An Exploration of Design Opportunities for "Gardening" End-User Programmers' Ideas

Jill Cao, Scott D. Fleming, Margaret Burnett
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, Oregon, U.S.A.
{caoch, sdf, burnett}@eecs.oregonstate.edu

***Abstract*—Despite recent advances in supporting end-user programmers, empirical studies continue to report barriers that end users experience in problem solving with programming environments. We hypothesize that an important barrier that still needs to be overcome is the lack of support for nurturing end-user programmers' *ideas* on how a program should be written or on how to solve programming difficulties. Therefore, in this paper, we present a qualitative empirical investigation and triangulate the results with theories from problem solving and creativity. Moreover, we explore design opportunities and a design space for "idea gardening", a new approach to nurturing end-user programmers' ideas and to helping them gradually gain expertise as they overcome barriers. Our results suggest that nurturing end-user programmers' ideas is a fertile area for research with an interesting, multidimensional design space.**

Keywords—end-user programming; mashups; problem solving; creativity; end-user software engineering.

I. INTRODUCTION

Over the decades, researchers have made remarkable strides in bringing programming capabilities to ordinary end users (e.g., [2, 11, 16, 20, 25]). Today, there are numerous programming environments for end-user programmers in both research and practice, with spreadsheets and database systems being arguably the most widespread examples. End-user programming has become so widespread that, according to the U.S. Bureau of Labor and Statistics, by 2012 the number of people using spreadsheets and databases at work will reach 55 million—an order of magnitude greater than the number of professional programmers [29].

Approaches to end-user programming have been particularly successful in overcoming three barriers: arcane language constructs that are difficult to learn, syntax rules that are difficult to memorize, and the difficulty of finding suitable example programs that other end users have written. One approach, programming by demonstration [20], removes both the use of arcane language constructs and the need to memorize syntax, as do many other kinds of visual programming approaches. The Natural Programming methodology re-envisions language constructs [25], as have spreadsheets. Many modern environments for end users support repositories of example programs (e.g., AgentSheets [27] and BluePrint [4]). Mashup environments' whole reason for existence is to enable reuse of others' programs.

Despite this progress, however, empirical results continue to report that programming remains difficult for end users (e.g., [5, 6, 11, 17]). Perhaps one reason is that much of the past work has focused on helping users to avert or solve lower-level problems, but these kinds of help are not enough. The challenges in programming also call for problem-solving skills, creativity, and design thinking, and our prior empirical investigations [5, 6] suggest that end-user programming environments generally lack mechanisms to nurture these kinds of skills.

In this paper, we therefore explore design opportunities for a new concept we term idea "gardening", which means helping (end-user) programmers to initiate and develop ideas *themselves* for programming solutions or for strategies for arriving at programming solutions.

This paper makes the following contributions:

(1) A demonstration of how to apply theories from problem-solving literature and creativity literature to recognize problems in end-user programmers' problem-solving processes.

(2) The first empirical results of end-user mashup programming from the perspective of end-user programmers' problem-solving processes.

(3) A demonstration of how to apply problem-solving and creativity theories to design solutions for helping end-user programmers initiate and refine *their own* ideas, first with an abstract solution and then with a rendition of the same solution in CoScripter [21].

(4) A multidimensional design space for research into idea "gardening".

II. RELATED WORK

As we have already pointed out, research in end-user programming has developed programming approaches to lower the barriers end users face. Outside of work on tutorials and on-line help systems, these approaches generally aim at the design of the language, example programs, or programming process.

Languages like AgentSheet [27] and CoScripter [21] exemplify the programming-by-demonstration technique, which allows users to demonstrate an activity that is stored as a program by the system. Some other visual programming languages for end users also use direct manipulation or even tangible manipulation to ease the cognitive burden on users to memorize syntax (e.g., Alice [16] and AutoHAN [2]). Natural

978-1-4577-1247-0/11/$26.00 ©2011 IEEE

programming is a language design paradigm that reduces language learning barriers by leveraging users' natural communication vocabulary and style, as in the Hands language for children [25]. Table data structures akin to spreadsheets are sometimes used to remove the need for iteration language constructs (e.g., Vegemite [21]). These kinds of approaches all aim to eliminate the need to learn arcane language constructs and/or to memorize syntax.

Examples are another form of support for enabling end users to overcome barriers. Gross and Kelleher investigated the strategies end users adopted in locating functionalities in unfamiliar code to inform the design of an interface aimed at helping users decide what example code to reuse [11]. FireCrystal allows a programmer to select UI elements of a web page to view the corresponding source code to facilitate learning by reusing that web page as a life example [23]. BluePrint automatically gleans task-specific example programs and related information from the Web [4]. HelpMeOut aids novice programmers' debugging of error messages by recommending example solutions previously used by their peers [12].

From a programming process perspective, some researchers have essentially "refactored" the responsibility of designing and programming. For example, meta-design pairs end-user programmers with professional programmers, to reduce the cognitive burden on end users by having them perform only part of the programming process (e.g., [9]).

However, although these approaches lower cognitive burden in one way or another, they do not attempt to nurture end users' program problem-solving *ideas*. The only work we have found of this nature aims to support professional designers. For instance, DENIM is a system that allows designers to sketch web sites at a high level [22]. Diaz et al. created a visual language to help web designers identify suitable design patterns [8]. We believe these kinds of approaches may be useful to a much wider range of audiences than just professional designers, and works of this type have helped to inspire this paper.

III. FORMATIVE INVESTIGATION

To begin our exploration, we conducted an empirical investigation to understand concrete instances of the "idea barriers" end-user programmers encountered in the context of mashup environments. Toward this end, we conducted a qualitative study in CoScripter. We also performed a new analysis of data obtained in an earlier study conducted in a different mashup environment, Microsoft Popfly.

CoScripter is an end-user programming-by-demonstration environment for web scripting and mashup building [21]. In CoScripter (Fig. 1), users demonstrate to the system how they would carry out a task on the Web (e.g., by filling out a form online to reserve a shuttle ticket to the airport). The system watches and translates users' actions into an editable script (Fig. 1-1), which the user can execute at a later time to perform the same task again.

CoScripter enables mashup programming via a table feature (Fig. 1-2). Users can create scripts that automatically copy data between the table and existing web pages. This enables the user to combine data from multiple web pages in the table and to flow text from one web page to form fields of another.

We conducted the CoScripter study with six university students (three males, three females) from a variety of majors (e.g., graphic design, accounting, wood science). The participants had little or no programming experience. We conducted the study one participant at a time using the think-aloud method. Participants first filled out a background questionnaire and completed a 20-minute tutorial that familiarized them with CoScripter. They then completed a self-efficacy questionnaire [7] adapted to the task of scripting. Participants then practiced "thinking aloud" before proceeding to the main task.

The participants' task was to create a mashup script to automatically search for two-bedroom apartments that rent for under $800 per month and are within a 30-minute drive of the Oregon State University campus. This required combining data from a search site, such as Craigslist or Apartments.com, with data from a maps site, such as Google Maps.

Participants had 50 minutes to complete this task. If they were unable to make progress for at least three minutes, the researcher prompted them with hints, such as suggesting that they try a different website or try using the table. The purpose of the hints was to maximize the data we were able to collect. The hints enabled the participants to regain progress so that we could go on to observe problems they encountered further along in the task. All participants received hints. Due in part to these hints, all participants finished the task. We collected videos of the participants as they worked.

To increase the generality of our results, we also analyzed videotapes from a qualitative study of Popfly that we conducted about 18 months ago [6]. As with the CoScripter study, we conducted the Popfly study with college students (six males, four females) with little or no prior programming experience. As in the CoScripter study, we used a think-aloud design and started with a tutorial. The task was to create a mashup that integrated movie-related information such as which films were showing at local theaters and news stories about each film. We videotaped the participants while they worked. Details of the Popfly study procedures can be found in [6]. No participants received hints. Participants had varying degrees of success on the task, with one person fully achieving all requirements.

Unlike CoScripter's programming-by-demonstration paradigm, Popfly is a visual dataflow language. In Popfly, users build mashups using programming constructs called *blocks*. Users can choose from existing blocks, each of which performs a set of operations such as data retrieval and data display. A block's operations may take input parameters. Users may connect blocks to form a network in which blocks can use output from their adjacent blocks as inputs. Fig. 2 shows a mashup example in which the Flickr block sends a list of images about "beaches" with their geographical coordinates to the Virtual Earth block (Fig. 2: top and middle) to display them on a map (Fig. 2: bottom).

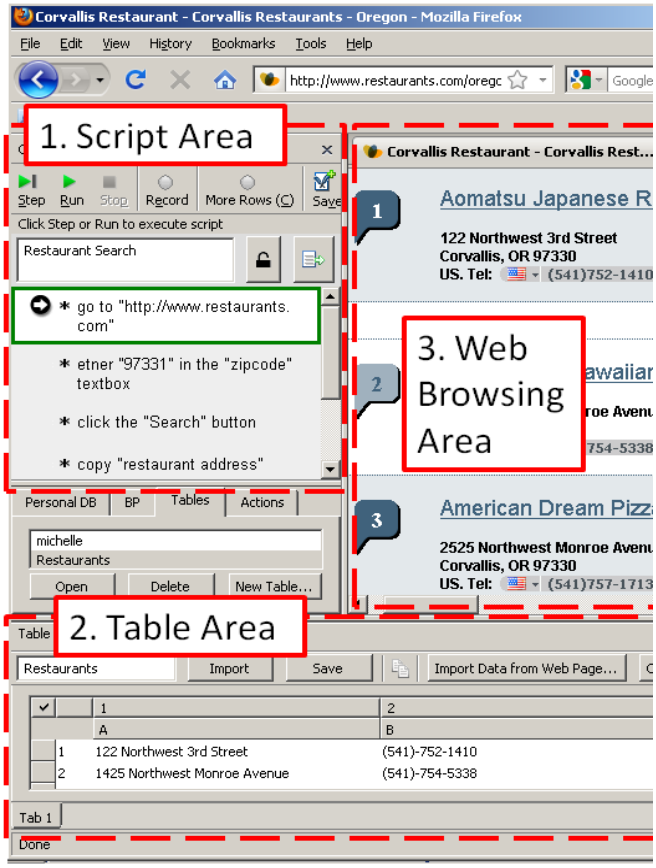

Figure 1. The CoScripter environment with three main parts, i.e., the Script Area, the Table Area and the Web Browsing Area.

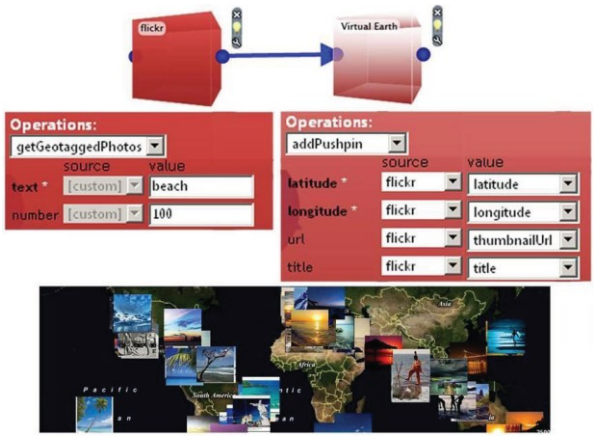

Figure 2. The pre-task tutorial example mashup in Popfly Mashup Creator. Top: the blocks. Middle: some of the blocks' settings. Bottom: results generated by pressing the Run button (not shown).

IV. Results and Implications

According to Simon, two types of skills are necessary for solving problems in a specific domain: domain-specific knowledge and general problem-solving strategies [32]. He equates domain-specific knowledge and general problem-solving strategies to the twin blades of a pair of scissors: "the scissors do indeed have two blades and … effective professional education calls for attention to both subject-matter knowledge and general skills" [32]. Bloom and Broder agreed, and showed that both mathematical domain skills (e.g., how to multiply integers) and general problem-solving strategies (e.g., establishing subgoals of a problem) are indispensible to a successful math problem-solver [3].

Thus, we present our results, organizing along these two skill sets, and triangulating with applicable theories related to problem solving.

A. Problem-Solving Strategies

1) Results in Problem-Solving Strategies

According to the literature on problem solving (e.g., [15]), the adoption of problem-solving strategies are affected by metacognitive skills, beliefs, and expertise. (A *strategy*, according to Webster Dictionary, is a careful plan or method for achieving a specific goal.) We discuss the first two of these here, and since expertise maps to Simon's concept of domain-specific knowledge, we discuss it in the next subsection (IV.B).

Metacognition is described by Flavell as the awareness of how one learns, the monitoring of understanding, the use of information to achieve a goal, and the assessment of learning progress [10]. Wickelgren proposes that when stuck on a problem, it is important for a problem solver to step back and analyze what he/she has been doing (a reflection on one's approach and a metacognition skill), rather than to keep thinking about the problem itself [34].

Some participants exhibited very little metacognition about solving the problems they ran into. For example, CoScripter-F2 did not show signs of metacognition about her problem-solving strategies. In the example below, she failed to step back to reflect on her strategy when unable to make progress. As a result, the researcher had to prompt her with hints to help her make progress.

> *CoScripter-F2: [Renames her script] "I don't know how to do it. Once I've done that [renaming script], I don't know what else to do."*

Participant Popfly-M3 likewise exhibited little use of metacognition. His main strategy of overcoming problems he ran into was to "try a different block" whenever the mashup stopped working, never reflecting on whether this strategy was a wise way of going about his problem solving.

> *Popfly-M3: [Mashup shows nothing] "So I try a different one maybe." … "Try a different one that I know how to use 'cause none of them worked yet or I can get to work."[Tries the Image Scraper block. Still does not work]*

On the contrary, when participants did reflect on their problem-solving approaches, doing so often helped. For example, Popfly-M5 made a breakthrough in his problem solving after changing his strategy to the use of incremental changes and testing:

> *Popfly-M5: "Simplicity" [Runs. Theater and movie info show]*
> *"Oh, ok. There we go. I was getting way too complicated."*
> *"It works well to run the program at each step."*

As to beliefs and attitudes, one kind of belief that can affect the adoption of problem-solving strategies is self-efficacy, a person's confidence in their ability to succeed at a specific kind of task. According to self-efficacy theory [1], people with low self-efficacy tend to be less flexible in their problem-solving strategies than those with high self-efficacy; for example, a low self-efficacy person may stay with a known approach even if when it is not paying off.

In both studies, low self-efficacy participants indeed demonstrated inflexibility. For example, CoScripter-F3 had the lowest self-efficacy in the CoScripter study (3.4 vs. an average of 3.77 for all participants), and she did not consider switching from a straight Google search to using the table to help with her task until the researcher prompted her.

CoScripter-F3: [after several trials with Google searches] "... I don't know how to say how far from OSU." [continues to ponder the search screen]
Researcher prompts with a question:"If you were to find out how far an apartment is from OSU, what would you do normally?"
CoScritper-F3:"I'd go to Google Map or something, if I had an address and I wanted to know how far it was… Oh [in the tutorial] you showed me how to do that using the table!"

2) Design Opportunities for Supporting Problem-Solving Strategies

Bloom and Broder asserted that the "habits of problem solving, like other habits, could be altered by appropriate training and practice" [3]. This leads to the possibility that suggesting an appropriate problem-solving strategy at moments of difficulty could nudge end-user programmers' program problem-solving skills up enough to enable them to form new ideas themselves.

For designers of programming environments to act upon such a possibility, a list of well-studied problem-solving strategies is needed. Therefore, we compiled a list of strategies from the literature on problem-solving [19, 26, 34] and creativity [24]. In particular, we focused on the most commonly discussed strategies from the literature (i.e., the strategies cited in multiple sources). We selected a spectrum of these strategies for breadth of situation coverage, leading to the following list of five strategies.

Working Backward is a strategy in which the problem solver starts with the goals of the problem in an attempt to work his/her way back to the givens of the problem as opposed to starting with the givens [34]. Polya argues that Working Backward is a common-sense procedure within the reach of everybody [26]. One way to bring this strategy to end-user programmers might be to allow them to start by envisioning the output of their program so that they can work backward from there to arrive at what might lead to the output.

With *Divide and Conquer*, the problem solver breaks the original problem into subparts, and works out each part individually to arrive at the solution of the original problem [33, 34]. In end-user programming environments, encouraging an end-user programmer to tackle some small part of the problem may not only provide momentum and insights on a potential overall solution, but also may increase less confident users' self-efficacy levels, with the follow-on potential of positive effects on their problem-solving skills as explained above.

Analogy encourages users to relate the problem at hand to problems they have seen or solved in the past [24, 26]. Polya mentions two ways a problem solver may leverage a solved problem in solving an unsolved problem: (1) use the solved problem's results, and (2) use the method for solving the solved problem in solving the unsolved problem [26]. One reasonably straightforward method for encouraging this strategy in end-user programming environments might be to store a history of a user's previously solved problems, and (more interestingly) to recognize similarities of an emerging solution with previous solutions in the history. Another more challenging possibility might be to log and catalog the user's previous *methods* to successfully solve a previous problem, making those methods available to the user at opportune moments.

Generalization is defined by Polya as passing from the consideration of one object to the consideration of a set containing that object. (Trained computer scientists will recognize induction/recursion as useful examples of Generalization.) Generalization can be helpful to end users in that it allows the user to start with a single, concrete case, which may be more tractable than considering the general case all at once. Leading an end-user programmer in this direction may lead to the same kinds of benefits as Divide and Conquer. For instance, programming by demonstration capitalizes on end users' familiarity with the concrete instance of an activity to help them produce an abstraction of that instance in their program.

Finally, with *Sleep on It,* a problem solver sets aside a difficult problem and comes back to it later, possibly with a fresh perspective [19, 24, 26]. Bringing this strategy to end-user programmers may be as straightforward as encouraging a stymied user to simply set aside the difficult subproblem, and instead attend to parts of the problem that are more approachable.

Note that all of the above strategy opportunities are about what a designer of an end-user programming environment might encourage an end-user programmer to do, but do not specify how the designer should go about offering the encouragement. Clippy-style pop-up interruptions have not been seen to be suitable for the kind of problem solving we are considering here, and a more subtle form of encouragement such as Surprise-Explain-Reward [35] with negotiated style interruptions is likely to be more suitable [28].

B. Programming Domain Knowledge

1) Results

In one way, end users are by definition domain experts. The advantage of end-user programming is, in fact, to bring their expertise of the problem domain directly to the program, without the need for intermediary professional programmers.

However, the other relevant domain here is programming itself, and it has been reported that many end-user programmers lack expertise in the programming domain (e.g., how to go about debugging), or in the language the user is trying to use [18].

Because the languages we studied, CoScripter and Popfly, were created especially for end users, one might not expect issues of programming expertise to arise. Interestingly, however, the expertise factor arose many times and at multiple levels. At the language construct level, CoScripter-M1 had trouble figuring out where the "repeat" command should go when he wanted his script to loop through the rows in his table, a skill learned early by successful students of computer science.

CoScripter-M1: "So I got my results [in the table]. I guess you can repeat it then."
[Adds "repeat" to the beginning of his script, which tells the script to repeat every line instead of just table computations]

At a more "design pattern" level, Popfly-F3 did not see a connection between the overall task she was trying to accomplish and the availability of a "library" of components/blocks that had been demonstrated to perform portions of the task, whereas computer science students learn early to use libraries/APIs to accomplish parts of a problem. Without recognizing the availability of component parts for her solution, she did not see how to even get started:

Popfly-F3: Oh, my gosh! This is very hard! Can you give me some reminders [hints]?

Difficulties like these and others—such as a poor mental model of the programming-by-demonstration concept and lack of understanding of the concept of inputs and outputs—played out in three ways: lack of a sense for how to get started (CoScripter-F2 below), running out of ideas to try very early (CoScripter-M2 below), or stubbornly staying with the same idea for a long time without considering other possibilities (Popfly-F4 below).

CoScripter-F2: "I don't know what to do…"
Researcher asks her to "show" the computer what she wants the script to do.
CoScripter-F2: "Umm?…" [Still does not know what to do.]

CoScripter-M2: [Enters search term: "2 bedroom apartment Corvallis OR". Clicks the "Search" button.]
[Tries a few search results, e.g., www.mynewplace.com]
"Those don't seem to work. I'm stuck."

Popfly-F4: "Oh there's no push pins [on the map]! These push pins are gonna haunt my nightmares…Why does that not work? Seems like it'd work but it doesn't work."
[Continues to try to get her idea to work without progress]

The above three examples have in common a scarcity of ideas that would be available to those with more expertise in the domain of programming. The notion of *ideational fluency* from the creativity literature suggests that scarcity of ideas is a problem-solving disadvantage, and that the more ideas one has, the greater chances of him/her arriving at useful and creative solutions to a given problem [24].

2) Design Opportunities for Supporting Programming Domain Knowledge

The scarcity of ideas that seems to be at the heart of many of the programming domain problems suggests specific design opportunities.

Relating to the first example (CoScripter-F2 above), when a user has no idea to start with, there is a clear opportunity to help them gain momentum with *Starter Ideas*. (This is a concept similar to Fischer's "seeding" [9].) The possibilities for such ideas could include strategy ideas (e.g., "try starting with a sketch of the output" or "look at all the blocks that produce maps") or very specific ideas (e.g., "a lot of people use the Google map page in problems involving addresses").

Now consider CoScripter-M2 and Popfly-F4, who had ideas, but ran into trouble and did not know how to move on. Creativity literature suggests that producing new ideas depends upon "mixing" ideas, often catalyzed by associations [24]. Osborn pointed to three "laws of association": contiguity, similarity, and contrast [24].

Osborn's points suggest two more design possibilities for nurturing end-user programmers' ideas: connecting users to ideas *similar* or *in contrast* to those being tried. For example, a similar idea to using a PhotoFlip block for pictures in Popfly is to use the PhotoCarousel block. Offering similar ideas may encourage the user to take into account options other than the ones they already have.

An example of contrasting ideas that a support system could suggest would be using a table block instead of using a photo block to display pictures, the results of which would be quite different. A drastically different idea would be to leave out the display block entirely to see what happens.

Contrasting ideas may encourage users to think outside the box. In creative design literature, "design fixation" [14] refers to the undesirable situation where the designer becomes overly focused on one idea, missing out on other opportunities. Indeed, in [6], we found some users reluctant to relinquish ideas that were not working for them. Design fixation hindered users' ability to redefine the design problem, termed "reframing" in design literature, a critical step in design problem solving [31]. This fixation also resulted in "over-elaboration" [6], i.e., continually elaborating on an idea that cannot ever work. Contrasting ideas may help to avert some of these problems.

V. Example: A Slice of the Solution Design Space

To consider how these design opportunities might be acted upon, we drew on a problem experienced by five of our participants (over both studies). Using this problem, we present one possible solution based upon the previous section's design opportunities, first in abstract form, and then show how it might be concretely instantiated in CoScripter.

A. An Observed Problem

The following issue from our formative study combines two of the CoScripter participants' experiences. (Composite) participant "Grace" was looking for 2-bedroom apartments near the OSU campus that were under $800/month. As instructed, she was trying to make a CoScripter script to automate the searching process so that she could run the script periodically to monitor price and to watch for new listings. She had difficulty even getting started, as already illustrated for CoScripter-F2 in the previous section, perhaps because she had

never before created a script for looking up apartments. Three participants from the Popfly study also experienced this problem in the context of their tasks.

B. An Abstract Solution

One design solution to Grace's problem is to bring to bear the Working Backward and the Analogy strategies, with a Starter Idea to seed her efforts. We first consider this solution abstractly, and then "concretize" it for the CoScripter environment. This design solution allows us to explore one slice of the design space.

As Fig. 3 illustrates, the solution begins with the Working Backward strategy. The system provides a user having trouble getting started (like Grace) with the suggestion that she give an example of the desired output. The system uses this example to infer a program that could lead to this output, and presents it to the user as a Starter Idea.

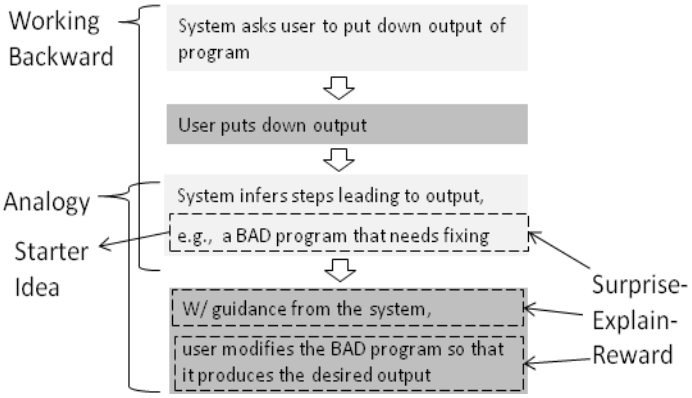

Figure 3. A design solution to Grace's "getting started" problem.

The inferred program need not be good—in fact, presenting a *bad* program that does *not* produce what the user wants may increase the user's engagement in the problem-solving process. Surprised at the undesirable output, the user may seek an explanation, which the system provides. Following the guidance in the explanation, the user can change the bad program to achieve a better version. This is an example of the Analogy strategy: the user changes elements of the bad program to elements better suited to the task.

Embedded in this solution is Surprise-Explain-Reward, a method for enticing a user into useful actions [35]. The essence of the method is to first arouse users' curiosity through the element of surprise (e.g., faulty output), and to then encourage them, through explanations *they* take the initiative to seek out, to make changes that can lead to a working program (the reward). One reason we embed this method as the interaction paradigm is that it does not interrupt users' attention when they are in the midst of problem solving [35].

C. "Concretizing" the Design Solution in CoScripter

Taking this solution to the CoScripter environment begins with attempting to interest Grace in the Working Backward strategy. Since Grace feels "stuck" and therefore is not engaged in tinkering or exploring solution ideas, we will assume that she is scanning the environment, seeking some clue about what to do. CoScripter notices a period of user inactivity and displays hints about two ways to get started (Fig. 4), the second of which can start Grace working backward. (An alternative way to offer starter hints is to display them as soon as CoScripter starts. This not only allows the interested users to follow through but also frees the system from having to detect when users are unable to start). Grace notices the hint, and since she does not know what actions she would want to record, she decides to try the table suggestion (Fig. 5).

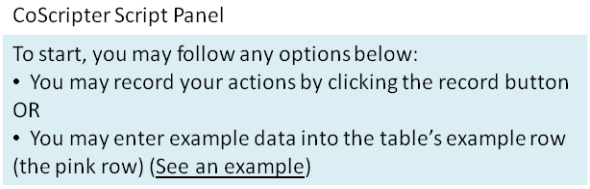

Figure 4. Hint on how to get started

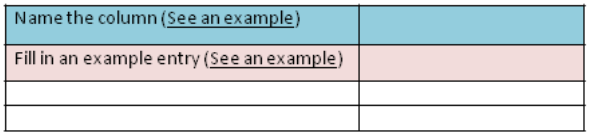

Figure 5. The table at the start

Grace fills in her apartment's address as an example of the desired output of the script (Fig. 6), i.e., an apartment address in the first column. This is the first step of the Working Backward strategy.

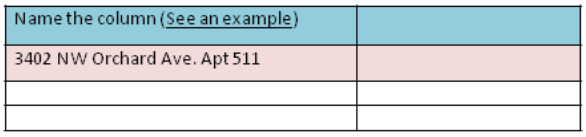

Figure 6. Table with an example entry

Grace's example output allows the system to infer a script that might work for Grace. The system recognizes the example as an address (e.g., through the help of an auxiliary tool like Topes [30] that can recognize common data types such as addresses and telephone numbers). The system also has a small database of a few specific websites containing common types of data. For example, it knows that restaurants.com contains addresses (of restaurants). Since Grace entered an address, the system uses this information and produces the script in Fig. 7. This script, which is an instantiated "design pattern" in template form, is a Starter Idea for Grace. But Grace's eyes glaze over at the sight of the script, so she just runs it instead, and gets the output in Fig. 8.

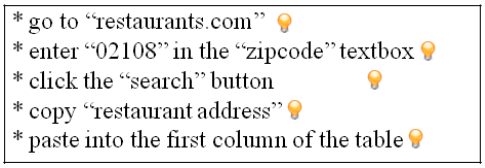

Figure 7. The starter script

Name the column (See an example)	
3402 NW Orchard Ave. Apt 511	
208 SW 2nd, Corvallis, OR 97333	
9th Street, Corvallis, OR	
…	

Figure 8. Output of the starter script

Grace is surprised that the script is showing restaurant addresses instead of apartments, so she takes a closer look at the script after all to seek an explanation. As she moves her mouse pointer over the lightbulb next to the first line of the script, a tooltip appears (Fig. 9). The explanation makes it clear to Grace that she needs to change from restaurants.com to something "like" restaurants.com that is about apartments instead of restaurants (Analogy strategy). She anticipates that this course of action will reward her with a working script.

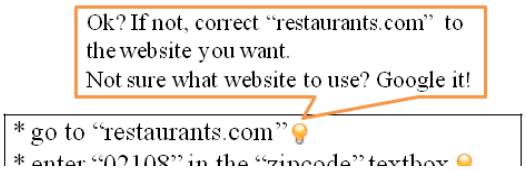

Figure 9. An explanation of the first line of the script

VI. A Multidimensional Design Space

The solution we have just presented is only a narrow slice of the design space for helping to nurture end-user programmers' ideas. Fig. 10 suggests an array of research opportunities in numerous dimensions.

One dimension (Fig. 10) is the intended *audience*. Although we have expressed the notion of nurturing ideas from an end-user programming perspective, there is also a dearth of support for nurturing professional programmers' ideas. A third possibility is computer science novices. The choice of audience would likely also impact the range of possibilities in other dimensions, e.g., types of strategies.

In helping people with their program problem-solving ideas, there are opportunities for supporting different *stages* of idea development (as laid out by the Design Cycle Model, a general model for creative design processes [13]): idea generation, idea implementation, and idea evaluation. The example solution in Section V does not address the entire model; it assists users in only the idea generation stage.

Another interesting dimension is the *sources* of meta-ideas (ideas about how to nurture ideas), i.e., who or what provides the seeding at the root of the nurturing? In our example solution, the system fulfilled this function, but other possible solutions may move this task to the user, to other members of a group of users, or even to a mixed-initiative approach.

The *attributes* of both the meta-ideas and the ideas being nurtured span a wide range. For example, research on supporting design activities often points to the importance of supporting *vague* ideas as well as concrete ones. (Usually programming environments support only concrete ones.) Having only a vague idea is common at the onset of a task and, as Polya pointed out, having a vague idea is better than having no idea [26]. Another attribute of an idea is *how good* it is; recall that our solution capitalized upon "bad" ideas to bestow a hint while at the same time cognitively engaging users by requiring them to think of how to correct the bad idea. Still another attribute is its *type*, ranging from an idea about a strategy for solving the problem, to an idea about the solution itself.

These dimensions, together with the remaining dimensions of Fig. 10 discussed in the earlier sections, suggest a number of interesting research questions, such as:

1. How might an idea gardening subsystem detect when a user is experiencing a difficulty? Or should instead the users be responsible for this, pressing a "Help" button when needed?
2. What are reasonable approaches for an idea gardening subsystem to decide which problem-solving strategies to present to a user?
3. Is there a danger of "trapping" users in a particular strategy, making it difficult for users to flexibly solve problems without the system constraining their way of working?
4. Will the presentation of numerous ideas overload users with too much information, rather than helping them?

And, perhaps most critical of all:

5. Will users become too reliant on the idea gardening subsystem, thereby becoming weaker problem solvers about their programs rather than stronger problem solvers?

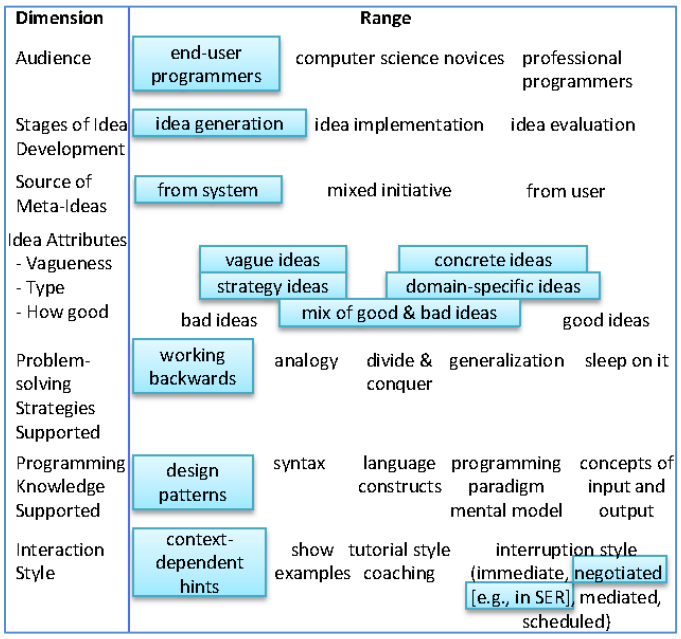

Figure 10. Design Space for gardening end-user programmers' ideas. Shading indicates parts of the space instantiated in this paper.

VII. Conclusion

In this paper, we have presented results of a theory-based empirical exploration, using two end-user mashup programming environments, into design opportunities to help end-user programmers initiate and refine *their own* ideas. We term the design implications of our results idea "gardening" opportunities—they aim at nurturing ideas end-user programmers devise. This notion is different from most tools

for end-user programmers, which aim to lower end-user programmers' cognitive burden in various ways, but do not attempt to directly nurture the ideas they have themselves.

Among our results were:

- Difficulties our participants encountered included metacognitive problems, low self-efficacy leading to inflexibility, design pattern barriers, and a scarcity of ideas that led to numerous issues. Interestingly, programming knowledge, sometimes thought to be unnecessary in end-user environments such as programming-by-demonstration, was still a problem for participants in both environments used in our empirical studies.
- Design solutions to these difficulties can be derived by applying problem-solving and creativity theories, such as enticing users toward particular problem-solving strategies to solve some of the issues they encountered. One example was to use the Working Backward strategy to overcome the "how do I even start" barrier.
- Generalizing the example solution revealed a multi-dimensional design space for "idea gardening", with interesting dimensions such as the stage of idea development, source of meta-ideas, and the attributes of ideas that might be supported.

These results are encouraging, suggesting opportunities for end-user programming researchers to better help end-user programmers to overcome problems and to potentially become more confident along the way. As a result of this kind of research, we hope that creators of future end-user programming environments will be able to anticipate and avert solve-or-abandon moments like this one:

CoScripter-M2: "Those don't seem to work. I'm stuck."

ACKNOWLEDGMENT

This work was supported in part by NSF grant 0917366.

REFERENCES

[1] Bandura, A. Self-efficacy: Toward a unifying theory behavioral change. Psychological Review 8, 2, 1977, 191-215.

[2] Blackwell, A. and Hague, R. AutoHAN: An architecture for programming the home, IEEE VLHCC, 2001, 150-157.

[3] Bloom, B. and Broder, L. Problem Solving Processes of College Students: A Supplementary Educational Monograph. University of Chicago Press. 1950.

[4] Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S. Example-centric programming: Integrating web search into the programming environment, ACM CHI, 2010, 513-522.

[5] Cao, J., Rector, K., Park, T., Fleming, S., Burnett, M., Wiedenbeck, S. A debugging perspective on end-user mashup programming, IEEE VL/HCC, 2010, 149-156.

[6] Cao, J., Riche, Y., Wiedenbeck, S., Burnett, M. and Grigoreanu, V. End-user mashup programming: Through the design lens, ACM CHI, 2010, 1009-1018.

[7] Compeau, D. and Higgins, C. Computer self-efficacy: Development of a measure and initial test. MIS Quarterly 19, 2, 1995, 189-211.

[8] Díaz, P., Aedo, I., Rosson, M., Carroll, J. A visual tool for using design patterns as pattern languages, AVI, 2010, 67-74.

[9] Fisher, G. End-user development and meta-design: foundations for cultures of participation, EUD 2009 (LNCS 5435), Siegen, Germany, Springer-Verlag, 2009, 3-14.

[10] Flavell, J. Metacognition and cognitive monitoring: A new area of cognitive-developmental inquiry. American Psychologist 34, 1979.

[11] Gross, P., Herstand, M., Hodges, J., and Kelleher, C. A code reuse interface for non-programmer middle school students, ACM UIST, 2010, 219–228.

[12] Hartmann, B., MacDougall, D., Brandt, J., Klemmer, S., What Would Other Programmers Do? Suggesting Solutions to Error Messages, ACM CHI, 2010, 1019 -1028.

[13] Herring, S., Jones, B. and Bailey, B. Idea generation techniques among creative professionals. Proc. HICSS, 2009.

[14] Jansson, D. and Smith, S. Design fixation. Design Studies 12, 1, 1991.

[15] Jonassen, D. Toward a design theory of problem solving, Educational Technology Research and Development 48, 4, Springer, 2000, 63-85.

[16] Kelleher C. and Pausch, R. Lessons learned from designing a programming system to support middle school girls creating animated stories, IEEE VL/HCC, 2006, 165-172.

[17] Ko, A., Myers, B. and Aung, H. Six learning barriers in end-user programming systems, IEEE VLHCC, 2004, 199-206.

[18] Ko, A., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., Wiedenbeck, S., The State of the Art in End-User Software Engineering, ACM Computing Surveys, 43(3), 2011, 21:1-21:44.

[19] Levine, M., Effective Problem Solving, Prentice Hall, 1994.

[20] Lieberman, H. Your Wish Is My Command: Programming By Example. Morgan Kaufmann. 2001.

[21] Lin, J., Wong, J., Nichols, J., Cypher, A., and Lau, T. End-user programming of mashups with Vegemite, ACM IUI, 2009, 97–106.

[22] Newman, M., Lin, J., Hong, J., Landay, J. DENIM: An informal web site design tool inspired by observations of practice. Human-Computer Interaction 18, 3, 2003, 259-324.

[23] Oney S. and Myers, B. FireCrystal: Understanding interactive behaviors in dynamic web pages, IEEE VLHCC, 2009, 105-108.

[24] Osborn, A. Applied Imagination: Principles and Procedures of Creative Problem Solving. Charles Scribner's Sons, 1953.

[25] Pane, J. and Myers, B. More natural programming languages and environments, in End User Development (Henry Lieberman et al., eds.), Springer, 2006, 31-50.

[26] Polya, G. How to Solve It: A New Aspect of Mathematical Method, Princeton University Press, 1973.

[27] Repenning, A. and Ioannidou, A. Broadening participation through scalable game design. ACM SIGCSE, 2008, 305–309.

[28] Robertson, T., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J., Beckwith, L. and Phalgune, A. Impact of interruption style on end-user debugging, ACM CHI, 2004, 287-294.

[29] Scaffidi, C., Shaw, M. and Myers, B. Estimating the numbers of end users and end user programmers, IEEE VLHCC, 2005, 207-214.

[30] Scaffidi, C. Sharing, finding and reusing end-user code for reformatting and validating data. J. Visual Languages and Computing 21, 4, 2010, 230-245.

[31] Schön, D. The Reflective Practitioner: How ProfessionalsThink in Action. Basic Books, 1983.

[32] Simon, H. Problem solving and education, in Problem Solving and Education: Issues in Teaching and Research, Tuma, D. and Reif, F., (eds.) Lawrence Erlbaum, 1980.

[33] Soloway, E., Learning to Program = Learning to Construct Mechanisms and Explanations, CACM 29, 9, 1986, 850-858.

[34] Wickelgren, W. How to Solve Problems: Elements of a Theory of Problems and Problem Solving, W. H. Freeman, 1974.

[35] Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M. and Rothermel. G. Harnessing curiosity to increase correctness in end-user programming. ACM CHI, 2003, 305–312.