predictions* window is where familiar displays its predictions about what you will do next. The predictions should look like this:

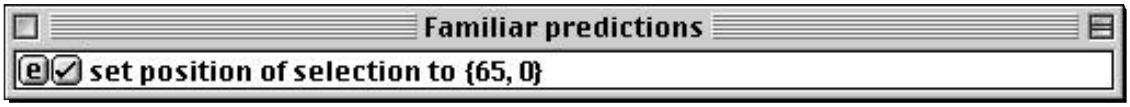


Only one of these predictions is correct—the next action you want to take is to select file “c.” Familiar uses lots of techniques to make predictions, and often these techniques give different results or predict incorrectly, especially in cases like this where it has only seen a few cycles of the task.

Since the top prediction is correct, you can ask Familiar to perform the command for you. There are two ways to do this: you can click on the text of the command, or on the *tick* button right next to it.

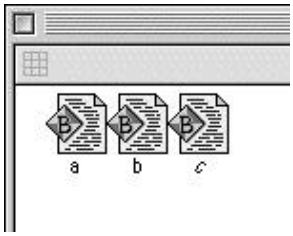
Click on the *select* command. Several things will happen now. First, the text of the command will be turned orange, indicating that the command has been selected and sent to the application. Second, the application will carry out the command. You should be able to see file “c” being selected in the background. The command will be processed by Familiar, and the usual messages will be displayed in the *status* window. The command will be added to the *history* window and new the *prediction* window will be refreshed with predictions for the next command. When Familiar has finished processing it will report it in the *status* window as usual.

Now Familiar will make a single prediction in the *prediction* window.

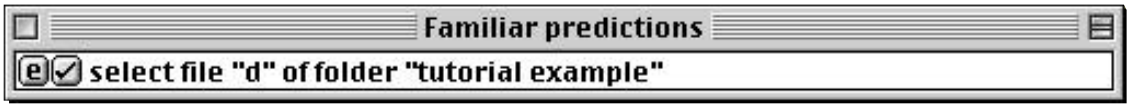


If you look at the history window you will see that the position of the first two files was set to {1,0} and {33,0}. This prediction will change the position by the same distance (32 units) so it is probably correct. Try clicking on the prediction.

The Familiar interface will be updated and the selection will be repositioned like this:

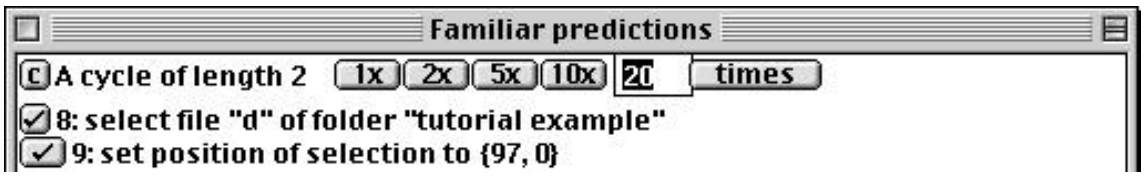


Familiar then will display this new prediction:



This is the correct prediction, and it would be easy to select it, but asking Familiar to execute one step at a time is not much use—it is probably faster to position the files yourself. What we would like to do is have Familiar perform more than one step.

Next to the *tick* button in the *prediction* window is the *expand* button (labelled "e"). We use this to see predictions of entire cycles of events. Try clicking on the *expand* button next to the prediction. Familiar will display all the repetitive cycles it has found that start with that command. In this case there are two, and the first cycle looks like this:



Below it is another cycle. We are not interested in the other cycle, and will ignore it (it will go away on its own). The first cycle predicts that the next step in the task (step 8) is to select file "d" and that the step after that is to reposition the selection to {97, 0}. This is indeed what we want to do.

There are two ways to make Familiar perform a complete cycle of the task. The first is to click on the last command in the cycle (in this case, number 9). Clicking on the first command in a cycle will only execute that command. The other is to click on the *one times* button (labelled "1x"). This will execute the cycle one time. Try clicking on the last command.

Familiar will execute both the commands in the cycle one after the other. Each command will be coloured orange before it is sent to the application, and the *history* and *status* windows are updated as though you had performed the command yourself. When the two commands have been executed, Familiar will display these predictions:

10: select file "e" of folder "tutorial example"

11 : set position of selection to {129,0}

To make Familiar perform more than one command at a time, you can use the buttons labelled *two times* ("2x"), *five times* ("5x"), and *ten times* ("10x") to repeat the cycle two, five, or ten times respectively. Try clicking on the *two times* button. Familiar will position another two files.

Familiar's interface includes a field for typing in a specific number of cycles. If you want to execute the cycle 12 times, for example, you can enter 12 in the text field and press the *times* button next to it. When you ask Familiar to perform several cycles, the field displays the number of cycles to go.

Familiar will run for as many iterations as you have specified or until you stop it or until one of the commands it tries to execute makes no sense. Sometimes Familiar doesn't stop when you think. For example, what does Familiar do in the cycle after it repositions file "z?"

When you have finished demonstrating the task choose *Stop recording* from the *Familiar* menu to turn Familiar off. When you want to start a new task, choose *Reset Familiar* from the *Familiar* menu.

Tips for using Familiar

Familiar can detect lots of different patterns in your actions, but does make mistakes. Here are a couple of tricks for making Familiar work better.

- **Work in numeric or alphabetical order**. Familiar can learn many "out of order" patterns but it is slower to learn this way.

- **Try not to make mistakes when you are demonstrating a task.** Familiar will usually figure out that you've made a mistake eventually, but starts out by assuming you have a reason for everything you do.
- **Reset Familiar if you are going to start a new task.** You can reset Familiar by choosing *Reset Familiar* from the *Familiar* menu. This will let Familiar know that you are starting a new task. It will also turn off recording if it is turned on.
- **Close any windows that might get in your way before you start a task.** Having a clear screen makes Familiar easier to see, but if you move windows around while you are recording you will confuse Familiar.
- **Test Familiar's predictions.** You can make sure Familiar is going to do what you expect by performing one or two cycles as a test. Remember that Familiar can make mistakes, and that you cannot simply undo the things you tell it to do.

If you are really interested in what patterns Familiar can spot and extrapolate, ask Gordon for an explanation.

D.3 Additional Familiar instructions

The “Tips for using Familiar” section of the Familiar tutorial document (Appendix C.2) concludes by informing the participant that “If you are really interested in what patterns Familiar can spot and extrapolate, ask Gordon for an explanation.” None of the participants requested further information, but if they had, they would have been given the following document.

What kinds of predictions does Familiar make?

Familiar can, and does, make many kinds of predictions. It uses several “pattern analysis schemes” to make separate predictions and displays the one that it judges most reliable. There are several patterns that the schemes recognise.

- **constants**: items that are the same from one iteration to the next, like label index of selection in the example above.
- **numeric series**: numbers that are increasing or decreasing in simple patterns like 1, 33, 65, 97....
- **alphabetic series**: letters that are increasing or decreasing regularly like a, b, c, d…
- **common enumerations**: like days of the week and months.
- **values it has seen before**: if you refer to an object or number in one step of a cycle, then refer to the same object again later, Familiar can spot the repetition.

All these can be used to extrapolate simple sequences with no special knowledge of the application where they are being used.

What about more complicated patterns?

More complicated patterns can be spotted using *contextual information* about the recording. Familiar asks the application you’re working in for this context. Fetching context is slow, but the extra data can improve Familiar’s ability to reason about what you are doing.

There are three types of context:

- **what things are**: familiar asks the application for more information about any object that does not have an obvious type. The most frequent example is the selection.
- **elements in a set**: often you want to work with a group of related objects, such as all the files in a folder or all the open windows. Once you start working with the elements in a set, Familiar can ask the application what other elements are there in the set that you might want to work on next.
- **properties of objects**: like the size, name, number, and label of a file. These properties can be used to reason about why you are doing things. Familiar can ask the application to send it every property of every object in the recording in order to help its reasoning.

By default, the first two types of context are turned on and the third is turned off. It would be best if we could leave all three types of context on all the time, but gathering the properties of objects is very slow. That is the only reason it is turned off.

As a general rule, you need context if there isn't enough information in the *history* window to make predictions. In the example you worked through context was relatively unimportant because you could predict every future action from the *history* window.

This was possible because the files were named in such a simple fashion. If the first three files had unrelated names, like "Arthur" and "Rugby" and "Mo" there would be no way to tell what the fourth file was named except to ask the application.

You can change the context that is gathered by choosing *Familiar settings* from the *Familiar* menu.

D.4 Calendar task instructions

The participants were asked to perform four variants of the calendar task, as described in Section 7.1. The instructions for each variant were printed on separate pieces of paper and handed to the user when they were asked to begin the task.

The two calendar examples referred to are reproduced on the following pages.

Calendar task 1

Use Microsoft Excel to duplicate the Calendar shown on the calendar example page. The calendar is one large spreadsheet.

It is important to get the values of the months, days, and numbers accurate, but formatting and printing your spreadsheet are not necessary.

Calendar task 2

Use Microsoft Excel to duplicate the Calendar shown on the calendar example page without using multiple selection.

Calendar task 3

Use Microsoft Excel to duplicate the calendar shown on the second example page without using multiple selection.

As with the earlier calendar, it is important to get the values of the months, days, and numbers accurate, but formatting and printing your spreadsheet are not necessary.

You may use Familiar if you want to.

Calendar task 4

Use Microsoft Excel to duplicate the Calendar shown on the calendar example page.

You may use whatever techniques you think are suitable, including multiple selection and Familiar.

	July	August	September	October	November	December
Tue			1			1
Wed	1		2			2
Thu	2		3	1		3
Fri	3		4	2		4
Sat	4	1	5	3		5
Sun	5	2	6	4	1	6
Mon	6	3	7	5	2	7
Tue	7	4	8	6	3	8
Wed	8	5	9	7	4	9
Thu	9	6	10	8	5	10
Fri	10	7	11	9	6	11
Sat	11	8	12	10	7	12
Sun	12	9	13	11	8	13
Mon	13	10	14	12	9	14
Tue	14	11	15	13	10	15
Wed	15	12	16	14	11	16
Thu	16	13	17	15	12	17
Fri	17	14	18	16	13	18
Sat	18	15	19	17	14	19
Sun	19	16	20	18	15	20
Mon	20	17	21	19	16	21
Tue	21	18	22	20	17	22
Wed	22	19	23	21	18	23
Thu	23	20	24	22	19	24
Fri	24	21	25	23	20	25
Sat	25	22	26	24	21	26
Sun	26	23	27	25	22	27
Mon	27	24	28	26	23	28
Tue	28	25	29	27	24	29
Wed	29	26	30	28	25	30
Thu	30	27		29	26	31
Fri	31	28		30	27	
Sat		29		31	28	
Sun		30			29	
Mon		31			30	

	July	August	September	October	November	December
1	Wed	Sat	Tue	Thu	Sun	Tue
2	Thu	Sun	Wed	Fri	Mon	Wed
3	Fri	Mon	Thu	Sat	Tue	Thu
4	Sat	Tue	Fri	Sun	Wed	Fri
5	Sun	Wed	Sat	Mon	Thu	Sat
6	Mon	Thu	Sun	Tue	Fri	Sun
7	Tue	Fri	Mon	Wed	Sat	Mon
8	Wed	Sat	Tue	Thu	Sun	Tue
9	Thu	Sun	Wed	Fri	Mon	Wed
10	Fri	Mon	Thu	Sat	Tue	Thu
11	Sat	Tue	Fri	Sun	Wed	Fri
12	Sun	Wed	Sat	Mon	Thu	Sat
13	Mon	Thu	Sun	Tue	Fri	Sun
14	Tue	Fri	Mon	Wed	Sat	Mon
15	Wed	Sat	Tue	Thu	Sun	Tue
16	Thu	Sun	Wed	Fri	Mon	Wed
17	Fri	Mon	Thu	Sat	Tue	Thu
18	Sat	Tue	Fri	Sun	Wed	Fri
19	Sun	Wed	Sat	Mon	Thu	Sat
20	Mon	Thu	Sun	Tue	Fri	Sun
21	Tue	Fri	Mon	Wed	Sat	Mon
22	Wed	Sat	Tue	Thu	Sun	Tue
23	Thu	Sun	Wed	Fri	Mon	Wed
24	Fri	Mon	Thu	Sat	Tue	Thu
25	Sat	Tue	Fri	Sun	Wed	Fri
26	Sun	Wed	Sat	Mon	Thu	Sat
27	Mon	Thu	Sun	Tue	Fri	Sun
28	Tue	Fri	Mon	Wed	Sat	Mon
29	Wed	Sat	Tue	Thu	Sun	Tue
30	Thu	Sun	Wed	Fri	Mon	Wed
31	Fri	Mon		Sat		Thu

D.5 Image task instructions

The participants were asked to perform four variants of the image task, as described in Section 7.1. The instructions for each variant were printed on separate pieces of paper and handed to the user when they were asked to begin the task.

Image task 1

> *This task uses the MacOS Finder, the Fetch FTP client, and the GIFconverter image manipulation program. If you haven't used these programs before, ask me and I'll show you how they work.*

Open the "tasks" folder and start "Fetch." Fetch will open a connection to the "pictures" directory on the server "rose". This directory contains 12 picture files named tile1.jpeg, tile2. jpeg, tile3. jpeg, etc. Each of the pictures is 60 pixels square.

Your task is to change the size of each image to 100 pixels square.

Follow these steps to resize a picture:

1. use Fetch to download it into the "Internet downloads" folder on this computer.
2. use the Finder to Open it.
3. use GIFconverter to Resize it:
4. select the picture by clicking on it,
5. resize the picture with *Size*… from the *Image* menu, and
6. save the picture with *Save* from the *File* menu.
7. use Fetch to put the file back in the "pictures" directory on rose.
8. use the Finder to delete any copies of the file you have left on this computer.

When you finish the task the image files should be stored on the server "rose" in the "pictures" directory, and any files you have downloaded onto this computer should be moved to the trash.

Image task 2

Resize the files in the “pictures” directory on the server “rose” to 100 pixels square as described in Image task 1 without using multiple selection.

Image task 3

Open the “tasks” folder and start “Fetch.” Fetch will open a connection to the “pictures” directory on the server “rose”. This directory contains 62 picture files named tile1.jpeg, tile2. jpeg, tile3. jpeg, and so on. The pictures are stored in the JPEG format.

Your task is to convert every picture in the “pictures” directory from JPEG format to PICT format. The converted files should be named tile1.pict, tile2.pict, tile3.pict, etc.

You do not need to resize the files.

Follow these steps to convert a picture from JPEG to Macintosh PICT format:

1. use Fetch to download it into the “Internet downloads” folder on this computer.
2. use the Finder to Open it
3. use the GIFconverter program to convert it from a JPEG file into a PICT file.
4. use Fetch to put the PICT file into the “pictures” directory on rose.
5. use the Finder to delete any copies of the file you have left on this computer.

When you finish the task the PICT files should be stored on the server “rose” in the “pictures” directory, and any files you have downloaded onto this computer should be moved to the trash.

You may not use multiple selection to perform this task.

You may use Familiar if you want to.

Image task 4

Convert the files in the “pictures” directory on the server “rose” from JPEG format to PICT format as described in Image task 3.

You may use any techniques you think are suitable, including multiple selection and Familiar.

D.6 Post-experiment interview questions

1. Did you consider doing the Save As operation in the last task by hand?
2. Have you ever used a Macro recorder?
3. Did you consider writing a Macro here?
4. What did you think of Familiar?
5. Do you think you personified Familiar?
6. Did you feel you knew what the explanations meant?

E Generating AppleScript code

Familiar makes predictions through a purely interactive interface, and discards the tasks it has learned when they are finished. The user cannot ask Familiar to remember a task—if they want to perform the task again they have to teach it to Familiar again. This problem could be addressed by having Familiar generate an AppleScript program to complete the entire iterative task. The AppleScript language is sufficiently powerful for this purpose: the complexity of a PBD system lies in inferring patterns; extrapolating them is comparatively simple.

Two motivations for providing an AppleScript program are immediately apparent: to store a program for later use, and to edit and extend a program. A program created for later use must be absolutely reliable. Some users simply assume that Familiar's interactive predictions are correct (Section 7.1) and are likely to transfer that faith to any program Familiar generates. A more promising scenario is that the user wants a program to examine and alter because they are unable to complete some task interactively or because they find it easier to have an agent generate part or all of a program than to write and debug it themselves. In this case, the user will see and read the code, so there is less risk in producing it (as opposed to an executable program).

Generated AppleScript has the potential to educate the end user, in accordance with Familiar's fourth design guideline (Section 3.3.4). Novice programmers can use the interactive interface, see code and become accustomed to its effects, then request the generated code and edit it in a conventional programming environment. This progression will transform the end user to a novice programmer.

```
1  For each command in the pattern
2      For each every (direct or named) parameter of the command
3          Print AppleScript to initialise parameter predictions

4  Print AppleScript for stating a repeat loop

5  For each command in the pattern
6      Print AppleScript to check for termination conditions
7      Print AppleScript for command name
8      If the direct parameter is predicted
9          Print AppleScript for generating direct parameter value
10     For each named parameter of the command
11         Print AppleScript for parameter name
12         Print AppleScript for generating parameter value

13 Print AppleScript for ending a repeat loop
```

Figure E.1 The algorithm for generating AppleScript code from Familiar.

E.1 Generation algorithm

Figure E.1 shows an algorithm for generating an AppleScript program to perform an iterative task. The input the algorithm is a cycle prediction, the same information that is used to generate the prediction window interface (Section 4.1). A cycle prediction comprises a set of command predictions, each of which consist of a command type and a set of parameter predictions.

The algorithm works by assembling parameter predictions into commands, executing each of these commands, then updating the predictions. Four blocks of AppleScript are generated. The first initialises each parameter of each command (Figure E.1, lines 1–3). The second starts a repeat loop (line 4). The third executes each command in the cycle and updates their parameter predictors (lines 5–12). The last closes the repeat loop (line 13).

The algorithm depends on adding methods to the pattern analysis schemes that generate AppleScript code describing their predictions. Explanations, accuracy estimates, and AppleScript commands are already generated in this manner.

E.2 A worked example

Figure 3.3 shows the user teaching Familiar to arrange the files in a folder in a straight line. Each iteration of the task in Figure 3.3 has two commands, *select* and *set*. The *select* command has a single parameter, the direct parameter (i.e. *select*)

```
1   -- initialise variables
2       --select parameter of select command
3       tell application "Finder" to get every file in folder "fruit"
4       set select_command_remaining to result
5       -- set parameter of set command requires no initialisation
6       -- to parameter of set command
7       set to_list_element_1 to 16
8       set to_list_element_2 to 29

9   -- repeat the cycle
10  repeat

11      -- select command
12      if length of select_command_remaining is 0 then exit repeat
13      set select_value to first item in select_command_remaining
14      set select_command_remaining to rest of select_command_remaining
15      tell application "Finder" to select select_value
16      -- set command
17      set to_value to {to_list_element_1, to_list_element_2}
18      set to_list_element_1 to to_list_element_1 + 64
19      tell application "Finder" to set position of selection to to_value

20  end repeat
```

Figure E.2 An AppleScript program that automates the arranging files task (Section 3.3.1).

which is predicted by *PAS-set*. The *set* command has two parameters: the direct parameter (i.e. *set)* is predicted by *PAS-constant*, and the *to* parameter is predicted by *PAS-extrapolation*.

Figure E.2 shows the AppleScript code that might be generated for this task. Lines starting with "--" are AppleScript comments inserted for the user's benefit and are not executed (e.g. lines 1, 2, 5, 6). The program is structured in four parts, corresponding to the four blocks of AppleScript output by the algorithm.

The first block contains the initialisation steps (Figure E.2, lines 1–8) that get the set of files to iterate over (lines 2–4), and store the position of the first file in two variables (lines 6–8). In the second, the loop is initiated with the *repeat* command (lines 9–10). The third block is the body of the repeat loop, which constructs and executes each command in the iterative cycle (lines 11–19). Each parameter is built by checking for termination conditions (line 12), calculating the value in this iteration (lines 13 and 17), and updating the necessary variables (lines 14 and 18). The parameter values are combined in a command and sent to the appropriate application (lines 15 and 19). Simple constant predictions, like the direct parameter of the *set* command need no initialisation (line 5) and can be embedded into the appropriate command (line 19). Others, like the *select*

parameter, require more complex initialisation (lines 2–4) and have several expressions in the repeat loop (lines 11–15). The fourth block simply closes the repeat loop with the *end repeat* command (line 20).

E.3 Discussion

A generated program must be as simple as possible if end users are to understand it. Fortunately, Familiar attempts a highly structured set of tasks, and AppleScript is designed to be used and read by inexperienced programmers. The program in Figure E.2 is relatively simple by the standards of an experienced programmer, but could easily baffle a non-programmer.

The algorithm has weaknesses and could lead to erroneous programs. First, the input is a cycle, and does not include initialisation events in the user's demonstration, only the commands making up each iteration; but the initialisation events may be necessary to the task (for example, creating the folder in Figure 3.8a, lines 2–4 and 9–11). Second, it may be important, but not obvious, which event is the first in each cycle. The program in Figure E.2, for example, would not work correctly if the *set* command were before the *select* command, even though this circumstance can be valid in the prediction window (e.g. Figure 3.9e).