The Evolution of End User Programming

For as long as there have been computers to program, there have been attempts to make programming easier, less technical, and available to a broader audience. End User Programming has been an area of academic research for thirty years. So why aren't end users programming? The field has had a few notable successes, several waves of interest, and a variety of quiet failures.

The lack of widespread success is due to a few serious obstacles which have yet to be overcome. We shall see how researchers have tried to overcome these problems in the past, and how, fortunately, the current environment of computer use is dramatically different from previous eras, and has some important new features that may overcome these long-standing obstacles. End User Programming on the Web has the potential for widespread success that can change the way people use computers.

The term "end user programming" proposes that although computer users do not know how to program, they would appreciate having some of the power of programming if only it could be obtained with little effort. Back in the 1960's, *using* a computer meant *programming* a computer. There was no need for the concept of "end user programming" because all end users were programmers. The 80's were a time of transition: I had a friend who—in 1980—wrote her Comparative Literature thesis on punch cards. Then the Macintosh came out in 1984, and soon computers meant “desktop computers”, command languages were replaced by direct manipulation, and end users strove for computer literacy. End users were no longer programmers; “literacy” meant knowing how to point-and-click in a word processor or spreadsheet. Now, in the new millennium, children can use a mouse before they can talk and they acquire 'literacy' skills by picking up a new application and using it. The generation where literacy was taught and computers were intimidating is now retiring from the work force.

End User Programming researchers point to spreadsheets as their big success. There are now millions of non-programmers who write spreadsheet commands. While this is true, it is an unsatisfying success for the field. Contemporary computer use has moved beyond word processing and spreadsheets; it involves interpersonal communication, web browsing, and internet shopping. Moving beyond the desktop, it includes cell phones and pdas, with text messaging, user-generated content on youtube, and traffic conditions on Google maps. Furthermore, true success will come not when the need for end user programming

is so great that users are compelled to learn a complex command syntax, but when programming becomes easy enough and natural enough that end users see it as a welcome opportunity that is both useful and enjoyable.

What's hard about regular programming languages is that the text you have to write to get something done is very remote from what is being done. To click on a button, you have to write something like this:

```
theEvent.initMouseEvent("mousedown", true, true, contentWindow,
1, (contentWindow.screenX + clickLocH), (contentWindow.screenY +
clickLocV), clickLocH, clickLocV, ctrlP, false, false, false, 0,
null)
```

The syntax is obscure and unforgiving, and many of the details are abstract and only indirectly related to the simple action being performed. In short, traditional programming languages are *obscure, abstract,* and *indirect*.

End User Programming (EUP) systems are able to simplify programming by addressing a limited domain and offering limited power. There have been two main approaches: Scripting Languages and Programming by Demonstration (PbD).

Scripting Languages

Scripting languages are special-purpose languages designed to handle a specific domain, such as spreadsheets or email or photo editing. While a programming language needs to be able to express arbitrary commands like the `initMouseEvent` command above, a scripting language may only need to express a few actions, such as `send` and `delete`, and a few objects, such as `messages` and `attachments`. So instead of the myriad possibilities of commands using the obscure format of the `initMouseEvent` command, a scripting language can use simplified formats like `send the message`.

Scripting languages walk a fine line between power and ease of use. The formulas used in spreadsheets have opted for considerable power, at the expense of requiring significant effort to learn. This design choice has been quite successful, in part because spreadsheets are used to perform numerical calculations, so their users are already resigned to working with mathematical formulas. While it's true that it is easier to learn to write `=SUM[A1:A12]` than to create commands like the `initMouseEvent` shown above, this is still a far cry from the simplicity required to turn most computer users into end user programmers. One approach that has helped make scripting languages easier to use is the "structure editor": the end user creates commands by selecting words from menus, and the editor guarantees that only legal combinations of words can be selected.

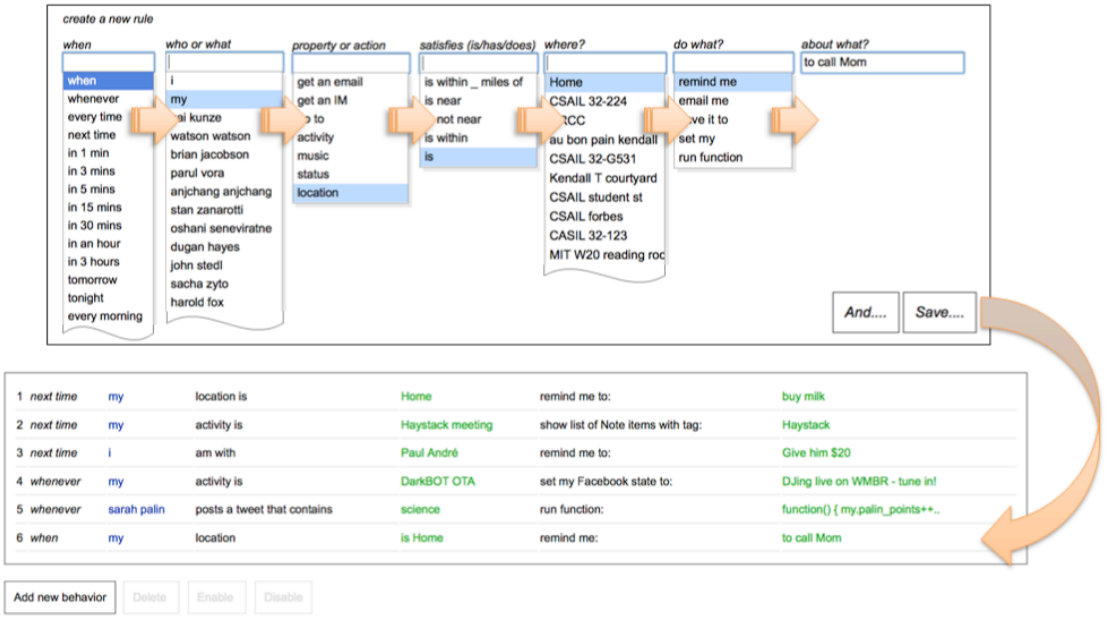

Figure 1. AtomsMasher's combined natural language and structure editor.

A big change that has come to end user programming in the last several years is the introduction of natural language systems. This is a great leap in ease of use that discards the need for a rigid syntax and allows the end user to express commands in English, or some other natural language. Five of the systems presented in this book use natural language, and the AtomsMasher system combines natural language with a structure editor (see Figure 1).

Programming by Demonstration

The end user of a Programming by Demonstration (PbD) system demonstrates an activity, and the system writes a program to perform that activity. This simplifies program creation in two ways: First, it eliminates *indirectness*, since the user interacts directly with an application by clicking its buttons, typing into its boxes, and dragging its objects. There is no need to know that the button that was clicked is the `dijit_form_ComboBox_0.downArrowNode`. Second, it eliminates the problems of an *obscure syntax*, since the system writes the commands for the user.

The classic challenges that must be addressed in creating programs from user demonstrations are 1) how to infer the user's intent, 2) how to present the created programs to the user, and 3) how to deal with special cases and messy, real-word data. In addition, a practical obstacle that has greatly limited the success of PbD systems has been the multitude of incompatible applications and computer operating systems, and the absence of scriptability and recordability in those

applications. We will see how these issues have been addressed in the past, and how present circumstances offer significant new opportunities for progress.

Inferring intent

I have a list of addresses that I want to add to my online Address Book (see Figure 2). After adding a few by hand, I wish I had a program that would finish this activity for me. In 1988, Witten and Mo [ref xx] created a PbD system that could automate this kind of activity.

```
John Bix, 2416 22 St., N.W., Calgary, T2M 3Y7. 284-4983
Tom Bryce, Suite 1, 2741 Banff Blvd., N.W., Calgary, T2L lj4. 229-4567
Brent Little, 2429 Cheroka Dr., N.W., Calgary, T2L 2j6. 289-5678
Mike Hermann, 3604 Caritre Street, N.W., Calgary, T2M 3X7. 2340001
Helen Binnie, 2416 22 St., Vancouver, E2D R4T. (405)220-6578
Mark Willianms, 456 45Ave., S.E., London, F6E Y3R, (678)234-9876
Gorden Scott, Apt. 201, 3023 Blakiston Dr., N.W., Calgary, T2L lL7. 289-8880
Phil Gee, 1124 Brentwood Dr., N.W., Calgary, T2L lL4. 286-7680
```

Figure 2. A list of unformatted addresses.

Semantics

Ideally, one would teach a PbD system just as one would teach another person: you would select *John*, and say "copy the first name and paste it into the *First Name* box in the Address Book". Actually, with a human assistant, you would just say "copy this information into the Address Book". Both approaches rely on the fact that a human understands the semantic concepts of people's names, addresses and phone numbers, and has enough experience with them to be able to identify those items in the text.

The main reason this task is difficult for a PbD system is that the system doesn't understand this real-world semantic knowledge. This problem is actually not unique to end user programming: it is a fundamental challenge behind all computer programming. A professional programmer who wants to make a program that will take postal address information from a page of text and use it to fill in an address form on a web page has to deal with exactly the same problem: how do you write a computer program that will figure out which part of the text is the first name, the last name, the street number, the street address, and so on?

Witten and Mo's system did what a programmer might do: it looked for patterns in the *syntax*—such as *a series of digits followed by a space followed by a series of letters and then a space followed by the letters "Dr.," or "Rd.," or "Ave.,"*—that corresponded to the *semantics*. When a user selects *Bix* in this example, the system

can make many inferences about why that word was selected: because it is the second word, the first three-letter word, the first word that is followed by a comma and a space, or perhaps the second capitalized word. Or, if the system had semantic information available, it might infer that the user was selecting a person's Last Name or the first word after a First Name. Deciding on the appropriate interpretation is termed *inferring intent* in a PbD system, and the correct inference is often a matter of semantics.

What's new and ground-breaking in the age of the Internet is that 1) large-scale semantic information is being collected by search engines and in knowledge bases like ConceptNet [ref xx], 2) programmers are writing detailed programs called *data detector*s [ref xx] to recognize semi-structured information like addresses, 3) web sites are formatting this information on their pages with microformats [ref xx], and, most importantly, 4) this information and these programs are readily available, free of charge, and are being continually updated. As a result, one of the major barriers to successful End User Programming systems is coming down. It is now becoming possible for PbD systems to circumvent the entire problem confronted in Witten and Mo's example by simply utilizing data detectors for names and addresses. The Citrine system, for instance, can take a line of text like that in Figure 1 and—in a single action—paste the appropriate parts into the various fields of a web form (see Figure 3). There will still be plenty of idiosyncratic tasks for PbD systems to automate, but now, those systems won't be annoyingly "stupid" because they don't have access to basic semantic concepts.

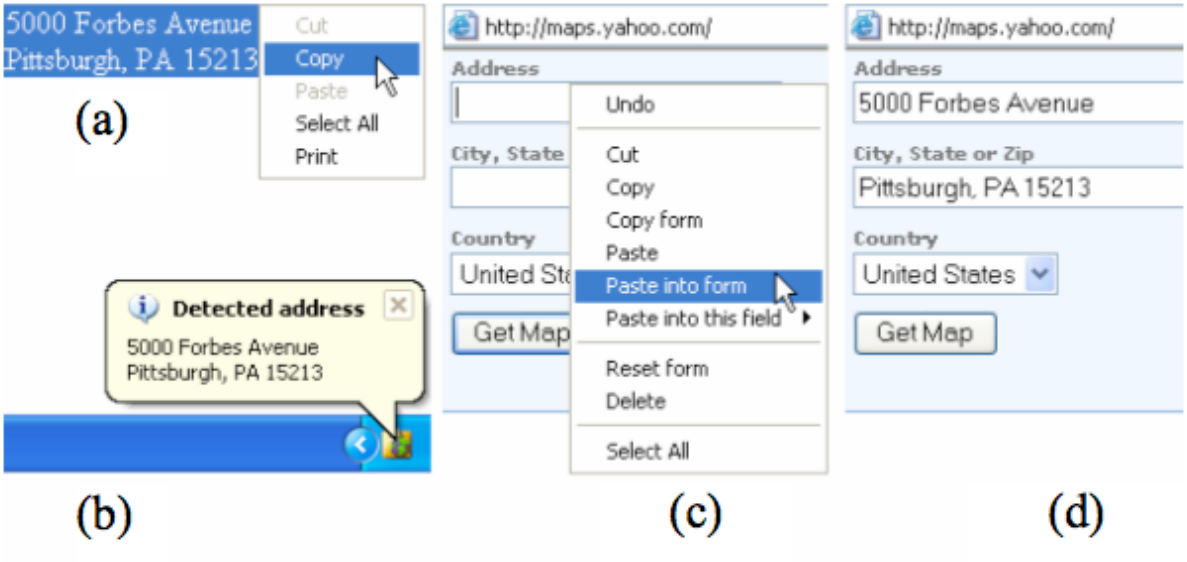

Figure 3. Citrine's address detector.

Choosing the right abstraction

Semantics account for many of the inferences that PbD systems need to make, but there are plenty more that are not a matter of semantics. For example, consider the possible reasons why I might click on a link on a web page. In order to do the right thing the next time I run my PbD-generated program on that web site, it's important to make the correct inference. When I visit my banking web site, I always look at the charges for the second date in the list, since that is the most recently completed bill (see Figure 4 (a)). Inferring from my demonstration that I always want to pick *February 3, 2009* would be wrong. On the other hand, when I check the current traffic on *cbs5.com*, I always pick the information for *South Bay*. Sometimes this is the fourth item in the list, sometimes it's the sixth, and sometimes it doesn't appear at all (see Figure 4 (b)).

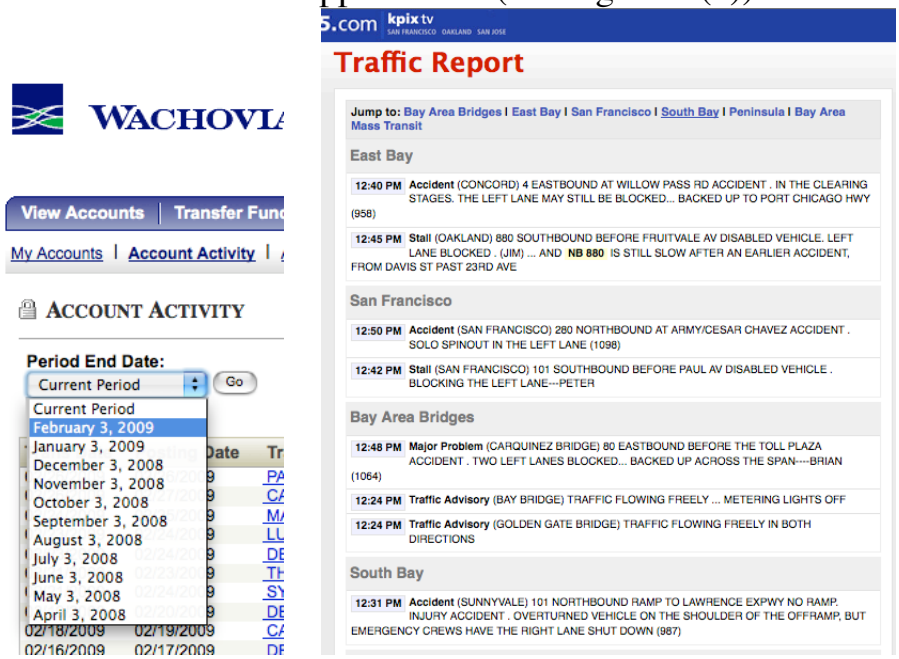

Figure 4. The correct inference for the item in (a) is *the second item*, while the correct inference for the item in (b) is *the "South Bay" item*.

Presenting programs to the user

As the examples in Figure 4 show, a PbD system can't always make the correct inference. Instead, the best that PbD systems can do is to generate the "reasonable" alternatives and let the user pick the right one. As Alan Kay has said, "When a human is using a computer, there is one intelligence there." [ref xx] A PbD system is nonetheless a great help to an end user, since *recognizing* the command for the correct inference is much easier than *writing* that command yourself.

In order for users to choose an interpretation, PbD systems need to be able to *present* their inferences to the user. *Presenting programs to the user* is therefore an important part of a PbD system. Witten and Mo's TELS system was able to generate fairly sophisticated programs for automating users' tasks, but it had no means of presenting those programs to the user. In addition to allowing users to select a correct inference, presenting programs is also important for establishing a user's trust in a program. When users do not really know what a program is going to do, they will be wary of running it. A third benefit of presenting programs to the user is that it enables users to correct and improve their programs. My Stagecast system [ref xx] presented programs visually, showing "before-and-after" pictures of what a command would do. Figure 5 (a) shows a command to make a train move forward along a track. A comparison with an equivalent scripting language program for the same train (see Figure 5 (b)) shows the potential improvement in ease of use that PbD can afford.

```
put iconName(icon of cd btn LookAhead) into BtnIconName
if the number of items in BtnIconName > 1 then
  put "True" into Staging
  if TheStage = 0 then put BtnIconName into PrevBtnIconName
  if BtnIconName contains "roadXing" then put LookAhead into XLoc
  if BtnIconName contains "Rotatetrain" then put 1 into TheStage
end if
if the mouseClick then checkOnThings the clickLoc
put LastLoc & return before PrevLocs
put LookAhead into LastLoc
put Dir & return before PrevDir
if the mouseClick then checkOnThings the clickLoc
add 1 to Counter
```

Figure 5. Instructions to move a train in (a) Stagecast and (b) HyperTalk.

Despite some successes, the problem of presenting abstractions to the user is still challenging for PbD systems. For instance, how can a system like TELS reasonably express to the user that it is locating a street address by searching for *a series of digits followed by a space followed by a series of letters and then a space followed by the letters "Dr.," or "Rd.," or "Ave.,"*?

Dealing with messy data

Once the user has demonstrated to TELS how to handle the first line in Figure 2, it would be great if TELS would process the rest of the lines automatically. However, data in the real world never meets our idealized expectations. For instance, line 7 is the first time an apartment number appears, and line 5 is the first example that uses an area code. A very successful approach to handling the inevitable exceptions that arise in practice is for the end-user's program to pause whenever it encounters a new situation and to let the human handle the special case. Termed *mixed-initiative*

interaction, this approach is a luxury that professional programs cannot afford, since they are expected to be robust and capable of handling whatever situations may arise. But in the domain of end user programming, where most activities are ad-hoc and the end user just wants assistance with what would otherwise be an unbearably tedious process, sharing a task between the human and the computer is appropriate and practical.

A newer technique that advances the practical use of PbD systems is called *simultaneous editing*. The Potluck system [ref xx] uses this technique to intelligently cluster messy data so that the end user can demonstrate how to handle each special case quickly and efficiently (see Figure 6).

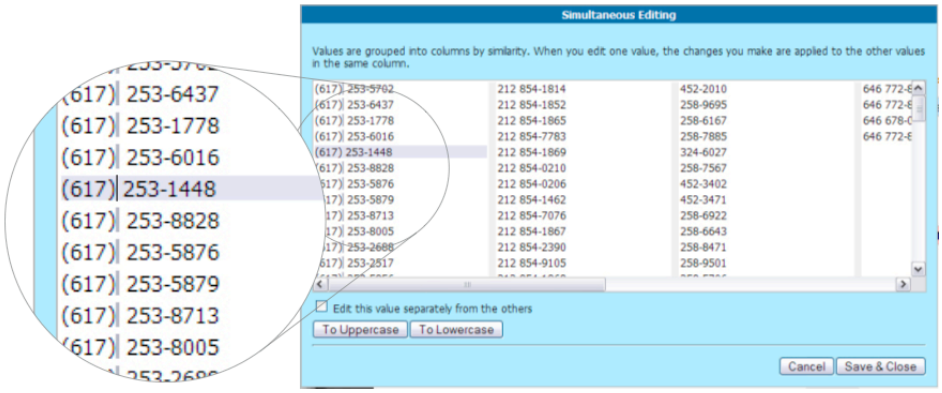

Figure 6. Simultaneous editing in Potluck.

The multi-platform barrier

The TELS system was not a plug-in that could be added to MacWrite or WordPerfect or any other text editor that was popular at the time. To test their ideas about PbD, Witten and Mo had to write their own custom text editor. All of the early work on PbD was done at a time when desktop applications were distributed as source code that could not be modified, and user actions in the applications could not be recorded or even scripted. Regardless of the usefulness of a PbD system, no one would be able to use it in their daily work.

We are in a very different world today. The popularity of the web means that many different kinds of applications, such as word processors, email and chat programs, as well as online banking and retail shopping, are all implemented on a single platform – the web browser. And thanks to the open source movement, the Firefox browser is available, complete with a simple means for adding custom extensions. For the first time, PbD systems can be added to a real platform that millions of people use in their daily lives. You will see that ten of the sixteen End User Programming systems described in this book are written as Firefox extensions. So

perhaps the greatest barrier of all to the widespread success of End User Programming has fallen.

Another tremendous advantage of the web platform for end user programming is that it is *declarative*. Web pages are written in HTML, which means that all items on the page have a *semantically meaningful* tag, identifying them as buttons, textboxes, and pull-down menus. This immediately solves the problem of *inferring semantics* that was discussed earlier.

The (near) future of End User Programming

The simplicity of HTML is credited as one of the reasons for the web's overwhelming adoption and success. However, along with being simple, HTML is also impoverished. This boon for End User Programming was at the same time a great leap backwards in user interface design and functionality. We lost the ability to drag and drop, to precisely arrange page layout, to draw anywhere on the page, and to update a small part of a page. Nuanced interactions were replaced with jumping from page to page.

It was not long before the simplicity of HTML was augmented with new techniques that bring back the richness of interaction that is possible in desktop applications. And these new techniques are posing a challenge for the future success of End User Programming. Flash, for instance, allows for rich user interactions. But no HTML appears on the part of a web page that uses Flash. When users click and type in Flash, Programming by Demonstration systems get no indication at all that anything has happened. Similarly, the use of javascript, the AJAX programming style, and web toolkits like YUI and Dojo are replacing the *declarative* format of HTML with *procedures*, or programs. The buttons and textboxes in Dojo all use the semantically meaningless DIV tag, and the only way to understand the semantics of a javascript procedure is to read the program.

Fortunately, the problem posed by toolkits may also afford its solution. Toolkits enable website developers to use semantically rich user interface objects without having to build them by hand. This means that if just one person goes to the trouble of documenting the meaning of the items in a toolkit, then that information is available for every website that uses the toolkit. It is also fortunate that the need for website accessibility—for blind users in particular—is a strong motivation for adding just this sort of semantic annotation to a toolkit. The ARIA specification [ref xx] is a standard for adding semantic annotations to toolkits, Ajax, and javascript. Further, there is a Social Accessibility [ref xx] project that can make these semantic annotations available everywhere on the web.

Future domains

Command line interfaces gave way to desktop applications and then web pages. Web pages are beginning to be replaced by web applications, and the next domain for innovative applications will be mobile devices. Mobile devices need the power of customization offered by End User Programming: they have small screens, so it's important that only relevant information be displayed; without a full keyboard, user input is difficult and constrained, so it is important that users can express their specific needs with just a click or two; and since they are used while people are on the move and their attention is limited, there is an even greater need for simple displays and interaction. If the opportunities of the web can break the barriers that have been limiting End User Programming, a new generation of end user programmers can flourish in the coming age of mobile devices.