CHAPTER

Fifteen

Domain-Independent Programming by Demonstration in Existing Applications

GORDON W. PAYNTER AND IAN H. WITTEN

Department of Computer Science,
University of Waikato

S
R
L

Abstract

This paper describes Familiar, a domain-independent programming by demonstration system for automating iterative tasks in existing, unmodified applications on a popular commercial platform. Familiar is domain-independent in an immediate and practical sense: it requires no domain knowledge from the developer and works immediately with new applications as soon as they are installed. Based on the AppleScript language, the system demonstrates that commercial operating systems are mature enough to support practical, domain-independent programming by demonstration—but only just, for the work exposes many deficiencies.

15.1 Introduction

Our aim is to add programming by demonstration (PBD) to commercial platforms in a way that is domain-independent and works with existing applications. Domain independence, if it can be achieved, is of huge benefit because it eliminates the difficult, time-consuming, and error-prone job of encoding domain knowledge in a satisfactory way, and it eliminates the brittleness that is associated with unexpected interactions between different pieces of domain knowledge. The ability to work with existing application programs is highly desirable for any interface agent because it eliminates the restriction that users can only use selected applications and removes the need to reimplement applications.

This chapter describes Familiar, a PBD package that operates on a popular computer platform, the Apple Macintosh. Familiar uses standard software (e.g., the Apple Finder, Microsoft Excel, and lesser-known applications such as JPEGView, Fetch, and GIFConverter) and communicates through AppleScript, a standard protocol. We have succeeded in achieving true domain independence: if you install a completely new recordable application that Familiar has never before encountered, issue the Begin Recording command, and start interacting with the application, you will reap the benefits of PBD right away. Success will depend on how well the application implements AppleScript, a limitation that we will discuss.

Macro recorders are a rare example of PBD systems that operate on standard computer platforms—both within application packages, such as spreadsheet macro recorders, and on a systemwide basis. Yet they have

serious limitations, limitations that have been recognized since the early days of programming by demonstration. Cypher (1993a) points out that their main failing is that they are too literal: they replay a sequence of actions at the keystroke and mouse-click level, without taking any account of context or attempting any kind of generalization. Most PBD systems aim higher, recording the user's actions at a more abstract level and making explicit attempts to generalize them. However, they have virtually all been demonstrated only in special, nonstandard, often tailor-made software environments.

Familiar builds on the functionality and interaction style of the Eager PBD system (Cypher 1993b). Familiar extends Eager in three important respects: interface, inference, and domain independence. Eager's "anticipation highlighting" technique presupposes domain knowledge, can only display one action at a time, and provides no explanation of the predictions—users are forced to trust the system (and do so reluctantly). Its inference engine does not tolerate mistakes in demonstrations, cannot explain the predictions it makes, and has a smaller set of generalization techniques available. Although it is domain independent in principle, Eager requires changes to the operating system and applications and has been demonstrated with only one application (HyperCard). Familiar overcomes these problems by using the AppleScript language to communicate with the user and other applications, gaining generality at the expense of Eager's polished interface.

Other PBD systems have used AppleScript to monitor the user and control applications, but they do not exploit its domain independence and high-level application knowledge. Lieberman (1998) has based two separate PBD systems on AppleScript that are tailored to specific applications. ScriptAgent uses inference techniques from Lieberman (1993) to write scripts for manipulating files in the Finder, and Tatlin seeks similarities between data in a spreadsheet and a calendar by interrogating them with AppleScript. Yvon, Piernot, and Cot (1995) use AppleScript as a macro recorder without generalization.

This chapter begins by describing interaction with the Familiar system, from the user's perspective. Because of space limitations, details of the implementation are beyond our scope (see Paynter 2000 for full information.) 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 because (historically) even domain-independent systems have been tightly coupled with prototype

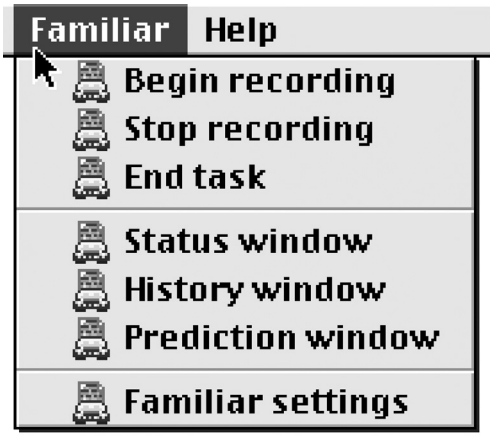
applications. We review the requirements of domain-independent programming by demonstration and finally discuss how AppleScript meets these requirements and what its shortcomings are.

15.2 What Familiar Does

Familiar's purpose is to help the user solve iteration problems in their standard applications and environments. We give three examples of Familiar's use. The first, rearranging a set of files into a horizontal row, 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 is to 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 done so. 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 each example the user first asks the agent to start observing their actions by selecting *"Begin Recording"* from the Familiar menu (Fig. 15.1), which is available in every application. They then proceed to demonstrate the task and interact with the agent. When they have finished working with

FIGURE 15.1



The Familiar menu.

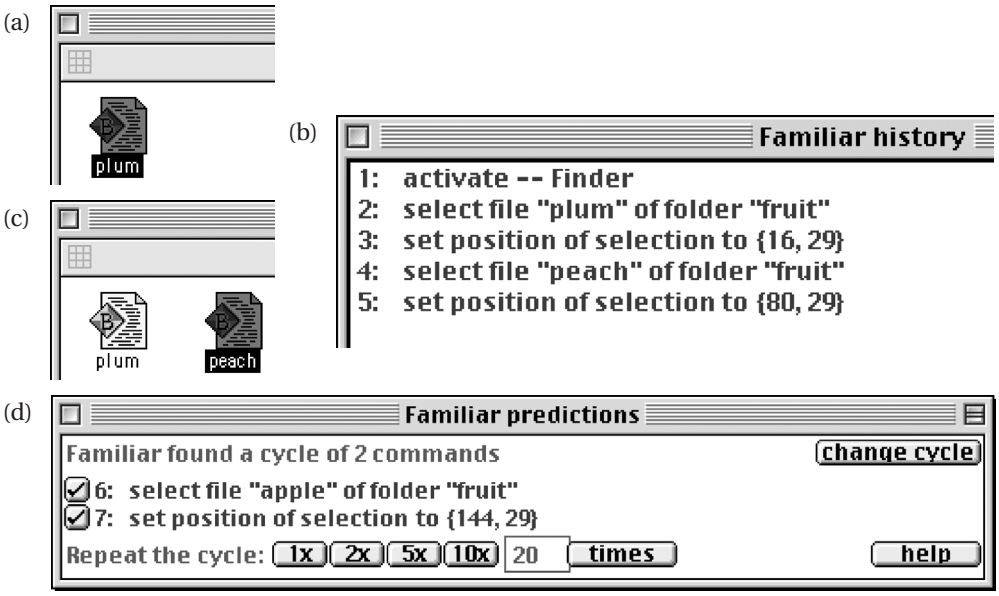
Familiar, they signal that the task is complete by choosing "End task" from the menu. The user can pause a demonstration at any time with the "Stop recording" command and continue it again with "Begin recording."

15.2.1 Arranging Files

To rearrange files into a horizontal row, the user begins by moving a convenient file, "plum," to the top left corner of the folder window (Fig. 15.2[a]). These actions are recorded by Familiar and displayed in the Familiar History window (Fig. 15.2[b]). 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 continues the demonstration by moving the file "peach" (Figure 15.2[c]); again their actions are recorded and displayed (Figure 15.2[b], events 4 and 5).

Each time it records a user action, Familiar attempts to generalize the event trace and infer the user's intent. After event 5 it detects a cycle,

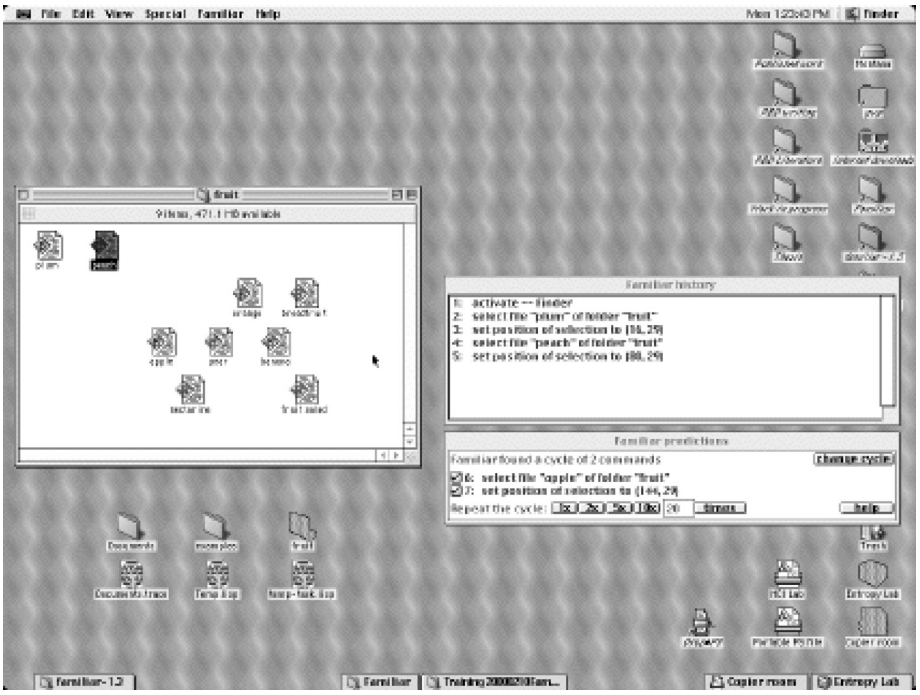
FIGURE 15.2



Using Familiar to arrange files: (a) a demonstration is recorded; (b) the history window; (c) a second demonstration is recorded; (d) a prediction is made.

S
R
L

FIGURE 15.3

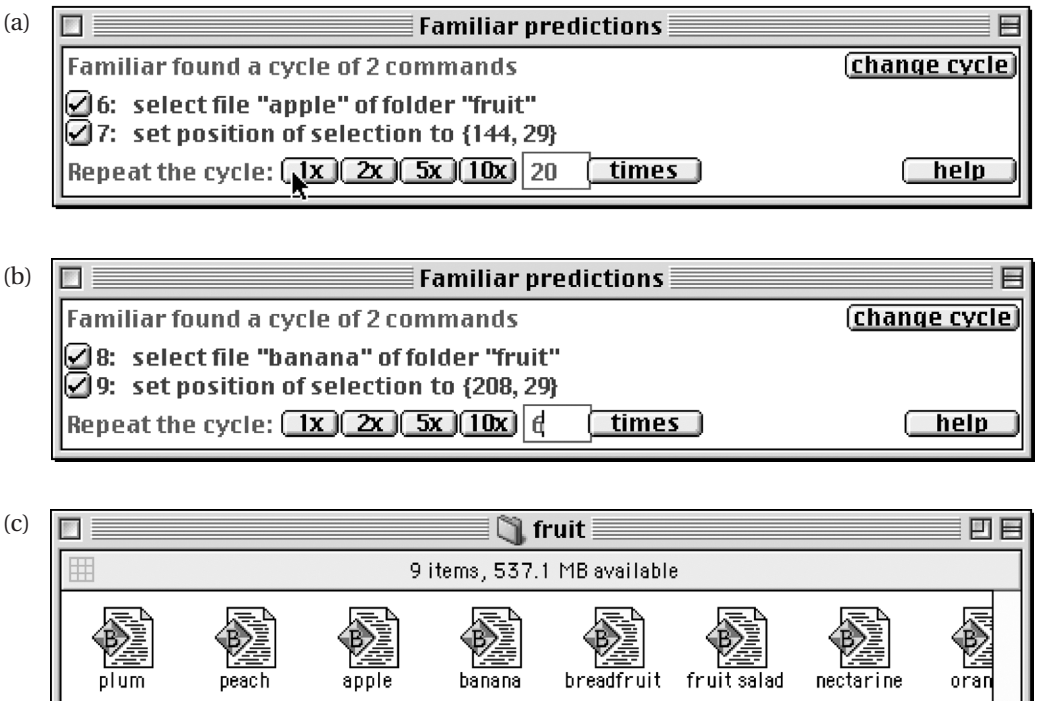


The screen after two demonstrations of the arranging files task.

predicts the next iteration, and presents this prediction to the user in the Familiar Predictions window (Figure 15.2[d]). In this case, Familiar anticipates that the user's next actions will be to "select file 'apple'" (event 6) and set its position (event 7). The user is satisfied with this prediction—the task involves arranging all the files in a row, irrespective of order, and "apple" has yet to be moved.

Figure 15.3 shows the entire screen as it appears after the user has demonstrated the first two examples. The fruit window, in which the user is demonstrating the task, appears on the left. Familiar is a stand-alone application, so its two windows remain in the background (right-hand side) until the user selects one, bringing it to the foreground. They take up only a small part of the total screen area and can be moved to increase visibility. Note that a flashing tape recorder icon has been added to the top left corner of the screen to show that AppleScript recording is active and that the Familiar menu has been added to the standard Finder menubar.

FIGURE **15.4**



Completing an iterative task: (a) a prediction is executed once; (b) the user requests six more iterations; (c) the task is complete.

The Predictions window can be used to perform the task. The "1x," "2x," "5x," and "10x" buttons execute the corresponding number of complete iterations of the cycle (Figure 15.4[a]). The user presses "1x" 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 positions the 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. When the entire iteration has been executed, Familiar displays its prediction for the next iteration (Figure 15.4[b]).

The user can instruct Familiar to execute a given number of iterations by entering the number from the keyboard. In this example the user, knowing how many files are left to position, replaces the default value of 20 (visible

in Figure 15.4[a]) with 6 (Figure 15.4[b]), and presses "times." After each iteration, Familiar pauses to redraw the Predictions window and decrement the number of cycles to go. When six iterations are finished, the task is complete (Figure 15.4[c]).

15.2.2 When Errors Occur

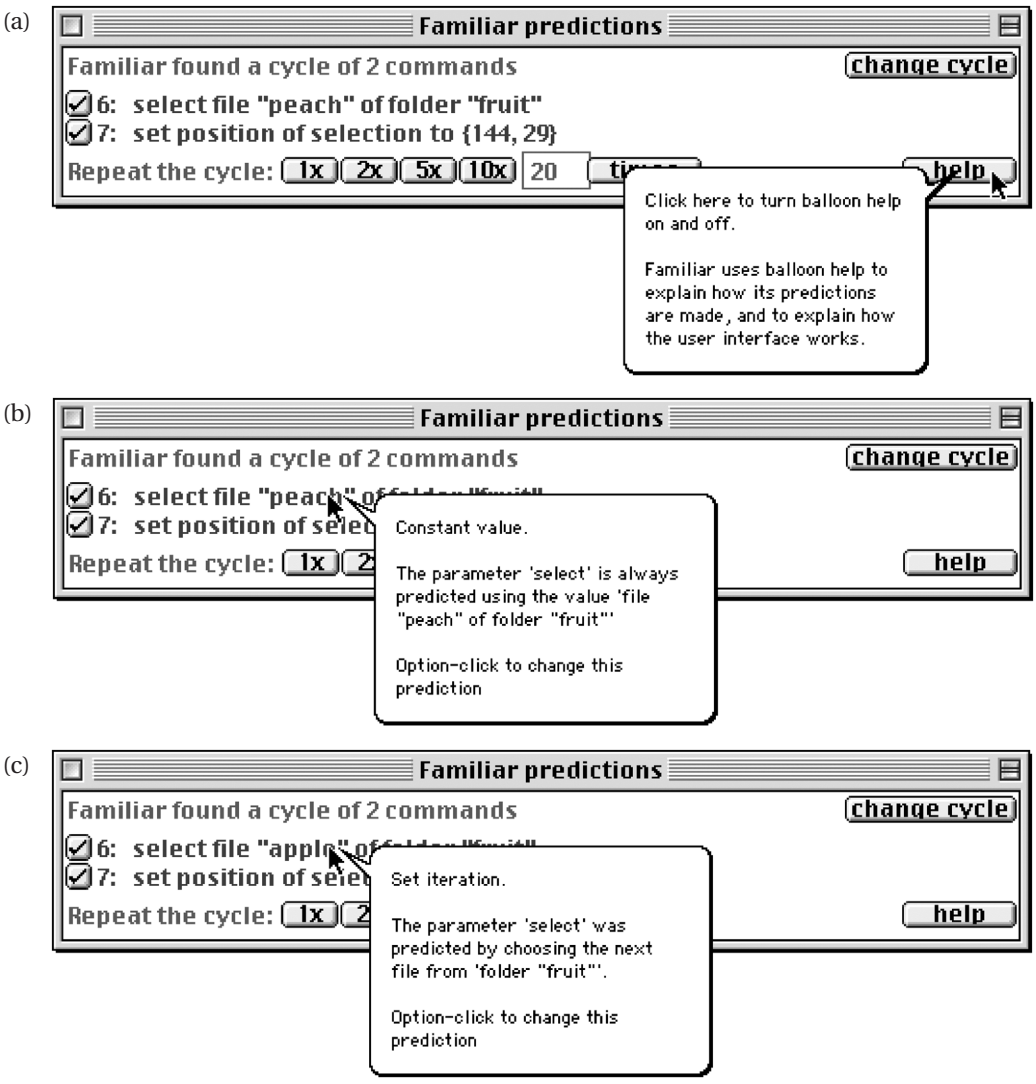
The Predictions window describes Familiar's predictions and accepts feedback about them. The simplest way to correct a mistake is to demonstrate another example in the standard application interface. Familiar will incorporate the new demonstration—and the fact that the old predictions were incorrect—into subsequent predictions. 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 "help" button gives feedback about Familiar's reasoning. The iterative pattern in Figure 15.5(a) is consistent with the two demonstrations of the task (Figure 15.2[a–c]), but the parameter of the "select" command (event 6) has not been extrapolated correctly: Familiar has predicted that the user will select "peach," but the user already moved this file (events 4 and 5) and wants to move a new one.[1] To find out why the agent has made the erroneous prediction, the user clicks on "help." The Macintosh "balloon help" feature is activated (Figure 15.5[a]) and used to explain predictions (Figure 15.5[b–c]). The user is concerned that the "select" parameter is incorrect, and a balloon explains that Familiar has reasoned that the user is positioning the same file in every iteration (Figure 15.5[b]). The prediction can be changed by option-clicking it, whereupon Familiar replaces "peach" with "apple" (Figure 15.5[c]). The new balloon explains that this prediction is generated by assuming that the user is iterating over all the file objects in the folder "fruit." The agent's reasoning—and thus its prediction—is correct, and the task can now be completed.

1. Familiar was artificially constrained to cause this prediction; normally, it would predict correctly.

S
R
L

FIGURE 15.5



Examining and changing an incorrect prediction: (a) balloon help is activated; (b,c) two predictions are explained.

S
R
L

15.2.3 Sorting Files

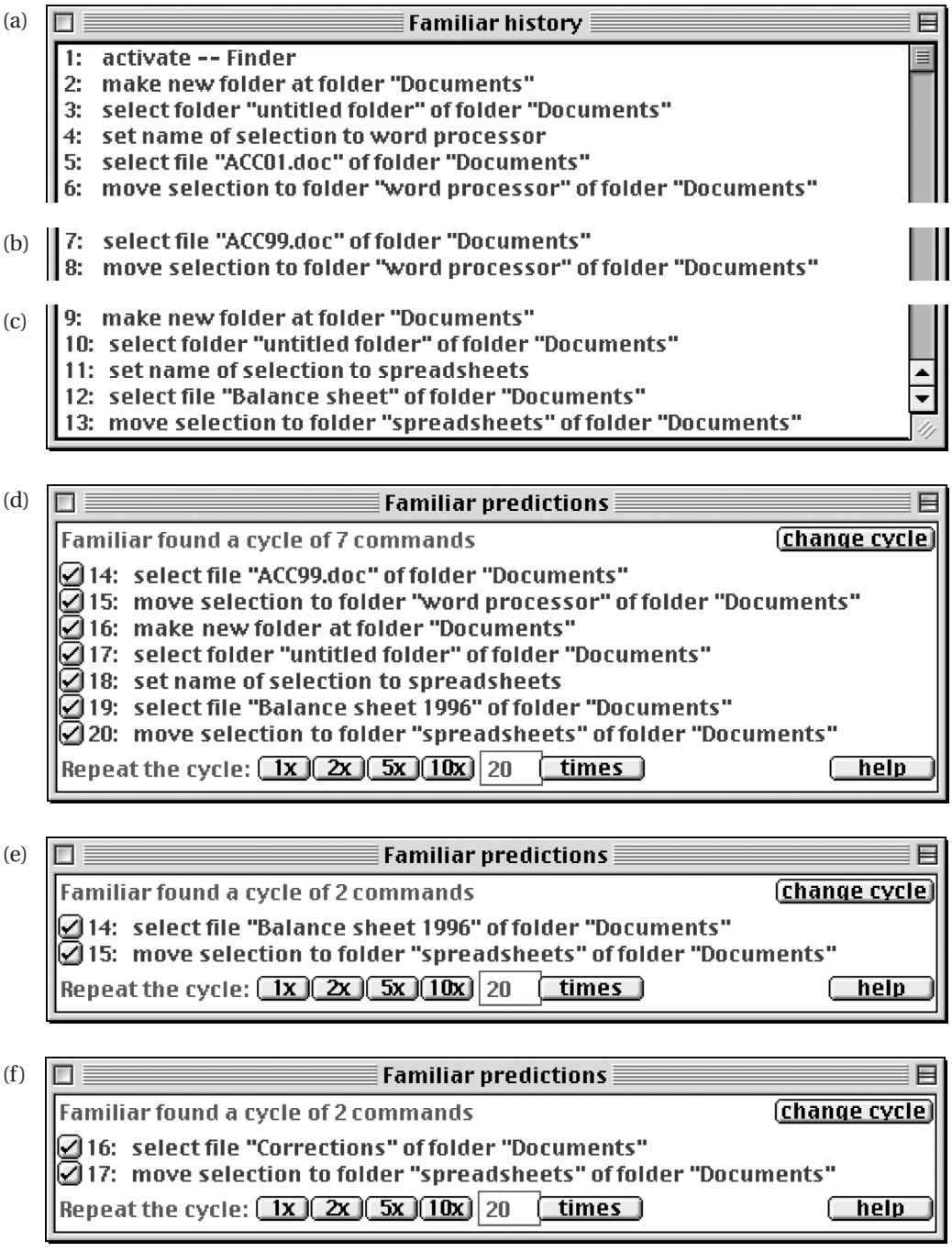
The second task is to sort a set of files into folders for word processor and spreadsheet documents. The user selects "Begin recording" (Figure 15.1) and starts by creating a new folder and renaming it "word processor." These commands are recorded and displayed (Figure 15.6[a], events 1–4) but do not contribute to any iteration—they are once-only initialization steps. The user then moves "ACC01.doc," a word processor document, into the new folder (Figure 15.6[a], events 5–6). To demonstrate the second iteration, the user moves "ACC99.doc" into the word processor folder (Figure 15.6[b], events 7–8). The third file, "Balance sheet," is a spreadsheet, so the user creates a folder called "spreadsheets" (Figure 15.6[c], events 9–11) and moves the file into it (Figure 15.6[c], events 12–13). Three iterations of the task have been demonstrated, but over half of the recorded events are initialization 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 15.6[d]). Unfortunately, it is completely wrong, and the entire cycle it is predicting is incorrect. Each iteration includes creating a new folder and renaming it "spreadsheets"; note that if we permitted Familiar the luxury of domain knowledge, this prediction could be suppressed because it does not make sense to create multiple folders with the same name. 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 15.6[e]). 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 15.6[f]), 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 15.6[f], event 17). Noticing the error, the user clicks on "help" and moves the mouse over the "move" command's "to" parameter. The help balloon (Figure 15.7[a]) explains that Familiar predicts the constant "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 for it to learn this classification task.

The user returns to the Finder and moves "Corrections" into the "word processor" folder. These actions are recorded (Figure 15.7[b]) and used to make a prediction for the next file (Figure 15.7[c]). 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

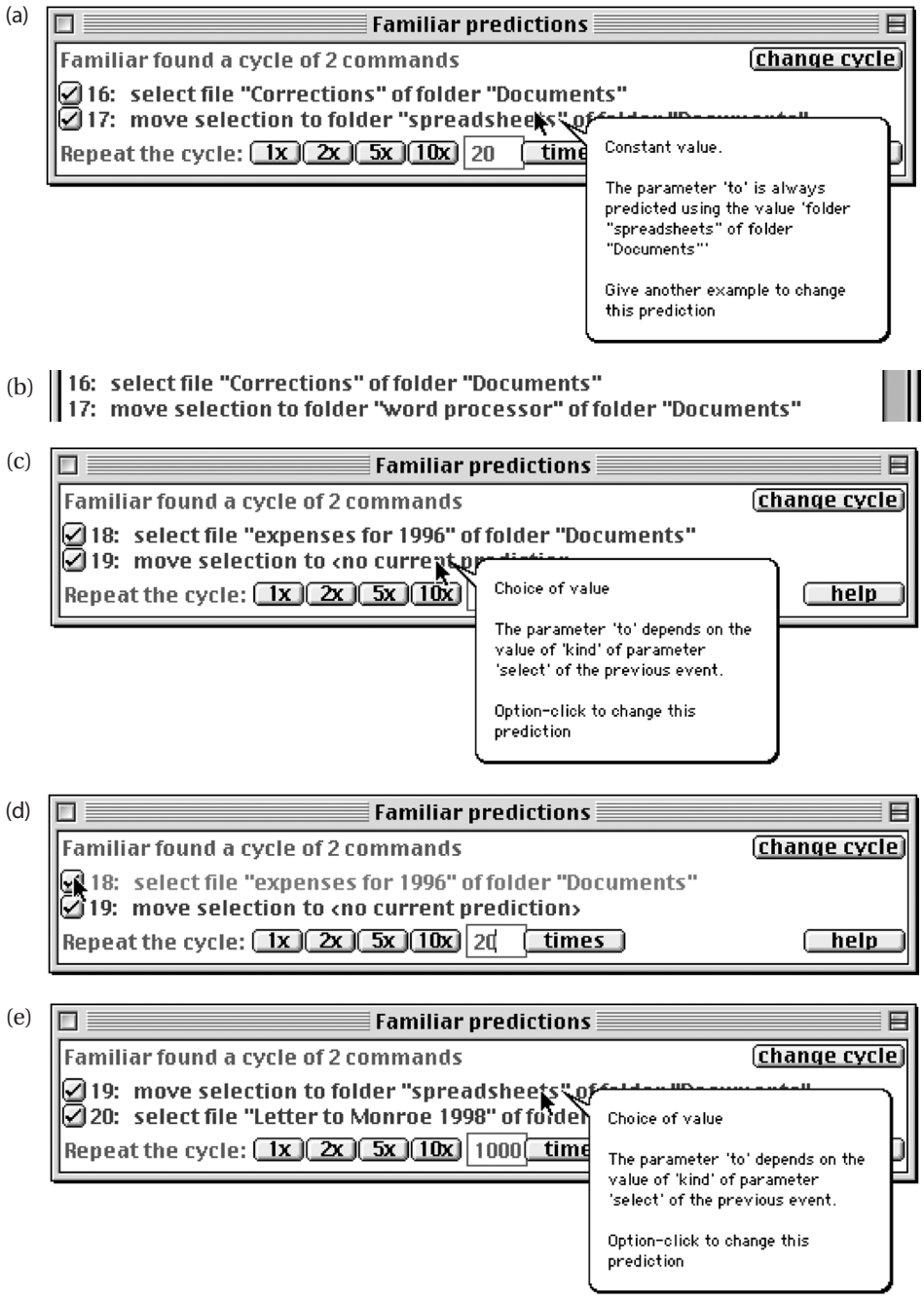
FIGURE 15.6



Changing an incorrect cycle: (a,b,c) three iterations are demonstrated and recorded; (d) the fourth iteration is predicted incorrectly; (e) the fourth iteration is predicted correctly; (f) the fifth iteration is predicted.

S
R
L

FIGURE 15.7



Changing an incorrect parameter: (a) an incorrect prediction is explained; (b) a new demonstration is recorded in the History window; (c) an incomplete, but correct, prediction is explained; (d) the user executes a single command; (e) a correct prediction is explained, and the user requests 1,000 iterations.

S
R
L

giving "no current prediction." The user 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 concrete prediction of event 19 after event 18 has been confirmed. To test the prediction, the user clicks on the "tick" beside event 18. Familiar executes it (Fig. 15.7[d]), adds this event to the History window, and displays its prediction of the next two events (Fig. 15.7[e]).

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 15.7[e]). 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 instructions from the user, who chooses "Stop recording" from the Familiar menu and continues with his or her work.

15.2.4 Converting Images

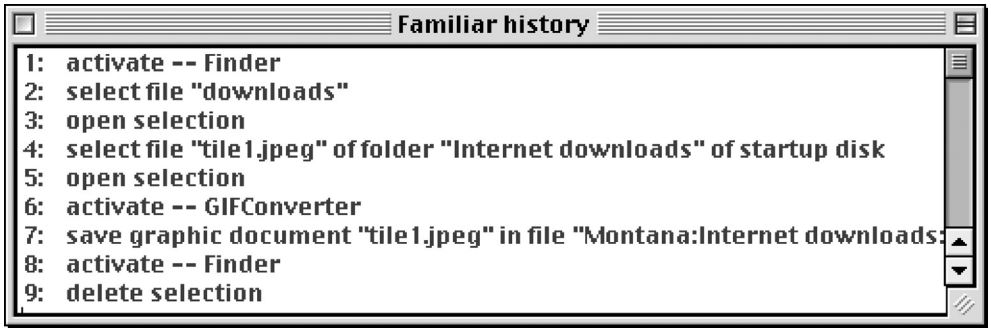
Complex tasks may involve multiple domains, longer demonstrations, and user errors. The History and Prediction windows in Figure 15.8 were generated from a demonstration performed by a subject in a user evaluation. The task (a subtask of a larger task that the subject identified then elected to complete with Familiar) involves two applications and has a noisy event trace.

In this example, Familiar is used to convert a set of graphic files from JPEG format to PICT format. The History window is shown after the first (Figure 15.8[a]) and second (Figure 15.8[b]) iterations have been demonstrated. In each iteration of the task, the user selects a JPEG file in the Finder (events 4, 10), opens it in the GIFConverter image manipulation program (events 5, 11), saves it as a PICT file (events 7, 13), and then deletes the JPEG file (events 9, 16). However, these actions are interspersed with others that are not part of the iterative loop. The first three events of the first iteration initialize the environment. Event 15 is a singular noise event—it was generated when the user shifted a window to get a better view and will never be repeated. In Figure 15.8(c) we see that Familiar has correctly identified a cycle of six significant events and predicted the next full iteration.

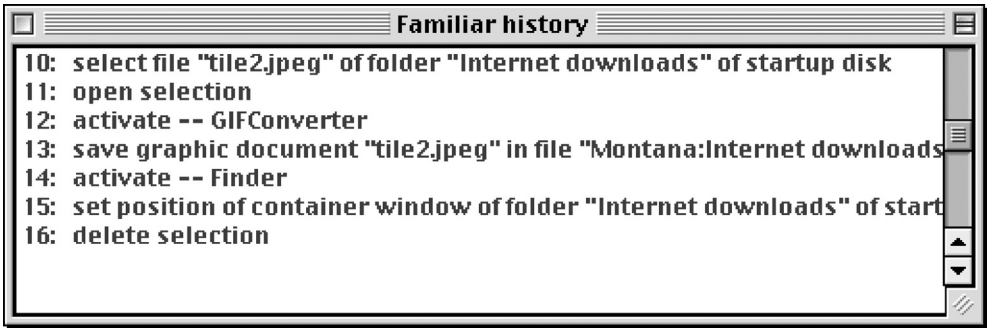
S
R
L

FIGURE 15.8

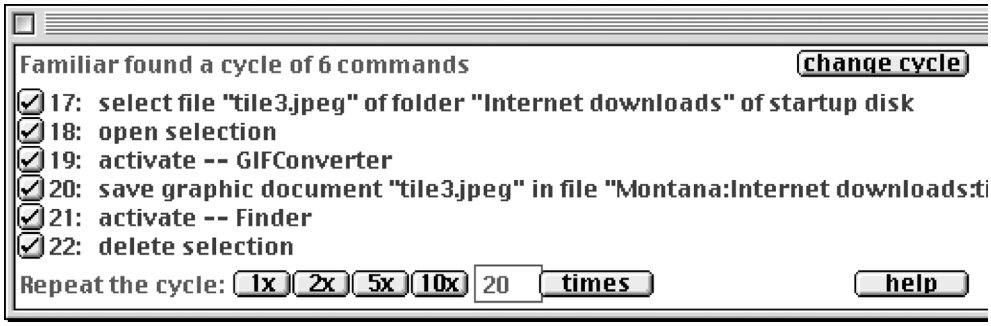
(a)



(b)



(c)



Converting image files: (a,b) two noisy demonstrations are recorded; (c) a correct prediction is made.

S
R
L

15.3 Platform Requirements

Familiar demonstrates that domain-independent PBD can be added to existing applications but is dependent on many services provided by the software architecture. These services are given in Table 15.1, which lists a minimal set of technical and nontechnical platform requirements that must be satisfied before domain-independent PBD is possible in existing applications. Three 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, Charron, and Joy 1999), animated help systems that require complete and exclusive control of the user interface (Bharat and Sukaviriya 1993; Miura and Tanaka 1998), 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 nontechnical: 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 such as Amulet

TABLE **15.1**

Platform requirements of domain-independent programming by demonstration systems.

Requirement	*Description*
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

S
R
L

(Myers and Kosbie 1996) or AIDE (Piernot and Yvon 1993) that is designed to support PBD.

- *Requirement 2: Recordability.* Most kinds of PBD monitoring and recording the user's actions. Each action is recorded, added to the event trace or command history, and analyzed 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 PBD 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* (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 unchanged from one invocation to the next. The latter describes the data the user is working on and the commands he or she has recently. It differs each time the program is run and often changes in response to user actions. PBD systems require access to the applications' 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.
- *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.

S
R
L

15.4 AppleScript: A Commercial Platform

AppleScript is an application-independent, high-level scripting language for the Apple Macintosh (Apple Computer 1993–1999). Familiar uses it to record the user's actions and to examine and control target applications. AppleScript is not the only commercial platform that can support PBD (see Paynter 2000 for alternatives), but it is one of the most sophisticated and is consequently an instructive model for future systems. This section describes AppleScript and its effect on the development of Familiar.

The AppleScript language is composed of control structures (loops, conditionals, statement blocks), common data types (numbers, strings, lists), and interprocess communication. Applications extend the language by adding their own commands; every application makes a dictionary of its commands, object classes, and enumerations available to other programs. Familiar uses the dictionary to model the application, making new, unseen, applications compatible immediately and without human intervention.

AppleScript is an attractive platform because it satisfies the previously noted requirements and provides an English-like feedback language. Applications that respond to AppleScript commands are called *scriptable,* and they satisfy the controllability and examinability requirements. Some scriptable applications also report what the user does; these are called *recordable* applications and meet the recordability requirement. Familiar can only work with applications that are both scriptable and recordable. The simple syntax of instructions such as "open folder 'fruit'" or "select file 'apple'" allow users to comprehend commands they would be unable to formulate themselves, particularly when the command describes a recently performed action. This relieves Familiar of the need to implement a second program representation (e.g., anticipation highlighting or a visual language) to communicate with the user.

The remainder of this section describes the weaknesses of AppleScript as a platform for PBD. These can be divided into those that are caused by its high-level event architecture, those that are problems with the language itself, and those that result from poor implementations of the AppleScript specifications.

15.5.1 High-Level Event Architectures

The high-level events employed by AppleScript characterize user actions in an abstract form that omits details of how each operation is performed. In

contrast, low-level events correspond directly to specific machine input, describing the user's physical actions rather than their effects. Although high-level events are easier to interpret and search for repetition, they do have drawbacks:

- *Data description:* Data description problems arise because application objects are identified by a description, not accessed directly (e.g., by pointers to memory). If an object changes so that it no longer matches its prior description, then it can no longer be accessed by an external system. Furthermore, 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.
- *Mismatch between interaction and 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 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.

15.5.2 Deficiencies of the Language

AppleScript was designed 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.

- *Single-user assumption:* Applications treat commands from agents as though they were user actions, which creates contention when agents activate applications, make selections, or use the clipboard at the same time as the user, because such operations involve global variables—the *frontmost* application, the *selection,* the *clipboard.*

- *Examinability:* Familiar regularly accesses data in other applications, but it is hampered by its inability to traverse object hierarchies at run time and find all the properties of specific objects. The "get every" command can be used to examine hierarchies, but it is difficult to tell whether a given application supports it. Inheritance information is optional and often omitted, preventing an agent from finding all the information relevant to an object. These are arguably implementation problems, but they are systemic because AppleScript lacks standards of compliance.
- *Speed:* AppleScript is notoriously slow, although this drawback has been alleviated by recent versions and machines. 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 without interfering with interaction, although it may offer predictions too late and retrieve data that are stale.
- *Timing:* High-level events are reported retrospectively, and agents have no access to data the event has overwritten. 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 that the user first selects an object (noun) and then applies some action (verb), a style that is reflected in "select-set" cycles (e.g., Fig. 15.2[d]). 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. Cypher (1993b) describes an unreleased version of AppleScript that lets the agent examine the application before and after each event, but this is incompatible with current software and 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.
- *Representing spatial relations:* Textual languages are often inadequate for describing graphical data, one of several shortcomings of English-like programming languages (Thimbleby, Cockburn, and Jones, 1992). Given application knowledge, it is possible to represent information graphically, but this sacrifices domain independence and consistency.
- *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, the Scriptable Text Editor identifies documents with index numbers, 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 it 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 inconsistently and cannot be relied on. As a result, agents are unable to undo their actions.

15.5.3 Deficiencies of 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:* Syntactic items are often chosen confusingly. The Microsoft Excel (version 5, 98) command to enter the formula =SUM(A1:A50) in cell B1 is

```
Select Range "R1C2"
set FormulaR1C1 of ActiveCell to "=SUM(RC[-1]:R[49]C[-
  1])"
```

This command 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 what `FormulaR1C1` means. 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 is simpler and more closely reflects the user's actions, but it is not used.

- *Recordings not matching actions:* The single largest problem with AppleScript recording is that the recording does not always reflect the actions the user has performed. This drawback confuses agents that rely

on AppleScript recording to monitor the user's actions, and it 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. For example, the formatting commands in Microsoft Excel (version 5, 98) misrepresent the user's actions by adding commands, while Netscape Navigator (versions 3–4.6) does not record the user clicking on a hyperlink.

- *Application behavior changing during recording:* Some applications behave differently when recording than they do normally, which impairs the user's ability to demonstrate. For example, Microsoft Word (version 6, 98) disables the mouse when recording is turned on.
- *Incompletely specified objects:* Objects are often described incorrectly in the application dictionary—particularly with regard to object inheritance. In the Finder (versions 7.5–8.1), 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 such omissions.
- *Errors (often fatal):* Many AppleScript implementations contain serious bugs. For example, the "set" command is missing from the Fetch (version 3.0.3) dictionary; the "resize" command recorded in GIFConverter (version 2.4d18) hangs the machine when replayed.
- *Lack of recordable applications:* Finally, there is a shortage of scriptable and recordable applications. Adding these features to an application incurs extra expense, so not all applications are scriptable. Fewer still are recordable—a prerequisite for Familiar.

15.5.4 Learning From AppleScript's Shortcomings

The shortcomings of AppleScript—which, nevertheless, is a usable platform for PBD—provide insight into the design of scripting architectures and scriptable applications. The three most important steps that application developers can take to make their scriptable applications cooperate with PBD systems and other agents are to design well for human users, to implement recording, and to use sensible data descriptions. Simone's (1995) "human scriptability guidelines" explain how to design the scripting implementation well for users. Syntax that is good for a user is good for an agent, because agents aim to understand and emulate the user.

It is important that AppleScript recording is implemented so that the agent can monitor the user. The recorded actions should describe the user's actions as closely as possible: every significant user action should be included, with no "extra" commands. Unfortunately, it is difficult to judge what might be a significant action, and 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. Recorded commands should both read well and parse easily, so simple terms are essential. Recordings should not include internal information, and the application's behavior must not change when recording is activated.

Many AppleScript implementations suffer from data description problems, preventing an agent from examining their data. Agents have no intuition, so it is important that the dictionary specifies every class completely, without omitting inheritance relationships. Persistent references are essential, because references that become stale or expire are of no use to an agent. Every property of an object should be accessible through the "get" command (including inherited properties). If a requested property does not exist, then return a null object, not an error—errors imply a mistake by the user or agent. The "get every" syntax should be universally supported, and every containee object of a class should share a superclass, so that it is possible to retrieve the contents of an object with a single command.

15.6 Conclusions

End users can be loosely defined as nonprogrammers who are skilled computer operators. They form the largest group who stand to benefit from programming by demonstration, but they are unlikely to give up the environments and applications they know for new, possibly 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 demonstrational tools. Instead, PBD must be added to existing applications if it is ever to be successfully used—or even evaluated—by typical users.

Familiar demonstrates that commercial operating systems are mature enough to support practical, domain-independent PBD—but only just. As more architectures support PBD's the platform requirements—users and applications, recordability, controllability, examinability, user interface, and

consistency—we anticipate that reliable, intelligent, domain-independent PBD systems such as Familiar will supersede the humble macro recorder.

Programming by demonstration systems are limited by the environment in which they operate, which explains why they are invariably implemented on research platforms. AppleScript is a (barely) adequate platform; it suffers from a number of deficiencies. Most stem from the fact that it was designed for end users, not for agents. We urge future designers of scripting languages to treat agents as first-class citizens.

References

Apple Computer Inc. 1992. *Macintosh human interface guidelines.* Reading, Mass.: Addison-Wesley.

———. 1993–1999. *AppleScript language guide:* English Dialect. Apple Computer Inc., Cupertino, CA.

Bharat K., and P. Sukaviriya. 1993. Animating user interfaces using animation servers. In *Proceedings of UIST '93.* Atlanta, GA: ACM Press, New York, NY.

Cheikes B. A., M. Geier, R. Hyland, F. Linton, L. Rodi, and H. Schaefer. 1998. Embedded training for complex information systems. In *Proceedings of Intelligent Tutoring Systems,* Berlin: Springer.

Cypher, A. 1993a. Bringing programming to end users. Introduction to *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

———. 1993b. Eager: Programming repetitive tasks by demonstration. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Halbert D. 1993. SmallStar: Programming by Demonstration in the Desktop Metaphor. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Kay J., and R. C. Thomas. 1995. Studying long-term system use. *Communications of the ACM* 38, no. 7: (July): 61–69.

Kosbie, D., and B. Myers. 1993. A system-wide macro facility based on aggregate events: A proposal." In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Lieberman, H. 1993. Mondrian: A teachable graphical editor. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

———. 1998. Integrating user interface agents with conventional applications. In *Proceedings of the International Conference on Intelligent User Interfaces,* (San Francisco, January). New York, NY: ACM Press.

S
R
L

Linton F., A. Charron, and D. Joy. 1998. OWL: A recommender system for organisation-wide learning. Technical report, The MITRE Corporation.

Miura, M., and J. Tanaka. 1998. A framework for event-driven demonstration based on the Java toolkit. In *Proceedings of the Asia Pacific computer human interaction conference.* Kanagawa, Japan: IEEE Computing Society, Los Alamitos, CA.

Myers, B. A., and D. S. Kosbie. 1996. Reusable hierarchical command objects. In *Proceedings of CHI '96,* Vancover, Canada: ACM Press, New York, N.Y.

Olsen, D. R., Jr., S. E. Hudson, T. Verratti, J. M. Heiner, and M. Phelps. 1999. Implementing interface attachments based on surface representations. *Proceedings of CHI '99.* Pittsburgh, PA: ACM Press, New York, N.Y.

Paynter, G. 2000. Automating iterative tasks with programming by demonstration. Ph.D. diss. University of Waikato, New Zealand.

Piernot, P. P., and M. P. Yvon. 1993. The AIDE project: An application-independent demonstrational environment. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Ritter, S., and K. R. Koedinger. 1995. Towards lightweight tutoring agents. In *Proceedings of the World Conference on Artificial Intelligence in Education (AI-ED '95),* (Washington, D.C., August). (Cited in Cheikes et al. 1998.)

Simone, C. 1995. Designing a scripting implementation. *Develop* 21 (March): 48–72.

Thimbleby, H., A. Cockburn, and S. Jones. 1992. HyperCard: An object-oriented disappointment. In *Building interactive systems: Architectures and tools,* ed. P. D. Gray and R. Took. Berlin: Springer.

Yvon, M., P. Piernot, and N. Cot. 1995. Programming by demonstration: Detect repetitive tasks in telecom services. In *Proceedings OZCHI '95.* Wollongong, Australia.

S
R
L