Introduction

HENRY LIEBERMAN

Media Laboratory
Massachusetts Institute of Technology

S
R
L

When I first started to learn about programming (many more years ago than I care to think about), my idea of how it should work was that it should be like teaching someone how to perform a task. After all, isn't the goal of programming to get the computer to learn some new behavior? And what better way to teach than by example?

So I imagined that what you would do would be to show the computer an example of what you wanted it to do, go through it step by step, have the computer remember all the steps, and then have it try to apply what you showed it in some new example. I guessed that you'd have to learn some special instructions that would tell it what would change from example to example and what would remain the same. But basically, I imagined it would work by remembering examples you showed it and replaying remembered procedures.

Imagine my shock when I found out how most computer programmers actually did their work. It wasn't like that at all. There were these things called "programming languages" that didn't have much to do with what you were actually working on. You had to write out all the instructions for the program in advance, without being able to see what any of them did. How could you know whether they did what you wanted? If you didn't get the syntax exactly right (and who could?), nothing would work. Once you had the program and you tried it out, if something went wrong, you couldn't see what was going on in the program. How could you tell which part of the program was wrong? Wait a second, I thought, this approach to programming couldn't possibly work!

I'm still trying to fix it.

Over the years, a small but dedicated group of researchers who felt the same way developed a radically different approach to programming, called *programming by example* (PBE) or sometimes *programming by demonstration* (the user demonstrates examples to the computer). In this approach, a software agent records the interactions between the user and a conventional "direct-manipulation" interface and writes a program that corresponds to the user's actions. The agent can then generalize the program so that it can work in other situations similar to, but not necessarily exactly the same as, the examples on which it is taught.

It is this generalization capability that makes PBE like macros on steroids. Conventional macros are limited to playing back exactly the steps recorded and so are brittle because if even the slightest detail of the context changes, the macro will cease to work. Generalization is the central problem of PBE and, ultimately, should enable PBE to completely replace conventional programming.

S
R
L

Significantly, the first real commercial market for PBE systems might be children. Children are not "spoiled" by conventional ideas of programming, and usability and immediacy of systems for them are paramount. We'll present two systems that have been recently brought to market and are enjoying enthusiastic reception from their initial users. David Smith, Allen Cypher, and Larry Tesler's Stagecast Creator, evolved from Apple's Cocoa/KidSim, brings rule-based programming by example to a graphical grid world. Alexander Repenning and Corrina Perrone-Smith's AgentSheets operates in a similar domain and for a similar audience. Ken Kahn's ToonTalk, a programming system that is itself a video game, uses a radically different programming model as well as a radical user interface. The crucial problem of generalizing examples gets solved in a simple, almost obvious way—if you remove detail from a program, it becomes more general.

One way in which PBE departs from conventional software is by applying new techniques from artificial intelligence (AI) and machine learning. This approach opens up both a tremendous opportunity and also some new risks. Brad Myers and Rich McDaniel treat the thorny issue of "How much intelligence?" from their wide experience in building a variety of PBE systems.

Of course, we can't convince people about the value of programming by example unless we have some good examples of application areas for it! Next, we move to some application areas that show how PBE can really make a difference. Everybody's current favorite application area is the Web. The Web is a great area for PBE because of the accessibility of a wealth of knowledge, along with the pressing need for helping the user in organizing, retrieving, and browsing it. The emerging developments in intelligent agents can help—but only if users can communicate their requirements and can control the behavior of the agent. PBE is ideal.

Atsushi Sugiura's Internet Scrapbook automates assembling Web pages from other Web sources, and he also explores Web browsers on small handheld devices. Matthias Bauer, Dietmar Dengler, and Gabriele Paul present a mixed initiative system: at each step, either the user or the agent can take action, cooperating to arrive at a "wrapper" that describes the format of Web pages.

Carol Traynor and Marian Williams point out the suitability of PBE for domains that are inherently graphical, such as Geographic Information Systems. If you can see it and point to it, you should be able to program it. PBE lets users see what they are doing, unlike conventional programming languages, in which graphical data can only be referenced in a program by filenames and numbers. They illustrate the utility of PBE for "user-

programmers"—those who specialize in the use of a particular application but also, at least occasionally, have the need to resort to programming. Patrick Girard embeds a PBE system in an industrial-strength computer-aided design (CAD) application. Designers of mechanical, electrical, manfacturing, or architectual systems can see the objects they are trying to design directly. Rich McDaniel moves from static graphics to the dynamic world of computer games, showing how interaction techniques can also demonstrate "hidden features" of applications that are not directly reflected in the graphics that the user will eventually see but are nevertheless crucial.

PBE can automate many common but mundane tasks that tend to consume a frustratingly large fraction of people's time. Text editing remains the application that people spend the greatest amount of time in, and so text editing applications are the target of the next set of PBE systems we'll look at. Tetsuya Masuishi and Nobuo Takahashi use PBE successfully for the common editing task of generating reports. Toshiyuki Masui's Dynamic Macro and PoBox systems use loop detection and a predictive interface to automate repetitive typing and editing, which can be especially important in minimizing typing in small handheld devices or for users with disabilities. The systems of Masuishi, Masui, and Sugiura have all been distributed to a large user community in Japan. Tessa Lau, Steve Wolfman, Pedro Domingos, and Dan Weld use the time-honored AI technique of version spaces to maintain a space of hypotheses about user actions, illustrating the synergy between work in machine learning and PBE.

Henry Lieberman, Bonnie Nardi, and David Wright also put PBE to work for user convenience, in training text recognition agents to recognize by example common patterns of data that occur in the midst of unstructured information. Their PBE system for developing text recognition grammars, Grammex, was the first interactive interface of any kind to make the powerful grammar and parsing technology accessible to end users. Alan Blackwell adds to this general approach a visual syntax for the grammar rules, which he shows increases user comprehension of the resulting programs.

We shouldn't forget programming environments themselves as a domain for PBE, even if the programming is done in a conventional write-a-file-and-compile-it programming environment. Jean-David Ruvini and Christophe Dony take advantage of the truism that people are creatures of habit. They have a software agent detect habitual patterns in a conventional programming language environment, Smalltalk, and automate those patterns.

Well, if PBE is so great, how come everybody isn't using it? It's our hope that they soon will. But we realize that PBE represents such a radical

S
R
L

departure from what we now know as "programming" that it is inevitably going to take a while before it becomes widespread. Despite the existence of many systems showing the feasibility of PBE in a wide variety of domains, conservatism of the programming community still seems the biggest obstacle.

Signs are growing, however, that PBE might just be beginning to catch on. Commercial PBE environments are beginning to appear, such as the children's PBE environments cited earlier that are now on the market. But it also makes sense to view more conventional user-programming facilities, such as so-called "interface builders," macros, and scripting systems, as the "poor man's programming by example." Some of these facilities are beginning to evolve in directions that may incorporate elements of the PBE approach. We also will need conventional applications to become more "PBE-friendly" so that PBE systems can use the conventional applications as tools in the same way that a user would operate them manually. Gordon Paynter and Ian Witten show how we might be able to leverage scripting language and macro capabilities that are already present or on the way for applications into full-blown PBE systems. This might facilitate an adoption path for PBE.

In programming, as in theater, timing is everything. Much of the work in PBE is involved with demonstrating how to do something, but equally important is when to do it. David Wolber and Brad Myers explore what they call "stimulus-response" PBE, in which we generalize on time and user input, to assure that PBE-programmed procedures are invoked at just the right time. Wolber also compares his PBE animation system to a conventional animation editor/scripting system, Macromind Director, which brings the similarities and differences of PBE versus conventional applications into sharp focus.

We then move on to explore some directions where PBE might be heading in the future. Alexander Repenning and Corinna Perrone-Smith show how we can take PBE a step further, using another important intuitive cognitive mechanism—analogy. We often explain new examples by way of analogy with things we already know, thus allowing us to transfer and reuse old knowledge. Repenning and Perrone-Smith show how we can use analogy mechanisms to edit programs by example as well as create them from scratch.

Robert St. Amant, Luke Zettlemoyer, Richard Potter, and I explore what at first might seem like a crazy approach. We actually have the computer simulate the user's visual system in interpreting images on the screen rather than accessing the underlying data. Though it may seem inefficient, it

neatly sidesteps one of the thorniest problems for PBE: coexistence with conventional applications. The approach enables "visual generalization"—generalizing on how things appear on the screen, as well as properties of the data.

Programming by example is one of the few technologies that holds the potential of breaking down the wall that now separates programmers from users. It can give ordinary users the ability to write programs while still operating in the familiar user interface. Users are now at the mercy of software providers who deliver shrink-wrapped, one-size-fits-all, unmodifiable "applications." With PBE, users could create personalized solutions to one-of-a-kind problems, modifying existing programs and creating new ones, without going through the arcane voodoo that characterizes conventional programming. In this collection of articles, we hope that the diversity of systems presented, compelling user scenarios, and promising directions for the future of PBE will convincingly demonstrate the power and potential of this exciting technology.