Fourteen

Learning Users' Habits to Automate Repetitive Tasks

JEAN-DAVID RUVINI AND CHRISTOPHE DONY

LIRMM, University of Montpellier

S

R

L

Abstract

Adaptive Programming Environment (APE), a software assistant embedded into the VisualWorks Smalltalk interactive programming environment, watches what the user is doing, draws on machine learning to learn the user's habits, and afterward offers to complete repetitive tasks on his or her behalf. The goal of the APE project was threefold: (1) to design an assistant able to automate repetitive tasks with a minimal amount of user's intervention, (2) to design an assistant able, as in programming-by-example (also called programming-by-demonstration) systems, to replay and automate complex repetitive tasks, and (3) to design an assistant that disrupts the user's work as little as possible—that is, that makes the right suggestion at the right moment. As a consequence, APE employs a machine-learning algorithm we have specifically designed to learn efficiently and rapidly not only what to suggest to the user but also when to make a suggestion.

14.1 Introduction

Entering repetitive sequences of commands (or repetitive tasks) is a well-known characteristic of human-computer interaction. To deal with this problem, early works have associated macro or script languages with interactive environments—for example, macros in Excel or Lisp scripts in Emacs. They allow the user to write a program that can be invoked later to perform a sequence of commands automatically. The limitation of this approach is that, generally, users do not want to or cannot spend too much effort on programming: writing a program often takes longer than performing a sequence of commands manually disrupts the user's work flow and requires programming knowledge that many users do not have. Recent advances to overcome these limitations came from different correlated fields of research: programming by demonstration (PBD), predictive interfaces and learning interface agents.

PBD systems (Cypher et al. 1993) let the user demonstrate what the task to be automated should do and then create a program from this demonstration. Macro recorders were the first examples of PBD systems, but they were limited because recorded commands are too specific (rote learning, no parameterization) to be reused. Sophisticated PBD systems, such as Mondrian (Lieberman 1993), create programs containing variables, iterative loops, or conditional branches from observing user's actions. Although PBD

S
R
L

does not require programming knowledge because the user does not have to write code, demonstrating a program takes time and disrupts the user's work flow.

Predictive interfaces (Darragh and Witten 1991) and learning interface agents (Maes 1994) observe the user while he manipulates the environment. They try to learn from the correlations between situations the user has encountered and the corresponding commands he has performed, and to predict after each new command what the next one will be. They assist him by afterward predicting and suggesting some commands to perform automatically. For instance, CAP (Mitchell et al. 1994), an assistant for managing meeting calendars, suggests default values regarding meeting duration, location, time, and day of week. OpenSesame! (Caglayan et al. 1997) runs in the background on Macintosh system 7 and offers to open or close files or applications, to empty trash, or to rebuild the desktop on the user's behalf. WebWatcher (Armstrong et al. 1995), an assistant for the World Wide Web, suggests links of interest to the user. Maes's (1994) assistants for handling electronic mail, scheduling meetings, and filtering electronic news advise the user for some application-specific operations such as managing mail, scheduling meetings, or selecting articles in news.

ClipBoard (Motoda 1997), an interface for Unix, tries to predict the next command the user is going to issue. The main advantages of these systems is that they do not require programming knowledge, nor do they disrupt the user's work flow because commands are automatically suggested. However, they do not create programs and thus only suggest single actions and not sequences of actions. Furthermore, the set of actions that most of these systems (expect ClipBoard and WebWatcher) can suggest is small and known in advance.

Eager (Cypher et al. 1993) is one of the most famous attempts to bring together PBD and predictive interfaces. Eager is an assistant for Macintosh Hypercard. When Eager detects two consecutive occurrences of a repetitive task in the sequence of a user's actions, it assumes they are the first two iterations of a loop and proposes to complete the loop. It is a PBD system because it is able to infer loops from observing a user's actions and to replay more than one action at once; it is a predictive interface because it is able to make suggestions without any user intervention. It is able to perform loop iterations until "a condition" is satisfied or following some typical patterns, such as days of the week or linear sequence of integers. Finally, Eager has an important characteristic: it makes a suggestion only after two consecutive occurrences of a repetitive task. As a consequence, it knows exactly when to make a suggestion and which suggestion to make. However, this characteristic is a limitation because in practice such occurrences are frequently not

S
R
L

consecutive but interleaved with other actions. Familiar (see chap. 15) takes on Eager's idea and extends it in many ways but does not address this limitation.

The goal of our work has been to design an assistant operating in a context where the number of possible user actions and possible values for the parameters of these actions are large, where repetitive sequences are not known in advance and not consecutive, and where these sequences are able to predict and replay repetitions composed of several actions, containing loops or conditional branches. None of the previously noted works addresses all these issues simultaneously. In such a context, a key issue is to design an assistant that makes "the right suggestion at the right moment"; an assistant that constantly bothers the user with a lot of wrong suggestions is useless because the user would rapidly ignore it. Wolber and Myers's chapter (Chapter 16) suggests a solution to this problem in the context of PBD system. It proposes to allow the user to demonstrate "when" to make a suggestion as well as "what" to suggest. APE takes another approach. It employs machine-learning techniques to learn efficiently and rapidly when to make a suggestion and which sequence of actions to suggest to the user.

As a case study, we present the APE (Adaptive Programming Environment) project. APE is a software assistant integrated into the Visualworks Smalltalk programming environment. Like Eager and Familiar, APE is able to detect loops and to suggest repetitive tasks iteratively.[1]

In the following sections we describe APE, demonstrating what it does and how it can be used. We explain what kind of repetitive tasks it is able to automate and how it automates them. We show what makes learning users' habits difficult, and we describe what and how APE learns. We compare experimental results of alternate approaches. Finally, we summarize lessons learned from this study and give perspectives for future research.

14.2 Overview of APE

APE is made of three software agents—an Observer, an Apprentice, and an Assistant—working simultaneously in the background without any user intervention. Table 14.1 defines our terminology, and Figure 14.1 describes the role of each agent.
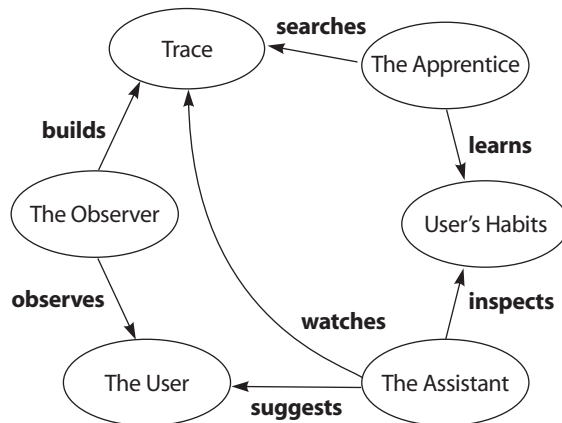
---

1. APE is written in Visualworks Smalltalk 3.0—ObjectShare, Inc., operational and publicly available at www.lirmm.fr/~ruvini/ape.

TABLE **14.1**

Definition of terms used throughout this chapter.

Term	Definition
Action	A high-level intervention of the user on the environment (as opposed to low-level interventions such as mouse movements and keystrokes), window manipulation, menu item selection, button pressing, text entering, etc. An action is parameterized by, among other things, the tool (e.g., *Browser, Debugger, Text Editor*) in which it has been performed.
Trace	A history of the user's actions
Task	A sequence of actions of the trace
Repetitive task	A task occurring several times in the trace
Situation	A sequence of actions of the trace of a given size n, n being a parameter of the learning algorithms
Current situation	The last n actions of the trace
Situation pattern	A regular expression matching one or more situations
Habit	A pair of "set of situation patterns—repetitive task" such that the situation patterns match the situations in which the user performs the repetitive task.
When-set	A set of situation patterns that match the situations in which the user has performed repetitive tasks
What-set	A set of habits

FIGURE **14.1**

14.2.1 The Observer

The Observer traps a user's actions, reifies them into dedicated Smalltalk objects (instances of classes shown in Figure 14.2), and stores them in the trace. It then sends messages in the background to the Apprentice and the Assistant to notify them that the user has performed a new action.

For example, when the user selects the "doIt" command of a text editor to evaluate an expression, an instance of the class ActionEditor is created and references to the involved text editor, the evaluated text, and the string "doIt" are stored, respectively, in the toolID, text, and action slots. Figure 14.3 shows an example of a part of a trace where each line is a simplified textual representation of an action (for clarity, we only show the most informative action parameters). Different classes represent the actions held in the different tools of the environment (browser, debugger) because they hold different versions of methods used by the learning algorithm, which does not handle all kinds of actions equally.

In this first implementation, trapping the user's actions has been achieved by directly modifying methods (up to 170) of the user interface

FIGURE **14.2**

```
Object
  Action (type toolName toolID date display)
    ActionApplication (action)
      ActionBrowser (parameter textMode
        selected)
        ActionDebugger ( )
        ActionFileBrowser ( )
        ActionParcelBrowser ( )
      ActionChangeList (index plug)
      ActionEditor (text index which)
      ActionInspector (parameter on)
      ActionLauncher ( )
      ActionParcelList ( )
      ActionWindow ( )
    ActionError (error object message)
```

Smalltalk hierarchy of action classes.

S

R

L

FIGURE **14.3**

```
ActionEditor(anEditor,'anArray
  stupidMesage','doIt')
ActionError('doesNotUnderstand','stupidMessa
  ge')
ActionDebugger(aDebugger,debug)
ActionWindow(aDebugger,'move')
ActionWindow(aDebugger,'resize')
```

A sample of the trace where the user opens, moves and resizes a debugger to correct an error.

layer in which the user's actions are fired. This result is not very satisfactory and should be improved in the future versions. It is a consequence of the lack of a standard mechanisms, such as the "advice/trace" mechanisms of Interlisp (Teitelman 1978) or "wrappers" mechanisms, of Flavors (Moon 1986), in the Smalltalk environment we have used. Such mechanisms have been developed for Smalltalk (see, e.g., (Böcker and Herczeg 1990), but none have been integrated in the Smalltalk environment we have used.
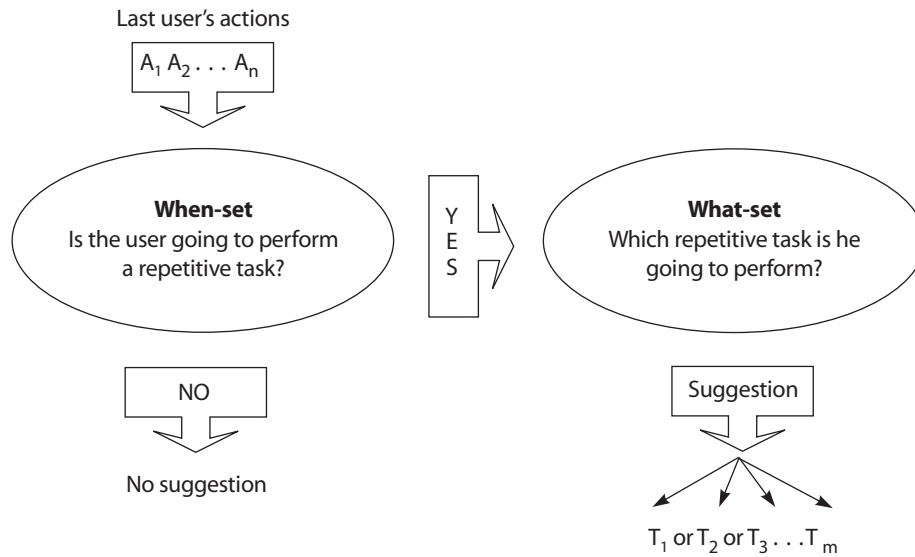
14.2.2 The Apprentice

The Apprentice activity is twofold: (1) It detects the user's repetitive tasks, and (2)it examines the situations in which repetitive tasks have been performed and uses two machine-learning algorithms to learn situation patterns and build two sets:

• the When-set of situation patterns matching the situations in which the user has performed the detected repetitive tasks, and

• the What-set of the user's habits (i.e., pairs "set of situation patterns—repetitive task," where the set of situation patterns reflects all the situations in which a given repetitive task has been performed).

The Apprentice is able to learn two kinds of situation patterns: situation patterns containing wild cards (i.e., a special character—noted "."—that

S

R

L

FIGURE **14.4**

Last user's actions

$A_1 A_2 \ldots A_n$

When-set
Is the user going to perform
a repetitive task?

Y
E
S

What-set
Which repetitive task is he
going to perform?

NO

Suggestion

No suggestion

T_1 or T_2 or $T_3 \ldots T_m$

The Assistant inspects the When-set and the What-set to make suggestions.

matches any single action or action parameter) and unordered situation patterns (the order in which some actions are performed does not matter) containing wild cards. The number of wild cards is not limited.

An occurrence of a situation pattern containing a wild card is learned when, for example, the user has examined several methods in a Smalltalk browser, named "=" in the testing protocol, for various classes of the MyGraphics category (see Figure 14.10 on page XXX). The detected repetitive task is "select protocol(testing), select method(=)," and the learned situation pattern is "select category(MyGraphics), select class(.)."

14.2.3   The Assistant

Observing the user, the Assistant uses the When-set to determine when to make a suggestion to the user; if it has to, it uses the What-set to determine what to suggest. More precisely, as shown in Figure 14.4, after each user's action, it inspects the What-set to answer the question "Is the user going to perform a repetitive task?" If the last user's actions match none of the situation patterns of the When-set, the answer is "no," and the Assistant makes

S
R
L

no suggestion. Otherwise, the answer is "yes," and the Assistant inspects the user's habits (What-set) to answer the question "Which repetitive task is the user going to perform?" It selects all habits[2] with a situation pattern that matches the current situation. Then, it displays in the Assistant window (see Figure 14.5), without interrupting the user's work, the actions composing the repetitive task of the selected habits. The user can ignore this window and these suggestions (nonobtrusive behavior) or mouse-click on one of them. In the latter case, the Assistant successively performs the actions and removes the suggestions from its window.

14.3 Illustrative Examples

This section provides four examples of what APE is able to learn and suggest.

14.3.1 Example 1

Repetitive tasks frequently appear while testing applications. Consider a user testing a multiprocess simulation of the classical "n-queens" problem, implemented by a main-class Board. Figure 14.5 shows two VisualWorks snapshots including both an Assistant window, labeled "Assistant," and the main APE window, labeled "Ape Agents." The Watch button shows that the three agents are active. In the top snapshot, labeled "situation before firing a habit," the user has selected, in a simple editor (named Workspace), a Smalltalk expression to create a board and to initiate the computation and is about to select the InspectIt item of that editor menu. Because the user is not performing this activity for the first time, a repetitive sequence has been detected and a habit has been learned, a situation pattern of which matches the current situation. The Assistant thus fires the habit—that is, displays in its window a text describing the proposed repetitive sequence of actions (opening four inspectors in cascade to show a particular field of a composed object). This repetitive task being exactly what the user intends to do, he mouse-clicks on that text to perform the sequence of actions leading to what is shown in the bottom snapshot labeled "situation after a habit has
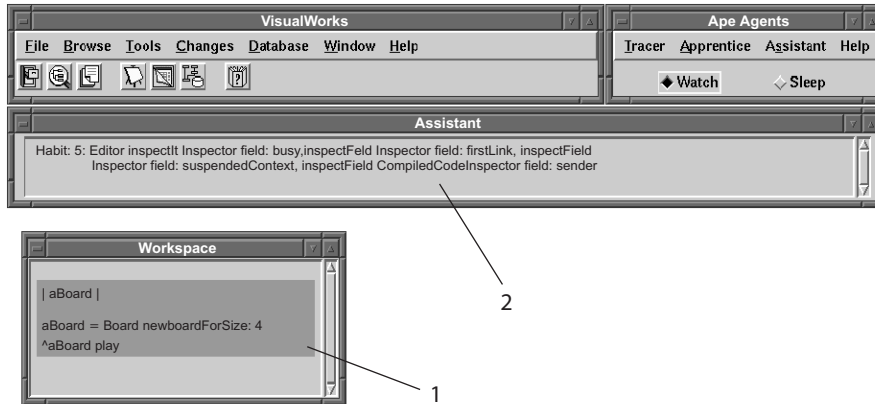
---

2. Browsing through a huge number of suggestions to find the right one puts a workload on the user. The number of suggestions the Assistant can make is a parameter of APE.
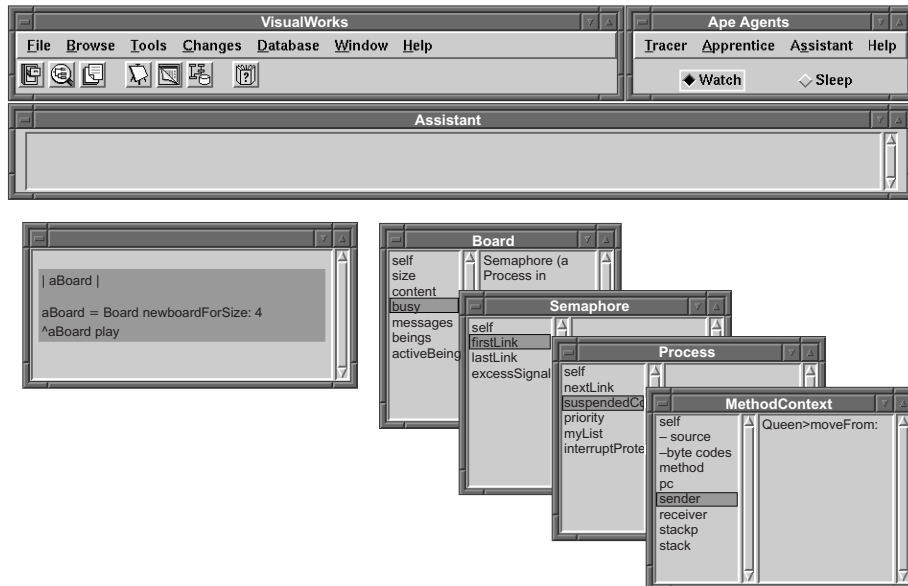
FIGURE 14.5

Situation before firing a habit

The Assistant suggests to open four inspectors...

VisualWorks

File Browse Tools Changes Database Window Help

Ape Agents

Tracer Apprentice Assistant Help

◆ Watch   ◇ Sleep

Assistant

Habit: 5: Editor inspectIt Inspector field: busy,inspectFeld Inspector field: firstLink, inspectField
Inspector field: suspendedContext, inspectField CompiledCodeInspector field: sender

2

Workspace

| aBoard |

aBoard = Board newboardForSize: 4
^aBoard play

1

Situation after a habit has been fired

The user mouse-clicked on the proposition

VisualWorks

File Browse Tools Changes Database Window Help

Ape Agents

Tracer Apprentice Assistant Help

◆ Watch   ◇ Sleep

Assistant

| aBoard |

aBoard = Board newboardForSize: 4
^aBoard play

Board

self
size
content
busy
messages
beings
activeBeing

Semaphore (a
Process in

Semaphore

self
firstLink
lastLink
excessSignal

Process

self
nextLink
suspendedCo
priority
myList
interruptProte

MethodContext

self
– source
–byte codes
method
pc
sender
receiver
stackp
stack

Queen>moveFrom:

*The user has just (1) selected an expression to create and test an instance of class
"Board." (2) The Assistant detects a known situation and suggests to execute a regis-
tered repetitive sequence of actions, displayed in the Assistant window, that evaluates
the expressions and opens four inspectors in cascade, leading to what is shown in the
bottom snapshot.*

S

R

L

been fired." In this case, the user has performed seven actions in a secure way with a single mouse click.

14.3.2 Example 2

Repetitive tasks also frequently appear while debugging applications. Consider the same user now debugging his "n-queens" application. The user has selected a Smalltalk expression (see Fig. 14.6, top snapshot, arrow 1) in the Workspace window, the evaluation of which (arrow 2) has raised an exception leading to the opening of an Exception window (arrow 3). Because this situation matches a situation pattern learned by the Apprentice, the Assistant offers to perform the related repetitive task: "open, move, resize a debugger and select stack index 5." This repetitive task is exactly what the user intends to do, and he mouse-clicks on the proposition (arrow 4), entailing the creation and correct positioning of a debugger window, as shown in the bottom snapshot.

14.3.3 Example 3

This examples shows that APE is able to automate sequences of actions iteratively (in a loop), even if the iterations are not consecutive. Suppose a user intends to modify the method "area" of all the classes belonging to a category named MyGraphics. Before working on a method "area," she wants to save it (back it up). She has first selected and saved the area method of the Circle class by performing the following actions: select the MyGraphics category (Figure 14.7[a], arrow 1), select the Circle class of that category (arrow 2), select the accessing protocol (arrow 3) and the area method of that protocol (arrow 4), and select "file out as . . ." in the browser menu (arrow 5) to save the method. Later, after having completed various tasks such as creating a new *Triangle* class, the user has selected and saved the area method of the *Diamond* class (Figure 14.7[b]). At this point, the Apprentice has detected two nonconsecutive occurrences of the repetitive task "select the accessing protocol, select the area method, file out as." The action preceding the first occurrence of this repetitive task is "select the Circle class" and the action preceding the second occurrence is "select the Diamond class." Because classes Circle and Diamond belong to the MyGraphics category, it infers that the user intends to save the area method of all classes of MyGraphics category. Hence, it assumes that these two occurrences are two iterations of the following loop: "For all classes of the MyGraphics category,
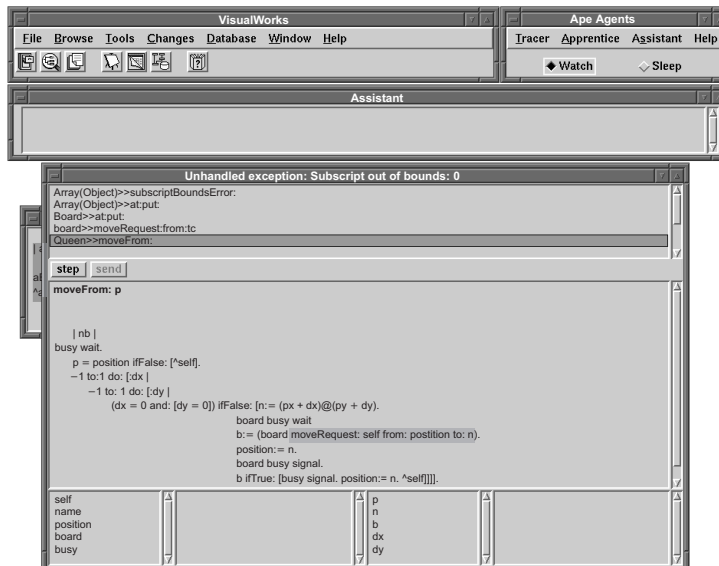
S
R
L

FIGURE **14.6**

Situation before firing a habit

The Assistant proposes to open a debugger
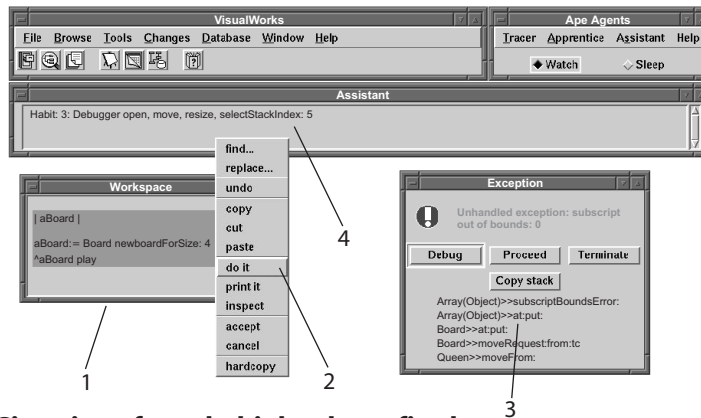
Situation after a habit has been fired

The user mouse-clicked on the proposition

The user has (1) typed and (2) evaluated an expression that (3) raised an exception. (4) The Assistant offers to open, move, and resize a debugger, and to select the fifth item in the debugger stack, as the user typically does (top snapshot). The user has mouse-clicked on the proposition, and the Assistant has performed these actions, resulting in a well-positioned debugger displaying a user's method (bottom snapshot).

S

R

L

do select the accessing protocol, select the area method, file out as." It learns a habit. As a consequence, as soon as the user selects the MyGraphics category, and whatever actions she has performed before, the Assistant predicts that she is about to save one more method area and offers to complete the loop (Figure 14.7[c]). If the user mouse-clicks on the suggestion in the Assistant window, the Assistant saves all area methods not yet saved (not shown).

14.3.4 Example 4

This last example shows that APE is able to help the user write repetitive pieces of code. Suppose a user has written several similar methods named "=", for various classes of the MyGraphics category in a browser. He has just selected the testing protocol (Figure 14.8, arrow 1) and is about to write a new method "=". The Assistant offers to insert a text template (arrow 2) containing some repetitive code (the asterisks denote nonrepetitive code). The user has mouse-clicked the suggestion, and the template has been inserted (arrow 3—"situation after a habit has been fired"—bottom snapshot).

14.4 Detecting Repetitive Tasks

This section explains what kinds of repetitive tasks the Apprentice is able to detect in the trace and how the Assistant automates them.

14.4.1 Repetitive Sequences of Actions

Detection of repetitive sequences of actions is achieved using a classical text-searching algorithm (Karp, Miller, and Rosenberg 1972). "Open, move, resize a debugger and select stack index 5" (Figure 14.6) is an example of a repetitive sequence of actions. To automate a repetitive sequence of actions, the Assistant simply replays the actions composing it.
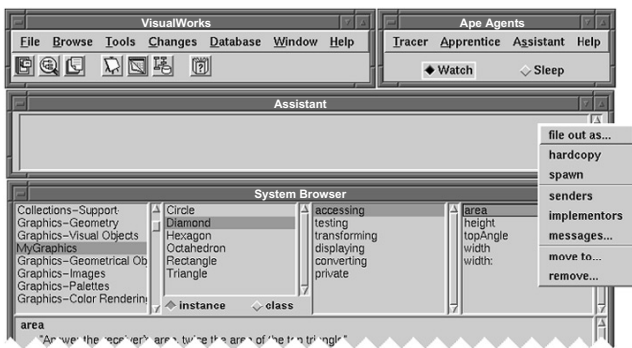
14.4.2 Loops

Each time the Apprentice detects a repetitive task, it supposes that the ___S corresponding sequence of actions could be the "body" of a loop and that ___R each occurrence of the sequence could be an iteration of the loop. It then ___L
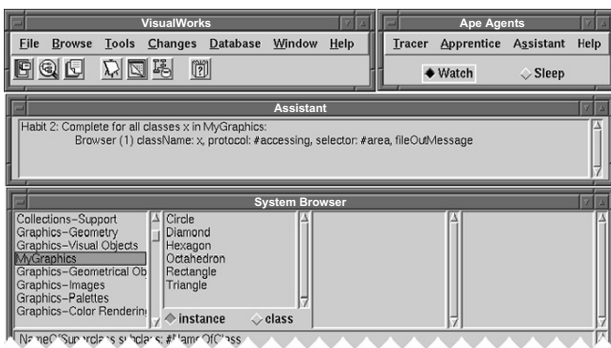
FIGURE **14.7**
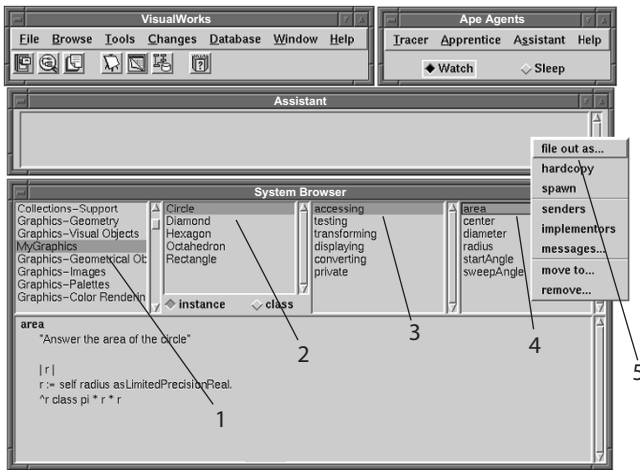
(a)

(b)

(c)

(a) The user saves ("file out as . . .") command) the method area of the class Circle. (b) After having completed various tasks, the user also saves the area method of class Diamond. (c) The Apprentice has detected a loop in the user's actions and has learned a habit. The Assistant offers to complete the loop (to save all area methods) as soon as it detects that the user is about to save one more area method.

S
R
L

FIGURE **14.8**
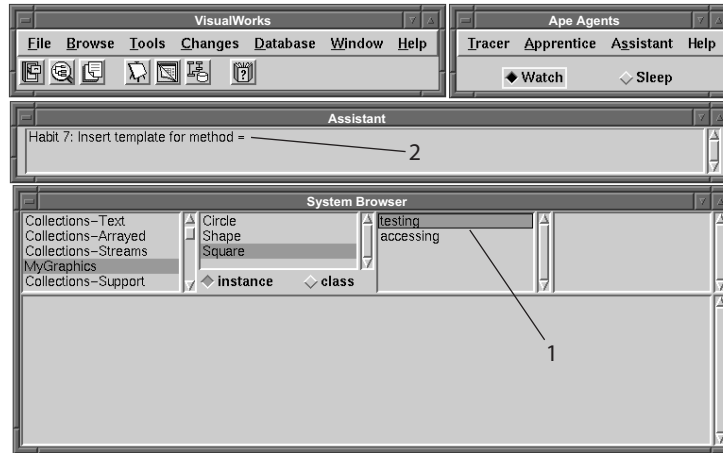
(a)
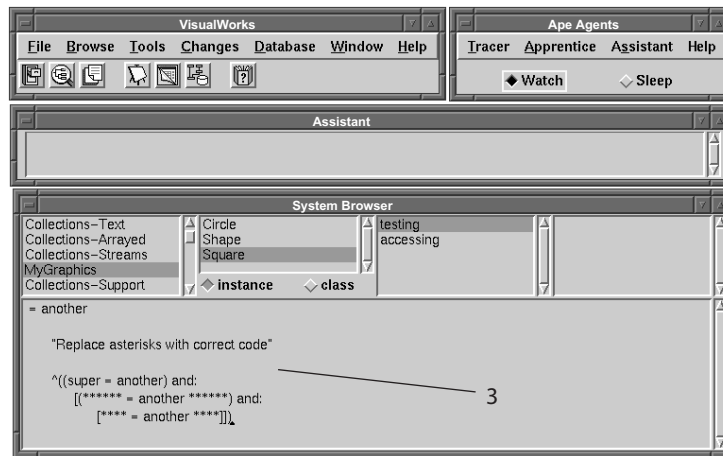
(b)

The user has written several similar methods named "=", for various classes of MyGraphics. He has just selected the testing protocol (1) and is about to write a new method "=" for class Square. The Assistant offers to insert a template (2). The user has mouse-clicked the suggestion, and the template has been inserted (3).

S
R
L

searches for relations between the actions preceding (or following) each iteration to determine the loop "variable." Example of such relations are classes belonging to the same category, methods belonging to the same class, subclasses of the same class, and so forth. Example 3 in Section 14.3 illustrates this case. The body of the loop is "Select the accessing protocol, select the area method, file out"; the action preceding the first iteration is "Select the Circle class"; and the action preceding the second iteration is "Select the Diamond class." The relation between these two actions is that classes Circle and Diamond belong to the same category. Thus, the Apprentice then infers that the selected class is the loop variable and builds the following repetitive task: "For all classes x of MyGraphics, select the x class, select the accessing protocol, select the area method, file out." To complete a loop, the Assistant plays the loop body for all the remaining values of the loop variable.

14.4.3 Writing of Repetitive Pieces of Code

To detect these repetitive tasks, the Apprentice compares the methods created by the user, line by line, using a simple string match comparison function. When it finds a set of methods that have a certain amount of their respective code in common, it assumes it has found a repetitive portion of code and create a template (Figure 14.8). To replay a writing of a repetitive piece of code, it inserts the template in the code window of the browser (again, see Figure 14.8).

14.4.4 Repetitive Corrections of (Simple) Programming Errors

The Apprentice compares the methods the user has modified and the way he did it. When it finds a set of method in which the user has replaced a portion C with another portion of code C′, it assumes it has found a repetitive correction and records the replacement. To replay a repetitive correction of code, it replays the recorded code replacement.

14.5 Learning User's Habits

Once it has detected the repetitive tasks, the Apprentice has to learn situation patterns to build the When-set and the What-set. In this section, we

explain what makes this learning task difficulty and how we overcome this difficulty.

14.5.1 What Makes the Problem Difficult?

We present in this section the requirements that direct the choice of the algorithms that the Apprentice uses to learn the situation patterns of the When-set and the What-set. Let us recall that the When-set is a set of situation patterns that match the situations in which the user has performed repetitive tasks; the What-set is a set of habits (i.e., a set of pair "situation patterns—repetitive task"). Let AL1 denote the algorithm used to build the When-set and AL2 denote the algorithm used to build the What-set.

- *Requirement 1: Low training time.* We distinguish "long-life" and "short-life" repetitive tasks. *Short-life* repetitive tasks are related to specific issues and appear in a small section of the trace, and the corresponding situation patterns have to be learned very rapidly from a few situations. *Long-life* repetitive tasks can reflect the general user's habits and can require a very long trace to be detected. Thus, the Apprentice is, on the one hand, able to learn situation patterns very rapidly on a small section of the trace to capture short-life tasks and, on the other hand, can also consider very long traces corresponding to several work sessions. Let us call *training time* the time required by a machine-learning algorithm to learn situation patterns. AL1 and AL2 must have a low training time.

- *Requirement 2: Low prediction time.* Of course, the Assistant is able to decide when to make a suggestion and which suggestion to make very rapidly. Let us call *prediction time* the time it takes to the Assistant to inspect a set of situation patterns and to determine which ones match the current situation. The prediction time depends on the way AL1 and AL2 represent the learned situation patterns. This prediction time must be very low to allow the Assistant to make suggestions (or to decide not to make a suggestion) after each user's actions.

- *Requirement 3: User-intelligible situation patterns.* We want the Apprentice to represent situation patterns in a humanly understandable way. This is not a critical requirement, but it allows the user to inspect or edit the learned habits. Comprehensible and controllable interfaces give the user a sense of power and control.

- *Requirement 4: AL1—Low error rate.* Finally, to be viable our Assistant has to make the "right suggestion at the right moment." This means that

it has to determine correctly when to make a suggestion. AL1 is said to "make" an error in two cases: (1) when one of the situation patterns it has learned matches the current situation but the user is not about to perform a repetitive task or (2) when none of its situation patterns matches the current situation but the user is about to perform a repetitive task. In case 1, the Assistant makes suggestions yet no suggestions should be offered, in case 2, it makes no suggestion when doing so would benefit the user. AL1 must have a low error rate.

- *Requirement 5: AL2—Low error rate and low generalization.* The Assistant also has to determine correctly what to suggest. AL2 is said to "make" an error in two cases: (1) when a situation pattern of a habit matches the current situation but the user is not about to perform the repetitive task of that habit or (2) when a situation pattern of one of the habits matches the current situation but AL1 has made an error[3] and the user is not about to perform a repetitive task. In case 1, the Assistant suggests the wrong repetitive task; in case 2, it makes suggestions but no suggestion is expected. Case 2 may occur if the situation patterns of the habits are too general. A too-general situation pattern will be matched by too many situations and the corresponding task proposed too frequently. AL2 must have a low error rate to make few errors in case 1, but it also has to generalize as little as possible to avoid too-general situation patterns in case 2.

14.5.2   Which Algorithms?

Various kinds of algorithms have been proposed in the field of machine learning. Concept learning, neural networks, and reinforcement learning algorithms do not meet requirement 1; instance-based algorithms do not meet requirement 2; instance-based, statistical, neural networks and reinforcement learning algorithms do not meet requirement 3.

Eligible kinds of algorithms are decision-tree algorithms because they miss none of the requirements 1, 2, or 3. These algorithms, like most of the machine-learning algorithms, have a low error rate and meet requirement 4. Although they have not been designed to generalize as little as possible,

---

3. Whatever algorithm is used to build the When-set, it may sometimes make errors: machine-learning algorithms with a null error rate have not been discovered yet and even do not exist for most of the learning tasks.
4. A commercial version of C4.5, called C5.0, is now available.

S
R
L

they are better suited for requirement 5 than instance-based or statistical algorithms.

Decision-tree learning algorithms have notably been used in CAP (Mitchell et al. 1994). The state-of-the-art decision-tree learning algorithm is C4.5 (Quinlan 1993).[4] C4.5 has low computing time (incremental versions of C4.5 exist) and is suited to learn the When-set (AL1). However, our tests (see Section 14.6) have shown that it learns too-general situation patterns and is not suited to learn the What-set (AL2). Hence, APE employs C4.5 to learn the When-set and a new algorithm we have designed to learn the What-set.

14.5.3 A New Algorithm

Our new algorithm, named IDHYS, is a concept-learning algorithm inspired by candidate elimination algorithm (Mitchell 1978).

Inductive concept learning consists of acquiring the general definition of a concept from training examples of this concept, each labeled as either a member (or positive example) or a nonmember (or negative example) of this concept. Concept learning can be modeled as a problem of searching through a hypothesis space (set of possible definitions) to find the hypothesis that best fits the training examples (Mitchell 1982). Learning user's habits is a concept-learning problem. All the situations preceding a repetitive task T can be seen as positive examples of the concept "situations in which the user is going to perform repetitive task T." The situations preceding any other task can be considered as negative examples of this concept. The searched definition is a set of situation patterns that match the situations in which the repetitive task T has been performed.

IDHYS searches the hypothesis space of the conjunctions of two situation patterns, one for each kind of situation pattern defined in Section 14.2: containing wild cards or unordered containing wild cards. It learns by building hypotheses that are the most specific generalizations of the positive examples. It processes the positive examples incrementally. It starts with a very specific hypothesis (indeed, the first positive example itself) and progressively generalizes this hypothesis with the subsequent positive examples. IDHYS does not build hypotheses for the negative examples, which are only used to bound the generalization process. This incremental bottom-up approach makes IDHYS not sensitive to actions with large sets of possible values for their parameters. As a consequence, it has a low computing time. Our test (see the next section) shows it also has a low error rate and does not overgeneralize to build the situation patterns.

S
R
L

Description of an earlier version of IDHYS can be found in Ruvini and Fagot (1998) and of a more complete one in Ruvini 2000).

14.6 Use and Experimental Results

APE, as described in the prior section, is implemented and experimentally used by ourselves and by a pool of fifty students enrolled in a Smalltalk course at the master's level. This section first analyzes user feedback and then describes and analyzes the technical experimental results.

Concerning user feedback, let us recall that our users are Smalltalk beginners, we do not yet have feedback from experienced programmers. About 70 percent of our students have considered the Assistant window for approximately ten minutes and then have forgotten it. Fortunately, results from the remaining 30 percent have been very interesting. The main reasons invoked by those who have not used the suggestions are

- the burden of looking at the Assistant window since nothing, except a modification inside this window, indicates when a suggestion is made;

- the difficulty of reading suggestions presented as a sequence of actions.

This feedback indicates that a great deal of work in that direction remains—namely, how to gently alert people and provide a better visualization of what the Assistant suggests? In addition, the interesting point is that those who have made the effort to use the tool have rapidly learned how to use it efficiently and have taken advantages of its capabilities. After a while, those users have learned (1) which suggestions are regularly made and which ones interest them and (2) when the suggestions are made. In other words, they have learned to take a look at the Assistant window when they are about to perform a repetitive task and when they do know that the suggestion will be made. The students have been able to anticipate APE's suggestions because APE has a low excess rate and makes few wrong suggestions.

Concerning technical results, APE correctly works and makes the suggestions we expected it to. It also makes many suggestions we did not think of. We report here experiments conducted on ten traces of 2,000 actions, collected during students' usage of the software.

How well does APE assist its users in practice? One way to answer this question is to train APE on a part of a trace (called the *train trace*) and then to test it on another part (called the *test trace*) to see how often one of its

suggestions coincides with user's actions.[5] Figure 14.9 plots these data. The horizontal axis gives the size of the train traces and the test traces used. APE employs C4.5 to learn the When-set and IDHYS to learn the What-set denoted by "C4.5-IDHYS").

However, as a comparison, we also report results when the Apprentice employs C4.5 to build both the When-set and the What-set (denoted by "C4.5-C4.5"), and when it learns the What-set only (and not the When-set) with C4.5 and IDHYS, respectively (denoted by "C4.5" and "IDHYS").The percentage of correct suggestions (a) is the percentage of repetitive tasks correctly suggested by the Assistant. The percentage of excessive suggestions (b) is the percentage of actions of the test trace not preceding a repetitive task for which the Assistant has made a suggestion. These percentages have been evaluated for a situation length varying from 1 to 10, and Figure 14.9 presents average results. This figure shows that

- the use of the When-set decreases the amount of excessive suggestions without decreasing the amount of correct suggestions,

- employing IDHYS to learn the What-set leads the Assistant to make less excessive suggestions and to suggest correctly more repetitive tasks, and

- the percentage of excessive suggestions increases with the trace size (see "C4.5-IDHYS").

Both the percentage of correct suggestions and the percentage of excessive suggestions increase with the situation length (not shown here). This shows that there is a trade-off between finding an assistant that makes few but correct suggestions (and perhaps misses some repetitive tasks) and finding one that constantly bothers the user with suggestions. Practically, we have chosen a learning frequency of 100 actions and a situation length of 3 actions. In this case, APE correctly suggests 63 percent of the repetitive tasks and makes an excessive suggestion for only 18 percent of users' actions. The average size of the repetitive tasks suggested by the Assistant during this experiment was seven actions (minimum, three; maximum, ten).

Another important question is how long it takes APE to learn users' habits. in practice, it takes APE fifteen seconds (measured on a PC with a 133-Mh processor) to learn users' habits (i.e., both the When-set and the What-set) from a trace containing 100 actions. A 100-action trace corresponds to about six minutes of user work. In other words, for every six minutes of user work, APE spends fifteen seconds to learn new habits. This result is quite

─────────

5. This is similar to the machine learning cross-validation process.

S
R
L

FIGURE 14.9

Percentage of correct suggestions

(a)

Percentage of excessive suggestions

(b)
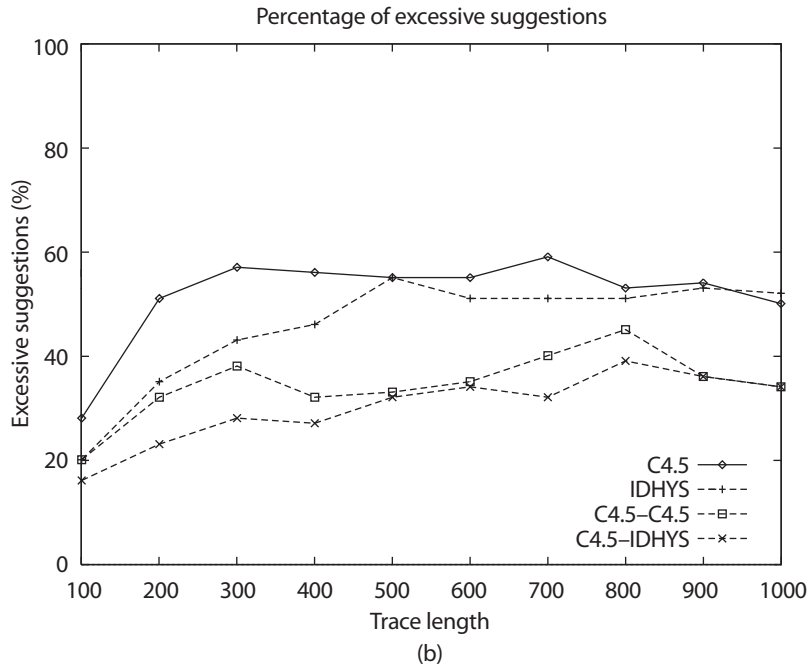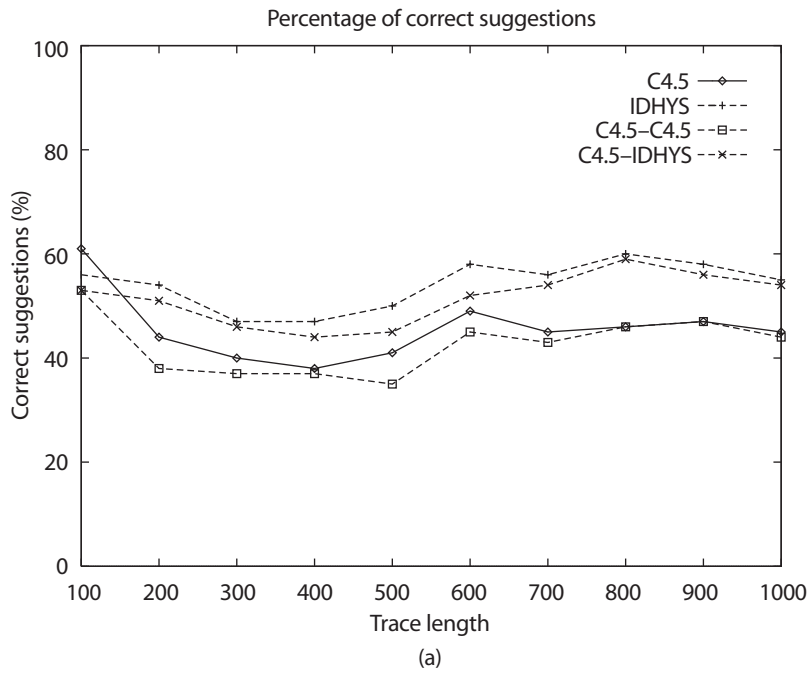
(a) The percentage of repetitive tasks that APE has correctly suggested and (b) the percentage of actions of the user, not preceding a repetitive task, for which APE has made a suggestion.

S
R
L

satisfactory. Note also that the Assistant makes suggestions (i.e., inspects the When-set and the What-set) in a matter of milliseconds.

14.7 Conclusion and Prospects

APE is one more step toward assistants that bring together programming by demonstration and predictive interfaces. It works and operates in a context in which the number of possible actions and possible values for the parameters of these actions are large and repetitive sequences are not known in advance and not consecutive. It is able to replay repetitions composed of several actions and containing loops. The lessons learned from this work are as follows:

• Minimizing the amount of wrong suggestions is an important issue. The system has to suggest the "replaying" of the right repetitive task at the right moment.

• Learning when to make a suggestion as well as what to suggest decreases the number of incorrect suggestions.

• The system must not learn too general habits. We have shown that our new machine-learning algorithm designed to learn habits reduces the amount of incorrect suggestions without degrading the quality of these suggestions.

• It is possible to learn a user's habits to anticipate repetitive tasks. Experimental tests have shown that APE is usable and efficient: it learns a user's habits in a matter of seconds and anticipates 63 percent of the repetitive tasks. It makes irrelevant suggestions for only 18 percent of the user's actions.

Concerning future work, it is clear that one of the main remaining issues is to present the suggestions made by the Assistant in a more attractive way. Programming by demonstration studies have addressed the problem of creating a graphical representation of a program or a sequence of actions, and they offer a direction for future research.

Although the integration of APE into a programming environment is an originality, it is not restrictive. APE could be integrated in other interactive environments such as Microsoft Windows, X window system, or Apple MacOS.

S

R

L

References

Armstrong, R., D. Freitag, T. Joachims, and T. M. Mitchell. 1995. WebWatcher: A learning apprentice for the World Wide Web. Paper presented at the AAAI spring symposium on information gathering.

Böcker, Hans-Dieter, and Jürgen Herczeg. 1990. What tracers are made of. *In Proceedings of the OOPSLA/ECOOP '90 conference on object-oriented programming systems, languages and applications* (October).

Caglayan, A., M. Snorrason, J. Jacoby, J. Mazzu, R. Jones, and K. Kumar. 1997. Learn sesame: a learning agent engine. *Applied Artificial Intelligence* 11: 393–412.

Cypher, A., D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, eds. Watch what I do: Programming by demonstration. Cambridge, Mass.: MIT Press; see also www.acypher.com/wwid/.

Darragh, J. J., and I. H. Witten. 1991. Adaptive predictive text generation and the reactive keyboard. *Interacting with Computers* 3, no. 1:27–50.

Karp, R. M., R. E. Miller, and A. L. Rosenberg. 1972. Rapid identification of repeated patterns in strings, trees and arrays. In *4th Annual ACM Symposium on Theory of Computing* (Denver, May 1–3).

Lieberman, Henry. 1993. Mondrian: A teachable graphical editor. In Watch what I do: Programming by demonstration, ed. A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky. Cambridge, Mass.: MIT Press.

Maes, P. 1994. Agents that reduce work and information overload. *Communications of the ACM* 37, no. 7 (July, special issue on intelligent agents): 31–40.

Mitchell, T. M. 1978. Version spaces: An approach to concept learning. Technical Report HPP-79–2, Stanford University, Palo Alto, Calif.

———. 1982. Generalization as search. *Artificial Intelligence* 18, no. 2 (March): 203–226.

Mitchell, T. M., R. Caruana, D. Freitag, J. McDermott, and D. Zabowski. 1994. Experience with a learning personal assistant. *Communications of the ACM* 37, no. 7 (July, special issue on intelligent agents): 81–91.

Moon, D. A. 1986. Object-oriented programming with flavors. In *Proceedings of the conference on object-oriented programming systems, languages, and applications (OOPSLA)* (New York, November), ed. Norman Meyrowitz. New York: ACM Press.

Motoda, H. 1997. Machine learning techniques to make computers easier to use. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*. (August 23–29). San Francisco: Morgan Kaufmann.

Quinlan, J. R. 1993. *C4.5: Programs for machine learning*. San Mateo, Calif.: Morgan Kaufmann.

S

R

L

Ruvini, Jean-David. 2000. Assistance à l'utilisation d'environnements interactifs: Apprentissage des habitudes de l'utilisateur. Ph.D. diss., Université Montpellier II, France.

Ruvini, Jean-David, and Christophe Fagot. 1998. IBHYS: A new approach to learn users habits. In *Proceedings of ICTAI'98.* Los Alamitos, Calif.: IEEE Computer Society Press.

Teitelman, W. 1978. *Interlisp reference manual.* Palo Alto, Calif.: Xerox Palo Alto Research Center.

S

R

L

S

R

L