CHAPTER

Eight

Demonstrating the Hidden Features That Make an Application Work

RICHARD MCDANIEL

Siemens Technology-To-Business Center

S
R
L

Abstract

With programming-by-demonstration (PBD), a user shows examples of a behavior that the computer is meant to perform instead of writing out textual instructions. Many PBD systems use machine-learning techniques to convert the user's examples into executable programs automatically. However, in order to use PBD as a general-purpose programming tool, the user must be able to demonstrate more than just the surface level interactions of a behavior. By combining PBD with interaction techniques for specifying data that is normally hidden, and by selecting important data at key moments, the user is able to give hints to the computer that clarify the vagaries of using examples alone. This enables PBD to be used in more situations where previously only textual languages could be used.

8.1 Introduction

Many programming-by-demonstration (PBD) systems have used machine-learning techniques to understand a user's intention. With machine learning, a PBD system can infer a program automatically without requiring the user to learn a complicated programming language. Unfortunately, machine learning can also be temperamental, unreliable, and, to the untrained user, mysterious. Since machine learning can fail, the system must keep the user in control and informed. Even simple programs can be too difficult to infer by examples alone. The user must be allowed to provide information beyond examples that gives the system information about the program's internal workings. The key is to gather this extra information without placing too many burdens on the user.

In the Gamut project (McDaniel and Myers 1999), I have experimented with techniques for seamlessly gathering the important information needed to infer complex behaviors. These include new interaction techniques that are used to demonstrate the desired behavior as well as new programming elements that the user includes to provide extra information. Many of these techniques were originally developed for other systems, including some that were not PBD systems. By combining these techniques into a single system, along with appropriate machine learning that uses these techniques, we can make PBD as practical as programming using

a typical language like C++, while at the same time still much easier to learn for nonprogrammers.

8.2 The Perils of Plain Demonstration

The machine learning in PBD can be seen as a form of *inductive* learning in which the computer infers more general properties from a set of specific examples. The examples thus become the user's primary means for communicating with the system. This method works well in principle because the user knows the task he or she wants the computer to perform. The user can provide examples by simply performing the behavior manually while the system watches. However, the computer cannot read the user's mind. When the user demonstrates, there are always variables inside his or her head. Sometimes the desired behavior depends on these variables, but the user is often not aware that the variables exist and may assume that their state is obvious.

Trying to coax the user to reveal hidden state is quite difficult; as a result, most PBD systems try to work without it. For instance, PBD systems use two typical interfaces to gather examples that I will call the "passive watcher" and using "explicit examples." The main difference between the techniques is who (or what) is responsible for determining when programming is desired. Using explicit examples, the user is responsible, whereas with the passive watcher, the computer is responsible.

The most commonly attempted interface is the passive watcher. The idea is that the computer acts like a helpful assistant who is constantly watching you as you work. Sometimes the system will recognize what you are doing, and if it can offer assistance, it will do so. A good example of this is in Microsoft Word's auto-correct, auto-indent, dictionary, and other features that all occur automatically because the system is passively recognizing patterns in the user's document as it is typed.

A passive watcher cannot normally request hidden state information from the user. The object that the user is creating is usually not a program, so the state cannot be represented as part of the product. For example, in a word processor, the product being created is a text document, not a program. Any behavior that a PBD system generates would not appear in the document itself, so the user has no obvious way to manipulate or refine the inferred behavior without switching to an alternate mode or editor. As a result, the passive watcher style of PBD operates by inferring as little hidden

S
R
L

state as possible. This limits it to expressing behaviors with at most one state variable, making general-purpose programming practically impossible. The main advantage of the passive watcher is that its service is not critical in performing the user's task. If it cannot infer a behavior for a particular situation, the user just performs the task manually as though the PBD system did not exist.

In the second PBD approach that uses explicit examples, the user is usually creating an application and has drawn the application's interface components using a graphical editor. The user shows the PBD system before and after samples of the desired behavior using the interface's components. The system then converts these examples into code by noticing which components have changed and inferring the constraints between those changes. Of course, this technique is not restricted to building graphical interfaces. It can be applied to any domain whose essential features can be treated like objects. For instance, consider a greenhouse application in which graphical objects are used to represent water hoses, sprinklers, motors connected to windows, and temperature gauges. Demonstrating behaviors with the graphical objects could be used to create programs for watering the user's geraniums.

Unlike the passive watcher interface, the products the user creates using explicit examples are behaviors within an application. As a result, the user expects the system to create behaviors on demand. The system cannot passively choose to fail when it cannot recognize what the user is trying to accomplish. As a result, inferring behaviors from explicit examples is especially demanding.

If the goal is to make a general-purpose programming system using PBD, one must certainly use an explicit example approach. Unfortunately, many implementations of the explicit example approach still fail to handle hidden state. Without the ability to represent complex state, a system cannot infer behaviors any more complicated than those inferred by a passive watcher and will typically fail.

8.3 Who Is Actually Programming?

A common misconception about PBD is that the computer system performs all the programming work. In truth, the user is performing all the difficult conceptual work. The computer simply provides a more convenient method for recording the user's thoughts. In some senses, a PBD environment is like an advanced programming editor. However, unlike most

editors, the user is not expected to write code textually. Instead, code is constructed by demonstrating the key behaviors of the application. This is still programming. The user must still carefully consider how the application works and must plan to demonstrate each behavior so that the computer can record it correctly.

Recognizing that the user is still a programmer eliminates the need for the system to magically interpret the user's every whim. Stated simply, if the user does not know how a behavior works, the system is not going to know, either. The goal of a PBD system should not be to try to do everything for the user but, instead, to facilitate the programming to make it easier to perform.

Since the user is in charge, the system must provide sufficient mechanisms for the user to communicate how a behavior functions. This means that the user must be able to represent the abstractions on which a behavior depends. Properly representing an abstraction is a generally difficult problem. More important, different users will represent the same abstraction differently. Thus, the system must be flexible to allow the user to specify an abstraction in the way that is the most comfortable for that individual.

8.4 Giving the System Hints

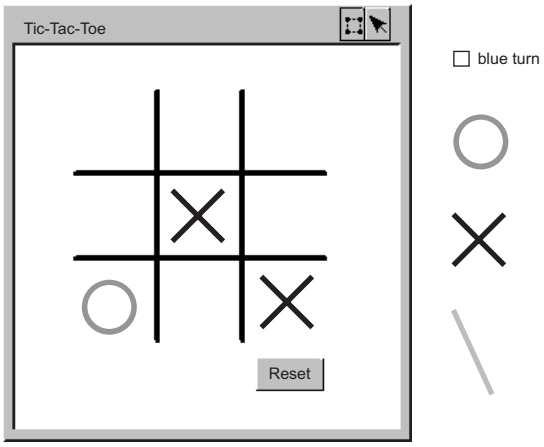
To demonstrate a complex behavior using explicit examples, the user must also be able to represent the abstractions that make the behavior work. The visible components of an application's interface do not always show all the aspects of the application's abstractions. For instance, in a chess game, there is nothing about the appearance of a bishop that suggests that it can only move diagonally. The techniques used to communicate information that is not apparent in the application's visible interface to address this problem are typically called *hints*. Hints provide extra channels for the user to represent things that the system would find too difficult to infer on its own.

8.4.1 Creating Special Objects

Many ways to give a system hints are possible including creating special objects and programming widgets, as well as using selection techniques for pointing out objects at key times. Most of these techniques come from

S
R
L

FIGURE **8.1**



A window frame surrounding a visible game area, with other objects stored along the margins to represent invisible state.

systems that do not use PBD and do not infer code. In many ways, these techniques can be applied to any programming system since the need to represent state is universally applicable.

A common strategy is to give the user a secondary area to draw objects that do not appear in the application's interface. This area acts like the margins in a book where the user can write down notes without writing over the main body of text. In the main application window of my own system, Gamut, which is a PBD programming language for building video games, a window frame encloses the graphics that will become the application's visible area (see Fig. 8.1). The rest of the area is all margin space for the user's off-screen objects. Gamut's margins are essentially the same technique as the "offstage area" in Gould and Finzer's (1984) Rehearsal World, which was an application builder that used a stage-acting metaphor. The offstage area was a separate window where the user could place unseen objects.

Objects for representing state can also be intermingled with visible objects on the screen. Several systems use techniques for drawing lines to represent geometric constraints. In Fischer, Busse, and Wolber's (1992) DEMO II, which was a PBD system for inferring small graphical behaviors, these were called "guidewires." In other systems, such as Jackiw and Finzer's

FIGURE **8.2**

Using an arrow line to indicate an object's speed and direction.

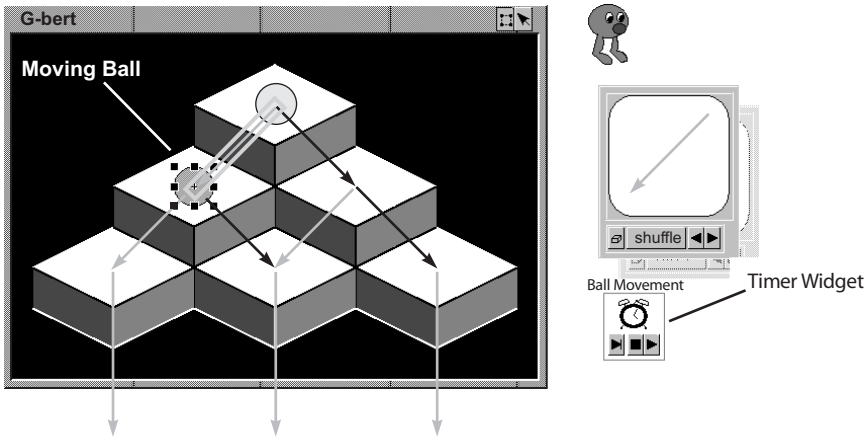
(1992) Geometer's Sketchpad, which was a tool for experimenting with geometric constraints, these objects were not even given special names. Figure 8.2 shows a scene in Gamut where an arrow is used to show an object's direction of motion in a video game. The user created the arrow to show the spaceship's speed and direction, which is not apparent from the ship's image. During demonstration, the user moves the ship and arrow combination to the arrow's end point and highlights the arrow's original position to tell Gamut that it is important.

Besides representing state, objects can also be used to represent common behaviors. Widgets are objects such as buttons and sliders that have not only graphical properties but also intrinsic behavior. For example, buttons can be pushed, and sliders can be adjusted to different values. These behaviors can be incorporated into an application without having to demonstrate the intrinsic behavior of the widget. Widgets can also be used to specify more complex behaviors. When the behavior of a widget becomes similar to operations that one performs in a programming language, it becomes a "programming widget." Timers are a common kind of programming widget. For example, the timer in Figure 8.3 causes the ball to move down the pyramid. The game randomly selects between two colors each time the timer ticks and moves the ball along the arrow with the matching color.

Though programming widgets can be powerful, the PBD system designer must be careful not to rely on them too heavily. For instance, in Rehearsal World, widgets were used to represent if-then conditions and for-loops. The user would draw these widgets in a window and then fill in their parameters with short segments of code. Widgets, when used in excess, become a programming language on their own. One of the goals of using PBD is to lessen the need for the user to program the structure of the application's code. If widgets are used too heavily, they can cause the same problems as writing code textually.

S
R
L

FIGURE **8.3**



A timer widget used to cause a ball to bounce downward randomly in a game.

8.4.2 Selecting the Right Behaviors

As important as representing state is, it is equally important that the PBD system applies that state in the right way. Like many artificial intelligence (AI) problems, inferring code in PBD can be seen as a searching task. The system must find the appropriate expressions and constraints for representing the behavior the user is demonstrating. Many different behaviors can always be produced from a given set of examples.

Two basic methods are used for bounding the number of choices the system can make when creating a behavior. In the first, the system writes out a list of possibilities and the user picks the most appropriate one. This is the technique that Kurlander and Feiner's (1988) Chimera uses, for example. Chimera was a tool for performing operations such as search and replace on complicated graphical drawings. When the different behavioral choices can be articulated in a few terse phrases, this technique works fairly well. However, for more complicated decisions such as when the choice concerns an expression that is nested several layers deep within the code, the user cannot always pick the right choice reliably. In these cases, the system can reduce its search space by having the user select the objects that the behavior uses. This technique is sometimes called *giving the system a focus hint.*

S
R
L

Examples of focus hints are shown in Figure 8.2 and 8.3. In both cases, a user is demonstrating to Gamut that a character in a game is following a path. To tell the system that the path is important, the user highlights it. This allows the system to focus on the path object and only generate constraints between the path and the moving object. Maulsby's (1994) Cima, which could recognize textual patterns using examples, also allowed the user to give focus hints. In addition, Cima allowed the user to provide negative focus hints, where marking an object indicated to the system that the object should not be used in the current behavior.

8.5 The Programming Environment Matters

With the ability to specify and point out all the state variables in an application, a PBD language can theoretically be as powerful as any textual language. In fact, Gamut has been shown to be Turing-complete (i.e., it can be used to create any Turing machine, making it computationally equivalent to any another programming language). However, anyone who has ever tried to program a Turing machine knows that theoretical equivalence makes no guarantee that a language is practical. The environment and feel that a PBD system provides are major factors in whether a user would want to use it.

Having a PBD language that is computationally equivalent to textual languages is only one step. People always ask, "Can your system do *X*?" Where *X* is some behavior that they think a PBD system could never possibly infer. The answer is invariably no. The fact is that a complete language definition is not sufficient to actually build applications. If one were to take an ordinary programming language like C and ask, "Can I use C to print 'hello world' to the screen?" the answer would also be no if it were not for the presence of the stdio.h library that provides the printf procedure. The power of a language is not entirely in its definition but also in the libraries that allow the language to make interfaces with the world. PBD languages are still immature in that none have any library capabilities. No language can provide comprehensive support for every possible application. At some point, the language has to be extended using a library interface.

Another important aspect of the environment is the manner in which the user fixes bugs. It is essentially impossible in any programming language to produce bug-free code, and this is still true in PBD languages. While it is generally not possible for the system to know whether the user's

application has a bug, the system can facilitate how easily the user can find problems and fix them once they are discovered.

The first requirement is that it should be possible to test any change the user makes immediately. If the user is forced to make several changes before the application can be tested and the application still does not work, it is very hard to discern the effects of the changes. It should be possible for the user to correct a problem as soon as it is discovered and see immediately whether the problem has been fixed. In Gamut, the system's inferences are represented as interpretable code. When the user adds a new example, the code is immediately modified to reflect the change, and it can be run as soon as desired.

The second requirement is that the user needs feedback to understand what the system is creating. For example, the system can use visualization techniques to show the user pictorial representations of the code, as in Chimera. One might also draw a dependency network to show which objects have behaviors and which objects those behaviors will affect.

The most explicit kind of feedback the system can provide for a behavior is to show its code. In fact, in some systems, the user might only demonstrate one example, after which the user is required to edit the code in order to fix all the errors the inference system made. This is essentially how a macro recorder works. The user records a single example using the macro recorder and then edits the code the recorder produces to add parameters, loops, and conditional expressions. Though this method works in principle, nonprogrammers generally find it difficult to write code even in small doses. So, although it is probably a good idea to make the inferred code available, forcing the user to write code is generally not good for a PBD system.

8.6 Conclusion

Programming by demonstration can be a powerful method for building applications, but it still the user's responsibility to know how the behaviors in the application work. By using the proper objects and techniques, a user can create practically any behavior, but the user must still learn how to use those objects and techniques. The user must still convert the abstract concepts that the behaviors use into representations that the computer can understand. While PBD can help make this process easier, it cannot eliminate the creative process the user must perform to turn an idea into code.

To represent the state, the user must be able to create objects that are not visible in the application interface. These objects might be placed in the margins of a window or in separate regions entirely. The user should also be allowed to draw special-purpose objects in the visible regions of the application to show the invisible graphical constraints that the application uses.

For a behavior to use hidden state, the user should be able to point out appropriate objects at key moments during demonstration. This can be accomplished by selecting objects or by selecting which code the system creates. Generally, techniques that do not rely on using code are more appropriate for nonprogrammers, but in some circumstances, selecting from a list of choices is fairly easy. Selecting objects, though, can apply to situations where the system cannot present a terse description.

Finally, the programming environment that the system presents is as important as the PBD language itself. The environment plays a key role in making an application easy to debug and to understand. Program visualization and other feedback are important in keeping the user informed. Furthermore, being able to test behaviors immediately allows the user to know quickly whether a change has been effective and allows the user to be confident that a behavior works.

One might ask that if PBD is not the panacea that lets every user program regardless of what they know, why bother in the first place? Why not encourage users to learn to program using standard techniques? The answer is simply that expressing a program using examples is tremendously easier and faster than writing a program textually. Any mental gymnastics that the user must make to represent hidden data in a PBD interface pales in comparison to the representational nightmares that the user might undergo to represent an equivalent structure in a programming language. Seasoned programmers can sometimes forget the years of training and effort required to learn a textual language. On the other hand, I have seen novices learn to construct complex data structures graphically in Gamut in hours. Granted, using PBD is not free from mental commitment. It only seems that way when compared to textual programming languages.

Though it has strong potential, PBD has not had much commercial success, but PBD research has only recently moved out of the laboratory and into real products. PBD seems to be following the path that AI software has blazed. The initial systems showed promise but fell short of expectations and languished in the backlash. But slowly, the concepts worked their way into common systems with new names such as *fuzzy logic* and *search*

engines. PBD will also work its way into the mainstream. However, it is still unclear what it will be called.

References

G. L. Fischer, D. E. Busse, and D. A. Wolber, 1992. Adding rule-based reasoning to a demonstrational interface builder. Paper presented at *ACM Symposium on User Interface Software and Technology, UIST'92,* Monterey, Calif., November. New York, NY: ACM.

L. Gould and W. Finzer. 1984. *Programming by rehearsal*. Palo Alto, Calif.: Palo Alto Research Center, Xerox Corporation.

R. Jackiw. 1992. *The geometer's sketchpad*. Berkeley, Calif.: Key Curriculum.

D. Kurlander and S. Feiner. 1988. Editable graphical histories. Paper presented at *IEEE Workshop on Visual Languages,* Pittsburgh, Penn., October. New York, NY: Computer Society.

D. Mauslby. 1994. *Instructible Agents*. Ph.D. diss. University of Calgary, Calgary, Alberta.

R. G. McDaniel and B. A. Myers. 1999. Getting more out of programming-by-demonstration. Paper presented at *ACM CHI'99Human Factors in Computing systems,* Pittsburgh, Penn., May. New York, NY: ACM.

S
R
L