Idea Garden: Situated Support for Problem Solving by End-User Programmers[†]

JILL CAO[1,*], SCOTT D. FLEMING[2], MARGARET BURNETT[1] AND CHRISTOPHER SCAFFIDI[1]

[1]*School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA*
[2]*Department of Computer Science, University of Memphis, Memphis, TN, USA*
[*]*Corresponding author: caoch@eecs.oregonstate.edu*
[†]*An early version of portions of this paper appeared in Cao et al. (2011).*

Although there have been many advances in end-user programming environments, recent empirical studies report that programming still remains difficult for end-users. We hypothesize that one reason may be lack of effective support for helping end-user programmers problem-solve their own way around barriers they encounter. Therefore, in this paper, we describe the Idea Garden, a concept designed to help end-user programmers generate new ideas and problem-solve when they run into barriers. The Idea Garden has its roots in Minimalist Learning Theory and problem-solving theories. Our proof-of-concept prototype of the Idea Garden concept in the CoScripter end-user programming environment currently targets three barriers reported in end-user programming literature. It does so using an integrated, just-in-time combination of scaffolding for problem-solving strategies, for design patterns and for programming concepts. Our empirical results showed that this approach helped end-user programmers overcome all three types of barriers that our prototype targeted.

RESEARCH HIGHLIGHTS

- We propose the Idea Garden concept to help users overcome barriers.
- We present a study that identified barriers end-user programmers encountered.
- We analyzed the barriers from the angles of problem-solving theories.
- We prototyped the Idea Garden for the CoScripter end-user mashup environment.
- An evaluation showed that the Idea Garden helped most users overcome barrier(s).

Keywords: development frameworks and environments; designing software; interactive learning environments; computational thinking; informal learning

Editorial Board Member: Françoise Détienne

Received 16 April 2013; Revised 18 March 2014; Accepted 24 April 2014

1. INTRODUCTION

Over the decades, researchers have made remarkable strides in bringing programming capabilities to ordinary end-users. 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, based on data from the U.S. Bureau of Labor and Statistics, the number of people using spreadsheets and databases at work has been estimated to be 55 million—more than an order of magnitude greater than the number of professional programmers (Scaffidi *et al.*, 2005). Further, research continues to make progress in designing programming languages that are easier for end-users to use (e.g. Pane and Myers, 2006), that help end-users find and reuse programs (e.g. Brandt *et al.*, 2010) and that even eliminate the need for explicit programming altogether (e.g. Lieberman, 2001).

However, despite this progress, empirical studies continue to show that programming remains difficult for end-users (e.g. Cao *et al.*, 2010a,b; Gross and Kelleher, 2010; Ko *et al.*, 2004). Prior empirical investigations suggest that one reason barriers persist is a lack of support in end-user programming environments for helping users to develop problem-solving skills needed to overcome barriers they encounter (Cao *et al.*, 2010a,b).

To address this gap, we have been working on a concept we call the *Idea Garden*, a design concept that extends end-user programming environments with situated assistance to help users form and develop ideas when problem solving, without interrupting or taking control from users. In contrast to prior approaches that attempt to solve problems *for* end-user programmers, the Idea Garden supports end-users as they solve problems *for themselves*.

This paper makes four contributions:

(i) The Idea Garden, a new design concept for programming environments that is aimed at helping end-user programmers problem-solve and generate *their own* ideas.
(ii) The application of theories from problem-solving literature and creativity literature that revealed problem-solving barriers end-user programmers encountered in two end-user mashup programming environments (Study 1).
(iii) A prototype instantiation of the Idea Garden for the end-user programming environment CoScripter (Lin *et al.*, 2009; Little *et al.*, 2007).
(iv) Empirical evidence of the Idea Garden's strengths and weaknesses in helping end-user programmers work through barriers identified by Study 1 (Study 2).

2. RELATED WORK

Historically, a central goal in the design of end-user programming environments has been to make programming as easy as possible.

One approach aiming at this goal is to *simplify* programming languages to make them easier for users to understand and use. For example, the Natural Programming project promotes designing programming languages to match users' natural vocabulary and expressions of computation (Myers *et al.*, 2004). One language in that project, the HANDS system for children, depicts computation as a friendly dog who manipulates a set of cards based on graphical rules, which are expressed in a language carefully designed to match how children described games (Pane and Myers, 2006). An empirical study demonstrated that this language significantly increased users' ability to complete programming problems (Pane and Myers, 2006). Another system, Vegemite, aims to make mashup-programming easy by providing table data structures akin to spreadsheets that remove the need for explicit iteration constructs (Lin *et al.*, 2009). In an early study, users said it would be easy for them to create new mashups with the system (Lin *et al.*, 2009). Still other programming environments such as Alice (Kelleher and Pausch, 2006) and AutoHAN (Blackwell and Hague, 2001) incorporate visual programming languages and direct or tangible manipulation to make programming easier for end-users. Indeed, these and other novel programming-environment designs have demonstrated considerable success at easing programming, by reducing the need for users to learn arcane language constructs and to memorize syntax.

Some environments are even designed with the goal of *eliminating* the need for explicit programming. For example, AgentSheets (Repenning and Ioannidou, 2008) and CoScripter (Lin *et al.*, 2009) are based on the approach called programming by demonstration where end-users demonstrate an activity from which the system automatically generates a program (Lieberman, 2001; Cypher *et al.*, 2010). Some such environments provide a way for users to access the generated code, but it is often not necessary for users to edit the code at all.

Another family of approaches seeks to *delegate* some of the programming responsibilities to other people. For example, meta-design aims at design and implementation of systems by professional programmers such that the systems are amenable to redesign through tailoring (configuration and customization) by end-user programmers (Andersen and Mørch, 2009; Costabile *et al.*, 2009; Fischer, 2009). In some large organizations, an expert end-user programmer, called a *gardener*,[1] serves to ease or eliminate programming among the organization's end-user community (Gantt and Nardi, 1992). Such a gardener creates reusable code, templates and other resources, and provides these to other users, whose programming tasks thereby become substantially simpler. While gardeners each focus on a particular end-user community, programming environments facilitate delegation of programming across communities by aiding reuse of code. For example, FireCrystal (Oney and Myers, 2009) is a Firefox plug-in that allows a programmer to select user interface elements of a webpage and view the corresponding source code. FireCrystal then eases creation of another web page by providing features to extract and reuse this code, especially code for user interface interactions. Another system, BluePrint (Brandt *et al.*, 2010), is an Adobe Flex Builder plug-in that semi-automatically gleans task-specific example programs and related information from the web, and then provides these for use by end-user programmers. Still other systems are designed to emulate strategies or heuristics that users themselves appear to employ when looking for reusable code, thereby simplifying the task of choosing which existing programs to run or reuse (e.g. Gross *et al.*, 2010; Scaffidi *et al.*, 2009).

[1]Gantt and Nardi's use of the term *gardener* is unrelated to our use of it with respect to the Idea Garden. Idea Garden is also unrelated to a work known as the 'Answer Garden' (Ackerman and McDonald, 1996) which, like Gantt and Nardi's work, is focused on a collaborative organizational setting, unlike our work here.

Although all of the above approaches help end-users by simplifying, eliminating or delegating the challenges of programming, none are aimed at nurturing end-users' problem-solving ideas. In essence, these approaches help end-user programmers by lowering barriers, *rather than by helping people figure out for themselves* how to surmount those barriers.

However, there are some works aimed at helping professional interface designers generate and develop ideas for their interface designs. For example, DENIM is a system that allows designers to sketch websites at a high level and thus helps the designers develop their design ideas (Newman *et al.*, 2003). As another example, Diaz *et al.* created a visual language that helps web designers develop their design ideas by suggesting potentially appropriate design patterns along with possible benefits and limitations of the suggested patterns (Díaz *et al.*, 2010). Even though professional interface designers are different from the end-user programmer audience we target (in that interface designers often do not engage in coding per se), as with that line of work, we seek to help end-user programmers generate new ideas and problem-solve about making their programs behave as desired. Thus, our approach was inspired in part by this line of work on interface designers.

Our work is also somewhat related to research that aims to help naive users learn programming, often through the use of new kinds of educational approaches, or special-purpose programming languages and tools (Dorn, 2011; Guzdial, 2008; Hudhausen *et al.*, 2009; Kelleher and Pausch, 2006; Lee *et al.*, 2013; Repenning and Ioannidou, 2008). However, these approaches aim to help users who aspire to learn programming per se, whereas our approach aims at a different population: those who do *not* wish to learn (more) programming, but are willing to do just enough programming to accomplish other tasks in their lives. One example of our target population could be an accountant who does *not* aspire to learn more than she already knows about spreadsheet formulas, but rather just wants to get her budget working correctly.

Given this target population, the most relevant work is the foundational research on Minimalist Learning Theory (MLT) (Carroll, 1990, 1998; Carroll and Rosson, 1987; van der Meij and Carroll 1998). Rooted in the constructivism of Bruner and Piaget, MLT is an education theory that explains how (and why) to design instructional materials for populations like the accountant above: the theory terms people like this 'active' computer users. Active computer users are those whose primary motivation is to *do* some computer-based task of their own, not particularly to *learn* computing skills. Active users are so focused on the task at hand that they are often unwilling to invest time in taking tutorials, reading documentation or using other training materials—even if such an investment would be rational in the long term. This phenomenon is termed the 'paradox of the active user' (Carroll and Rosson, 1987). Helping users who face this paradox learn *despite* their lack of interest in learning per se is the goal of MLT.

One way that MLT targets active users is to emphasize the importance of task-focused activities. Specifically, it suggests that effective learning activities for active users should (1) permit self-directed reasoning, (2) be meaningful and self-contained, (3) provide realistic tasks early on, (4) be closely linked to the actual system and (5) provide for error recognition and recovery.

To target active users like these, MLT rests upon four major principles (van der Meij and Carroll, 1998):

> *MLT Principle 1: Choose an action-oriented approach*. Since MLT's 'active users' are by definition eager to act, the theory recommends providing an immediate opportunity to act; encouraging self-directed exploration and innovation; and designing instruction in a way that puts the user's own (sub)goals and activities first, rather than putting the delivery of information first.
>
> *MLT Principle 2: Anchor the [learning] tool in the task domain*. Almost a corollary to MLT Principle 1 is the recommendation that learning tools should use real tasks as instruction activities. To accomplish this, the theory also recommends that the structure or sequence of instruction follow the structure or sequence of the real task.
>
> *MLT Principle 3: Support error recognition and recovery*. MLT recommends that a tool should prevent mistakes when possible, provide error information that supports not only detection, but also diagnosis and recovery, and provide immediate, 'on-the-spot' error information.
>
> *MLT Principle 4: Support reading to do, study, and locate.* MLT explains that users' reading style of instructional materials can vary: some read just enough to do something, others study in more breadth and others look up what they were looking for. Thus, MLT recommends designing for all three styles. To do so, it recommends being very brief, not spelling out everything and making each instructional unit self-contained, without needing cross-referencing/navigation to other sections of the instruction.

In the next section, we will return to these four principles, and explain how they have been embodied in the Idea Garden concept.

3. THE IDEA GARDEN: CONCEPT

To achieve our goal of helping 'active' end-user programmers generate new ideas and problem-solve *in situ*, when they are engaged in programming, we have devised a concept called the *Idea Garden*. The Idea Garden concept provides a new kind of scaffolding for problem-solving strategies and programming knowledge.

Table 1. Constraints on Idea Garden features' content, form and style.

Type of constraint	Constraint
Content	CC1: Idea Garden suggestions must contain problem-solving strategies (see Section 5.2 for an example)
	CC2: Idea Garden suggestions must contain programming concepts and design patterns (see Section 5.2 for an example)
	CC3: Any code examples must be intentionally imperfect matches to the user's needs (see Section 5.2 for an example)
Form	FC1: Idea Garden suggestions must follow the appropriate template. These templates enumerate suggestion content in a host-independent way (see Section 5.2 for an example)
Interaction style	ISC1: Interruption style: negotiated. Idea Garden suggestions never pop up uninvited; instead, an Idea Garden merely adds notifications that suggestion content is available. These notifications cannot pop up either; they can be added only when the GUI would have needed to repaint for other reasons (see Section 5.1 for an example)
	ISC2: Personality: non-authoritative. An Idea Garden feature's appearance and suggestion tone should not suggest that it is an automatic problem-solver or an authoritative figure. (See Section 5.1 for an example)

We define an Idea Garden as

(i) a subsystem that extends a 'host' end-user programming environment to provide suggestions that
(ii) follow the principles from MLT as per Idea Garden Design Principles 1–4 (Section 3.1) and
(iii) follow the form, content and interaction style constraints enumerated in Table 1. Specifically, Idea Garden suggestions:
(a) provide guidance about problem-solving strategies, programming concepts and design patterns (Table 1: CC1 and CC2);
(b) are generated by combining host-independent templates (Table 1: FC1) with information about the user's task;
(c) are intentionally imperfect (Table 1: CC3) and presented in a non-authoritative tone (Table 1: ISC2);
(d) are presented via a negotiated interruption style (Table 1: ISC1).

The following subsections discuss the theoretical underpinnings of the Idea Garden concept, as well as the principles that guide and constrain an Idea Garden's suggestions.

3.1. The Idea Garden's adherence to the MLT principles

One element of the Idea Garden's definition is that it must conform to the MLT principles (which were enumerated in Section 2), because that is the learning theory that addresses the Idea Garden target population of 'active' computer users; i.e. those who are more interested in doing their tasks than they are in learning for the sake of learning computing skills (Carroll and Rosson, 1987). The Idea Garden follows these principles in the following ways.

Idea Garden Design Principle 1: Require self-directed reasoning: This design principle was derived directly from MLT Principle 1 (action-oriented instruction). This is one way the Idea Garden contrasts with existing tools that attempt to solve users' problems automatically: such tools do not require users to reason for themselves. The Idea Garden therefore aims to *not* be helpful *unless* the user engages in self-directed reasoning. Specifically, it suggests strategy alternatives (Table 1: CC1) and provides (intentionally) incomplete or imperfect suggestions (Table 1: CC3), all of which require the user to actively try things out and problem-solve in order to make substantive progress on the task at hand.

Another way the Idea Garden requires self-direction is in leaving the workflow decisions to the user. In particular, the Idea Garden contrasts with assertive instructional agents, such as Microsoft Office's Clippy, that violate users' self-directedness. Whereas Clippy uses immediate-style interruptions that hijack the user's attention whenever the *system* decides it has something to say through, e.g. pop dialogs or animations, the Idea Garden uses negotiated-style interruptions, which inform users of pending messages but do not force the users to acknowledge the messages (McFarlane, 2002; Table 1: ISC 1). Negotiated-style interruptions, by definition, leave the decision to the *user* as to whether and when to bring up the new messages. Perhaps because of this characteristic, negotiated-style interruptions have been shown to promote not only *learning* of new features in an end-user programming environment, but also *debugging effectiveness* better than immediate-style interruptions (Robertson *et al*., 2004).

Idea Garden Design Principle 2: Based on the user's own tasks. We derived this principle from MLT Principle 2 (anchored in task domain). Adherence to this principle is the reason the Idea Garden customizes suggestions to the user's *own* code as it currently exists. Each suggestion is then, by definition, tied to the task that the user has already chosen to initiate, thereby giving suggestions relevance and realism. Adherence to this principle is also the reason the Idea Garden extends, rather than replaces, existing host environments.

Idea Garden Design Principle 3: Leave error messages and recovery to the host environment. MLT Principle 3 points out the importance of providing mechanisms for active users to recover from errors. Since an Idea Garden extends an existing host, it defers to the host for error recovery. We do not expect the Idea Garden to replace error facilities in the host.

Idea Garden Design Principle 4: For do-ers, studiers and references. Because MLT Principle 4 recommends brief yet self-contained content, Idea Gardens present as little information as possible, and always leave out details that the user will have to figure out themselves (Design Principle 1). At the same time, much of its content is expandable, allowing studiers to see more without navigating away 'in place'. Finally, Idea Gardens provide a context-free way to look up Idea Garden suggestions even if the user's current context is not one that would normally trigger that suggestion.

3.2. Idea Garden's constraints on form, content and style

An Idea Garden needs to be well integrated with its host environments; thus, Idea Garden will not all look exactly alike. Not only will the content of an Idea Garden's suggestions need to reflect the host-supported programming concepts and paradigms (Table 1: CC2), but also the host's GUI conventions and affordances will vary. To allow the needed flexibility to conform to the host, we include in the Idea Garden definition a set of constraints, enumerated in Table 1. The constraints mainly elaborate upon and supplement the Minimalist Learning principles discussed above.

3.3. Idea Gardens as extensions to end-user programming environments

An Idea Garden *extends* an existing end-user programming environment. We chose this approach because the Idea Garden is not intended to replace all the different kinds of helpful information that a programming environment might offer, such as tutorials for the novice user, examples and so on. Instead, its goal is to *supplement* those kinds of information when the user has run into barriers despite having those other resources.

The Idea Garden is general enough to extend any host environment that allows enough communication to monitor the user's actions and code, and to add communications of its own to that environment. Specifically, it can extend any host environment that: (1) allows the Idea Garden to retrieve the user's data and code as it appears to the user (i.e. on the screen) and as it appears to the machine (i.e. after parsing), that (2) allows the Idea Garden to change the user's code (e.g. by inserting constants or lines of code) and that (3) allows the Idea Garden to annotate the programming environment and/or user's code with interactive widgets (e.g. tool tips, buttons, graphics or font changes).

4. STUDY 1: END-USER PROGRAMMERS' IDEA BARRIERS

Within this framework, we needed an understanding of what kinds of 'idea barriers' end-user programmers encounter. Toward that end, we began with a formative empirical investigation. We chose mashup programming environments as the context for the study, because web automation by end-users has become increasingly popular (e.g. Cypher *et al.*, 2010; Scaffidi *et al.*, 2008; Zang *et al.*, 2008).

4.1. The Mashup environments

To help ensure the generality of our findings, we studied users of two different mashup environments: IBM's CoScripter tool and Microsoft's Popfly tool. Mashups are web applications end-user programmers can create that interactively combine data from multiple internet sources (Wong and Hong 2007). Both environments aim to cater to end-user programmers by adopting techniques for facilitating programming by end-user programmers. Specifically, CoScripter produces scripts through programming by demonstration where end-user programmers demonstrate an activity from which the system automatically generates a program (Lieberman, 2001). In contrast, Popfly provides a drag-and-drop interface for programming in a visual language.

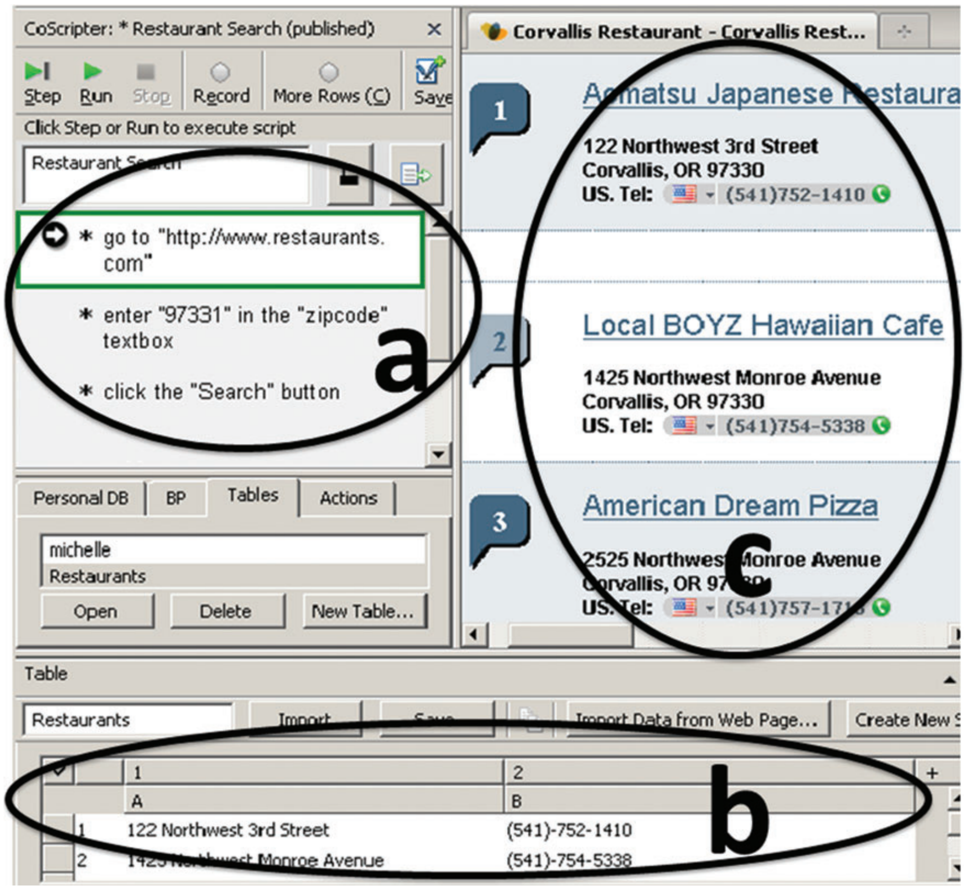
4.1.1. CoScripter

CoScripter is an end-user programming-by-demonstration environment for web scripting. In CoScripter (Fig. 1), users demonstrate to the system how they would carry out a task on the web by actually performing the task themselves. For example, to demonstrate to the system how to find a list of local restaurants, a user could go to restaurants.com, enter search criteria, e.g. zip code, and hit the 'Search' button. The system watches and translates users' actions into a script (Fig. 1a). The user can execute this script at a later time to perform the same task again.

CoScripter enables mashup programming via the table feature (Lin *et al.*, 2009; Fig. 1b). Users can create scripts that automatically copy data between the table and web pages. This enables the user to combine data from multiple web pages in the table and to effect flow of data from one web page to fill-in-the-blank fields of another to compute additional information, the essence of a mashup. For example, to calculate travel time to restaurants, a user could create a script that loads a web page listing all restaurants in an area (e.g. by using restaurants.com; Fig. 1c), copy those restaurants' addresses to a table (Fig. 1b), send each restaurant address to another web page (e.g. Google Maps) that computes travel time via public transit and, finally, copy this output into the table column.

4.1.2. Popfly

In contrast to 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 each take input parameters. For example, a Flickr block can retrieve photos from Flickr.com's web services based on input parameters such as photo names and



* go to "http://www.restaurants.com"
* enter "97331" in the "zipcode" textbox
* click the "Search" button

1 Aomatsu Japanese Restaura
122 Northwest 3rd Street
Corvallis, OR 97330
US. Tel: (541)752-1410

2 Local BOYZ Hawaiian Cafe
1425 Northwest Monroe Avenue
Corvallis, OR 97330
US. Tel: (541)754-5338

3 American Dream Pizza
2525 Northwest Monroe Avenue
Corvallis, OR 97330
US. Tel: (541)757-1713

	1	2
	A	B
1	122 Northwest 3rd Street	(541)-752-1410
2	1425 Northwest Monroe Avenue	(541)-754-5338

Figure 1. The CoScripter environment's (a) script area, (b) table area and (c) browsing area.

photographer names. Users may connect blocks to form a network in which blocks can use output from other blocks as inputs. Figure 2 shows an example mashup in which the Flickr block sends a list of images about 'beaches' with their geographical coordinates to a Virtual Earth block (Fig. 2: top and middle) to display them on a map (Fig. 2: bottom).

4.2. Empirical methods

The goal of our formative empirical study was to reveal the kinds of programming barriers that end-users encounter. We obtained the data by asking end-users to perform programming tasks while we observed. For the Popfly tool, we had already conducted such a study for another purpose[2] (Cao *et al.*, 2010a) and were able to reanalyze the data set, as we describe in detail below. For the CoScripter tool, we had not yet conducted such a study and needed to collect a new data set. Our results (Section 4.3) are based on our qualitative analysis of both data sets together.

[2]The previous study adopted a design theory lens to understand end-user programming.

4.2.1. *Participants*

In total, our CoScripter and Popfly data sets contained user data from sixteen university students. The CoScripter data set had six participants (three males, three females), and the Popfly data set had ten (six males, four females). The participants were from a variety of majors (e.g. graphic design, accounting, wood science) and had little (e.g. high school Visual Basic) or no programming experience. Below, we refer to participants with identifiers, such as *CoScripter-F2*, to indicate the participant's environment (i.e. *CoScripter* or *Popfly*), gender (i.e. *F* or *M*) and ID number within that environment (e.g. 2).

4.2.2. *Programming tasks*

Each study called for participants to create a mashup. The CoScripter participants' primary task was to create a mashup script to automatically search for two-bedroom apartments that are rented for under $800 per month and are within a 30-min drive of the Oregon State University campus. This task required combining data from a search site, such as Craigslist or Apartments.com, with data from a maps site, such as Google Maps. The Popfly participants' task was to create a mashup that integrated movie-related information, such as movies currently

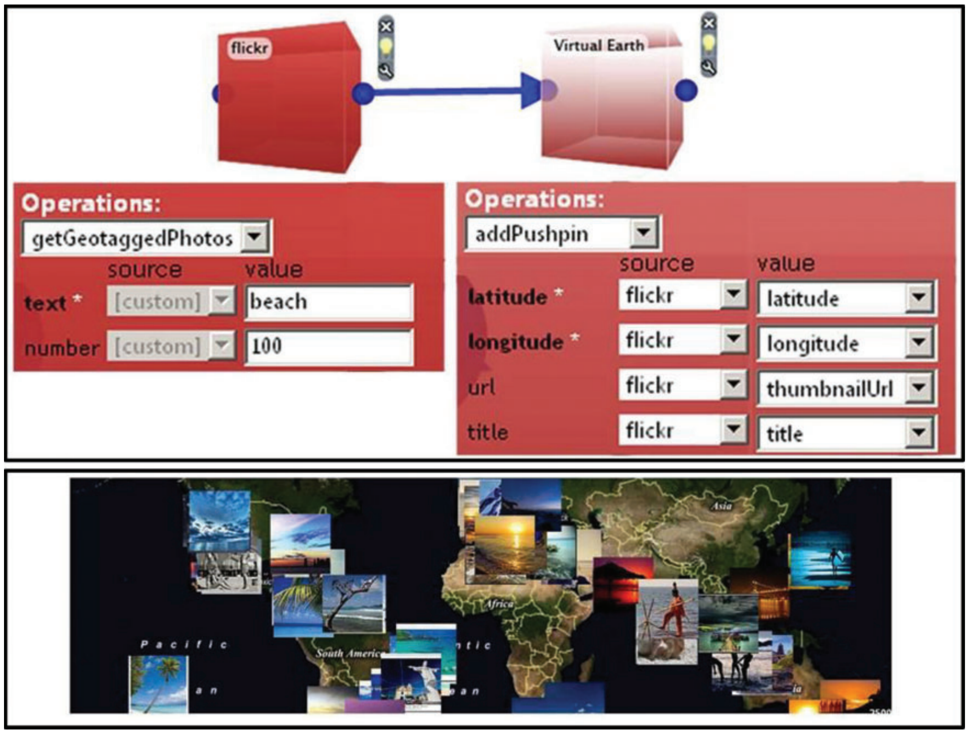


Figure 2. An example mashup in Popfly Mashup Creator. Top: the blocks. Middle: some of the blocks' settings. Bottom: results generated by pressing the Run button (not shown).

showing at local theaters, theater information and news stories about each movie.

4.2.3. Procedures

For each environment, we conducted user sessions one participant at a time using the think-aloud method. Participants first filled out a background questionnaire and completed a 20-min hands-on tutorial that familiarized them with the mashup environment. They then completed a self-efficacy questionnaire (Appendix) based on Compeau and Higgins' questionnaire (Compeau and Higgins, 1995) that we adapted to the task of creating a script. Participants practiced 'thinking aloud' before proceeding to the main task. Finally, they had 50 min to complete their task.

To collect as much data as possible, we wanted to allow participants to get past whatever current barrier was holding them up, and move on reasonably soon, so that they might reveal subsequent barriers. Toward that end, when users were unable to overcome barriers in the tools, we provided assistance. Specifically, when Popfly participants were unable to make progress for 15 min, the researcher administered an extra 5-min mini-tutorial; half of these participants (two males, three females) received the mini-tutorial. Some users still spent a lot of time on barriers, so we revised the threshold for the second study with CoScripter. For these participants, if they were unable to make progress for at least 3 min, the researcher prompted them with hints, such as suggesting that they try a different website or try using the table; all of these participants received at least one hint. We recorded audio and video of the participants as they worked on the main task as well as video captured their screen activity.

We analyzed the audio, video and screen-captured data using content analysis (Seaman, 2008) to categorize these data into the barriers users encountered, their problem-solving strategies, and programming knowledge users applied or failed to apply when faced with the barriers. We used as evidence of a barrier a participant verbalizing a need for help, turning to the programming environment's help facility, asking the researcher for help, or the researcher offering assistance from observing a participant struggle without progress on a single subproblem for at least 3 min. Our analysis revealed different patterns of participant responses to barriers, with a demonstration of a lack in problem-solving strategies, and programming knowledge. We triangulated findings across participants and environments to identify crosscutting phenomena, which we report below.

4.3. Results and implications

According to Simon, two types of skills are necessary for problem solving in a specific domain: *domain-specific knowledge* and *problem-solving strategies* (Simon, 1980). He equates domain-specific knowledge and general problem-solving strategies to 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' (Simon, 1980). 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 indispensable to a successful math problem-solver (Bloom and Broder, 1950).

Consistent with this prior work, we have organized our results under the headings of problem-solving strategies and domain-specific knowledge below, and we explain the relationship of each result to applicable theories about problem solving.

4.3.1. Problem-solving strategies

In general, participants struggled the most when they failed to consider whether their current approach to the problem was succeeding. In other words, participants were stymied when they demonstrated little metacognition: a person's conscious awareness and monitoring of his/her own thoughts, understanding or learning (Flavell, 1979).

For example, CoScripter-F2 showed no sign of metacognition regarding her problem-solving strategies. Her quote below shows that she failed to reflect on her strategy when she was having difficulty making 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."

As a result, the researcher had to prompt her with hints to help her make progress.

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. He never showed any signs of 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]

Conversely, 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."

The demonstration of low metacognition by some participants may, in turn, have been due to low self-efficacy: a person's confidence in their ability to succeed at a specific kind of task (Bandura, 1977). According to self-efficacy theory, people with low self-efficacy tend to be less flexible in their problem-solving strategies than those with high self-efficacy, such as staying with a known approach even if when it is not paying off.

In both studies, low self-efficacy participants indeed demonstrated this inflexibility. For example CoScripter-F3, who had the lowest self-efficacy score in that study (3.4, vs. an average of 3.77 for all participants), did not even consider switching from a conventional Google search to some other approach—such as using a table, as she had just practiced during the hands-on tutorial—until the researcher's prompt nudged her into reconsidering her problem-solving strategy:

CoScripter-F3: (after several trials with Google searches) "We can set the price range, and how many bedrooms but 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?"

CoScripter-F3: "I'd go to Google Map or something, if I had address and I wanted to know how far it was… Oh you showed me how to do that using the table [from the tutorial]!"

Literature on problem solving emphasizes the inadvisability of such inflexibility, explaining that when a person is stuck on a problem, it is important to step back and analyze what he/she has been doing (Wickelgren, 1974). Creative design literature agrees, using the term 'design fixation' to refer to the undesirable situation where the designer becomes overly focused on one idea, missing out on other opportunities (Jansson and Smith, 1991). Conversely, successful design requires periodically approaching the problem from a different perspective, which is called *ideational fluency* in creativity literature, and *reframing* in design literature (Schön, 1983). Our results suggest that an analogous challenge faces end-user programmers.

Viewed from the perspective of this literature, our results suggest two possible ways to help struggling users increase their problem-solving flexibility. One way is to entice such users toward metacognition, for example, by suggesting problem-solving strategies that the user has not yet considered. The second way suggested by this literature is to enhance users' self-efficacy. According to self-efficacy theory (Bandura, 1977), an effective means to increase self-efficacy is through direct experience. These two possibilities can actually converge if, as above, an Idea Garden suggests a new strategy that a struggling user succeeds with, and that success increases the user's self-efficacy. With increased self-efficacy, the user may then become more flexible in his/her choices of problem-solving strategies in the future, as Bandura's theory predicts.

4.3.2. *Programming domain knowledge*

Problem-solving strategies cannot be applied without basic knowledge in the domain where those strategies are to be applied (Simon, 1980). In one sense, end-users are by definition domain experts, and the goal of end-user programming is to bring their domain expertise directly to a programming environment without the need for intermediary professional programmers. However, the other crucial domain here is *programming itself*, and it has been widely reported that many end-user programmers lack expertise in the domain of programming (e.g. concepts of input and output, how to go about debugging), or in the programming language (e.g. a mental model of the programming paradigm being used).

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, issues of programming knowledge 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:

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 abstract ‘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 blocks (analogous to a library of APIs) that could each perform a different portion of the task. Without recognizing the concept of bringing together component parts for a 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]?

Likewise, several participants from the CoScripter study failed to grasp the ideas of using multiple webpages, the CoScripter’s equivalents to APIs, to accomplish their task. For example, CoScripter-F1 did not try to use a second webpage like Google Maps to calculate the driving time between the university and the list of apartments she found from Apartments.com. Instead, she fixated on finding the needed driving time together with the apartments on Apartments.com.

Finally, at the program design level, participants experienced difficulties in generating ideas that could be developed into solutions. These difficulties played out in three ways: lack of a sense for even how to get started (CoScripter-F2), running out of design ideas to try very early (CoScripter-M2) or choosing a starting idea that leads down a wrong path (Popfly-F4).

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: [Enter 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]

These examples have in common missing programming knowledge that was needed to move ahead, such as how ‘repeat’ actually works, how to compose a solution from parts and the notion of making use of well-known design patterns. (Clearly, an end-user programmer who lacks knowledge of a certain design pattern cannot possibly apply it.) Consequently, helping end-user programmers to problem-solve will necessarily also require at least some support for proper usage of the language’s constructs, and for concepts and design patterns available for that programming language/environment.

5. AN INSTANTIATION OF THE IDEA GARDEN IN COSCRIPTER

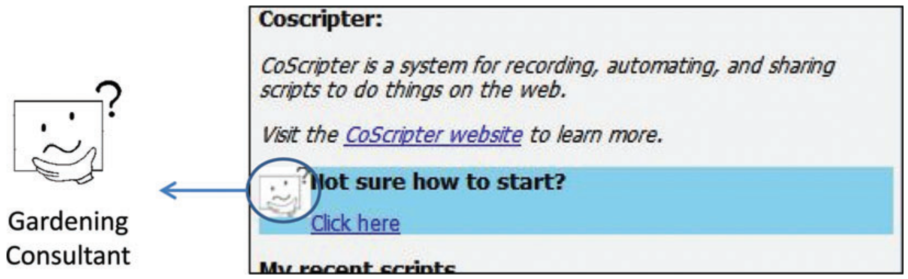
To build upon the findings of Study 1, we instantiated the Idea Garden via a prototype to facilitate iterative development and evaluation of the concept. We chose CoScripter as the host environment for this prototype.

In the initial prototype of the Idea Garden, we designed features to target three end-user programming barriers: not having an idea of how to even start (*How-to-Start*), not understanding how to compose existing modules and functions to create a program (*Composition*), and not understanding how to generalize from operating on a single data item to operating on multiple data items (*More-than-Once*). We selected these barriers because they occurred frequently in both Study 1 and in prior end-user programming literature (e.g. Ko *et al.*, 2004).

In this section, we show how the initial prototype fulfills each of the constraints in Table 1, what the user sees in this prototype and how the prototype constructs its suggestions.

5.1. The Idea Garden prototype’s interaction style

In the prototype, users interact with the Idea Garden as part of their other interactions with the host environment (here, CoScripter). The Idea Garden’s tight integration into the host



Coscripter:

CoScripter is a system for recording, automating, and sharing scripts to do things on the web.

Visit the CoScripter website to learn more.

Not sure how to start?

Click here

My recent scripts

Figure 3. The Idea Garden inserts suggestions (highlighted text) directly into the CoScripter user interface. The Gardening Consultant icon is circled here.

environment is because, as explained in Section 3.1, Idea Garden adheres to the MLT principle (Carroll and Rosson, 1987) that emphasizes the importance of the linkage of the materials to the actual system where the user is working.

Recall from Section 3.1 that Idea Garden suggestions do not pop up uninvited, like the infamous Clippy of Microsoft Office would. Instead, Idea Garden communications follow the Surprise-Explain-Reward approach (Wilson *et al.*, 2003), which includes the negotiated style of interruptions (Robertson *et al.*, 2004; Table 1: ISC1). Therefore, the Idea Garden never interrupts: instead, it adds a small indication of the presence of new information to the environment's user interface (either in unused whitespace or by extending the containing window), with the indicator only appearing or disappearing when the relevant CoScripter window refreshes after a user operation. Thus, as per Surprise-Explain-Reward, the indicator is meant to entice users in need of more information to pursue the explanatory material it has available.

In keeping with the constraint of a non-authoritative personality (Table 1: ISC2), The Idea Garden communicates its suggestions via a non-authoritative character we call the *Gardening Consultant* (Fig. 3). Imbuing such characters with a personality can evoke emotions in the user, such as humor, appreciation or social feelings, and when such emotions are positive, they can enhance the quality and creativity of users' ideas (Lewis *et al.*, 2011; Nass and Moon, 2000). Also, a recent study showed that end-user programmers respond well to instructions given in a non-authoritarian voice (Lee and Ko, 2011). Therefore, the Gardening Consultant's icon looks like a tentative, quizzical face, intended to provoke mild humor. Some of the suggestions also contain questions, to reinforce this non-authoritarian personality. The Gardening Consultant understands the user's problems in CoScripter about as much as a teacher gardener understands problems in a student gardener's garden: a lot in general, but not that much about that particular student's soil, neighboring plants, resident insects, etc.

For example, when the user starts with a new, blank script, the Gardening Consultant icon in Fig. 3 appears. The reason it appears is because the Gardening Consultant 'knows' that a user facing a blank screen might be having trouble even getting started, and has a suggestion about that, if the user is interested. Clicking on the icon will reveal a suggestion for how to succeed, such as the one in Fig. 4. Figures 5–7 show other suggestions the user can access. Each of these suggestions is tailored to the user's current context, using mechanisms to be explained in Section 5.3.

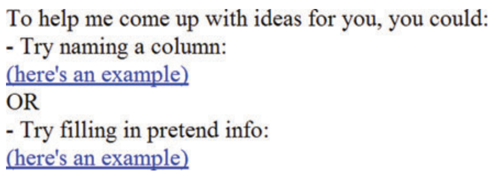
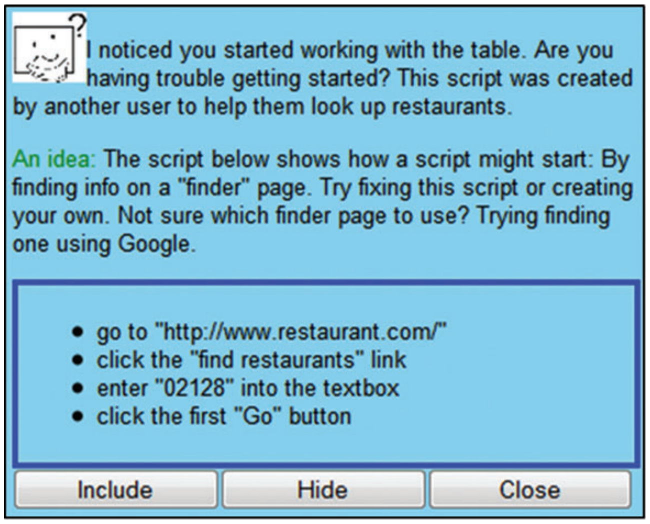

To help me come up with ideas for you, you could:
- Try naming a column:
(here's an example)
OR
- Try filling in pretend info:
(here's an example)

Figure 4. Start-with-a-column-name: This suggestion targets the How-to-Start barrier. The user can view it by clicking a special 'ideas cell' in the CoScripter table area (not shown). The idea is to nudge the user into the beginning step of working backward from the ultimate goal; the next suggestion, Fig. 5, encourages the next step.


I noticed you started working with the table. Are you having trouble getting started? This script was created by another user to help them look up restaurants.

An idea: The script below shows how a script might start: By finding info on a "finder" page. Try fixing this script or creating your own. Not sure which finder page to use? Trying finding one using Google.

- go to "http://www.restaurant.com/"
- click the "find restaurants" link
- enter "02128" into the textbox
- click the first "Go" button

Figure 5. Finder-page: If the user names a column with empty cells and clicks or hovers on the Gardening Consultant icon in the script, the Gardening Consultant suggests examples of a design pattern, which we call the Finder pattern, to populate the column.

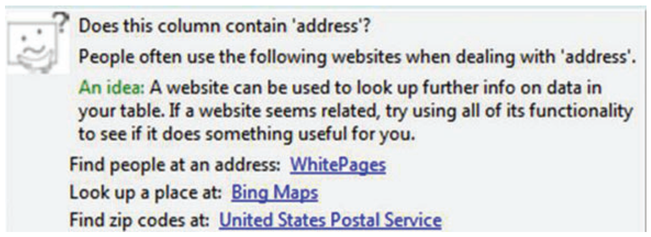

Does this column contain 'address'?
People often use the following websites when dealing with 'address'.
An idea: A website can be used to look up further info on data in your table. If a website seems related, try using all of its functionality to see if it does something useful for you.
Find people at an address: WhitePages
Look up a place at: Bing Maps
Find zip codes at: United States Postal Service

Figure 6. Compute-value-with-web: When the user populates a column, the Idea Garden infers the data type of the column and looks up a list of web pages that take the data type to compute/display new values. The user can view this suggestion by hovering over the table's special 'ideas cell' (not shown). The suggestion's goal is not to produce the perfect web page, but rather to help users think about websites as computational tools that can compute an answer on demand (such as distances or currency conversions).

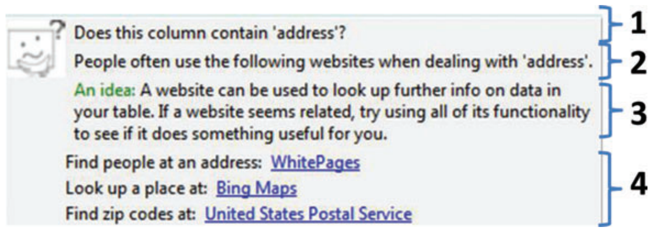
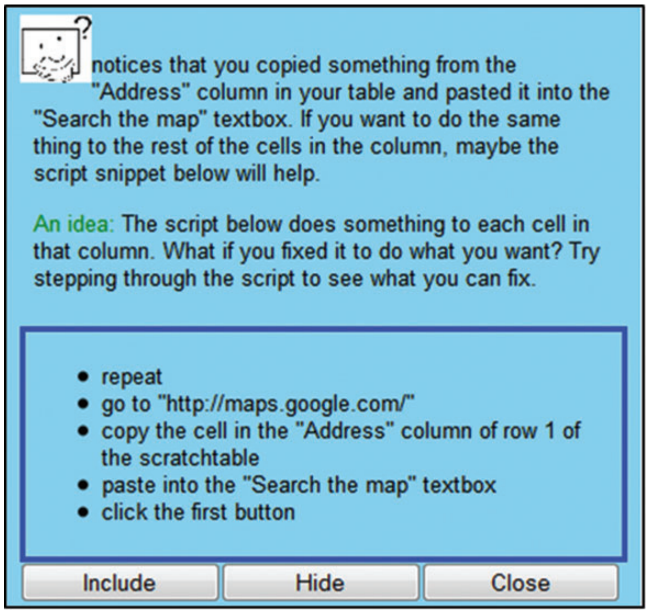

Does this column contain 'address'?
People often use the following websites when dealing with 'address'.
An idea: A website can be used to look up further info on data in your table. If a website seems related, try using all of its functionality to see if it does something useful for you.
Find people at an address: WhitePages
Look up a place at: Bing Maps
Find zip codes at: United States Postal Service

Figure 8. Suggestion structure. (1) The Gardening Consultant 'wonders' about context and (2) comments on this context to provide rationale for concrete examples in 4. (3) The Gardening Consultant summarizes the gist/essence of the idea and (4) suggests concrete examples as action items. (Start-with-a-column-name (Fig. 4) does not include 1 and 2 because the user has already told the system their context by the time he/she invokes this suggestion.)


notices that you copied something from the "Address" column in your table and pasted it into the "Search the map" textbox. If you want to do the same thing to the rest of the cells in the column, maybe the script snippet below will help.

An idea: The script below does something to each cell in that column. What if you fixed it to do what you want? Try stepping through the script to see what you can fix.

- repeat
- go to "http://maps.google.com/"
- copy the cell in the "Address" column of row 1 of the scratchtable
- paste into the "Search the map" textbox
- click the first button

Figure 7. Generalize-with-repeat: When a user has copied and pasted one cell of a column into a web page to retrieve a value, the last recorded line of the script displays the Gardening Consultant icon. The user can click it to view the suggestion targeting the More-than-Once barrier. The script example in this suggestion is incomplete so that the user must actively engage with it by editing code.

5.2. Suggestion form and content

All suggestions follow a template, as per Table 1: FC1. For example, the template used in our initial prototype is shown in Fig. 8. Item 4 of that template also satisfies a content constraint in Table 1, namely CC3, which says that any code examples must be intentionally imperfect matches to the user's needs. As an example, a user who is looking for apartments is unlikely to find WhitePages directly useful but the idea of looking up additional information based on what the user already has in his/her table (here, looking up people living at an address through WhitePages) is applicable across tasks.

Using this template, the initial Idea Garden prototype offers the suggestions enumerated in the column heads in Table 2. The top section of the table maps each suggestion to the barriers it targets.

The remaining sections of Table 2 show which suggestions deliver each programming concept, design pattern and problem-solving strategy (as per Table 1: CC1 and CC2) in our prototype. Future prototypes could support additional concepts, patterns, and strategies. We describe each of these content types below.

Programming concepts content. By *programming concepts*, we do not mean syntax. The Idea Garden's suggestions are intended to fill conceptual, 'beyond-syntax' gaps in programming knowledge. These gaps involve fundamental programming concepts, such as gaps in notions of input/output, dataflow and iteration (Cao *et al.*, 2010b, 2011; Ko *et al.*, 2004; Zang and Rosson, 2009). In the context of CoScripter, these concepts arise in the guise of retrieving data from web pages, passing data between tables and web pages, and iterating through all the rows of a table. For example, the suggestion in Fig. 7 introduces the concept of iteration.

Design pattern content. Generally, *patterns* are reusable solutions to the common design problem (Alexander, 1979), and their use has become particularly popular in software design (Gamma *et al.*, 1995). This first instantiation of the Idea Garden aims to assist users in applying two particular patterns: Finder and Repeat-Copy-Paste. The Finder pattern solves the design problem of how to pull and store data from a web page, so that data can be subsequently processed. Following the pattern, a CoScripter programmer demonstrates and records loading a web page and then (bulk) copies a list from the page to populate a blank table. The Repeat-Copy-Paste pattern solves the design problem of how to automatically push a table of data into a web form. Following this pattern, a CoScripter programmer writes script code such that, for each row in the table, the script copy/pastes values from table to the web form and submits the

Table 2. The Idea Garden prototype for CoScripter provides suggestions that target three barriers.

		Suggestions			
		Start-with-a-column-name (Fig. 4)	Finder-page (Fig. 5)	Compute-value-with-web (Fig. 6)	Generalize-with-repeat (Fig. 7)
Barriers targeted:	How-to-Start	X	X		
	Composition			X	
	More-than-once				X
Programming concepts reflected in suggestions					
Web page retrieval	Retrieve web page (get from URL and/or post web form)		X		
Dataflow	Copy data between web page and table, or between web pages			X	
Iteration	Loop through rows of table, operating on each row				X
Design patterns reflected in suggestions					
Finder	Load web page and (bulk) copy a list to populate a blank table		X		
Repeat-copy-paste	For each row, copy-paste value from table to web page and submit				X
Problem-solving strategies reflected in suggestions					
Work backward	Identify specific goal, then figure out how to achieve that goal	X	X		
Analogy	Find the solution to a similar problem, and then adapt it for problem at hand		X	X	
Generalization	Solve the general class of problems, of which the problem is one instance				X

We designed suggestions with the intent of helping users with three programming concepts, two design patterns and three problem-solving strategies. The programming concepts, design patterns and problem-solving strategies fulfill the content requirement of the Idea Garden design concept as shown in Table 1. See Figs 4–7 for screenshots.

form. Thus, our design patterns represent common ways that users structure their scripts to solve problems, a notion similar to ‘programming plans’ (Soloway and Ehrlich, 1984). Prior research has not identified a full range of patterns commonly used by CoScripter users, but we have personally found the two patterns above to be helpful when creating CoScripter mashups. Therefore, we have chosen these two for our initial prototype.

Problem-solving strategies content. The Gardening Consultant’s suggestions attempt to help users adopt problem-solving strategies that we identified from the problem-solving literature (Levine, 1994; Polya, 1973; Wickelgren, 1974). We chose common strategies that appeared in multiple sources. The Idea Garden’s suggestions try to make these strategies concrete. For example, the working backward strategy involves starting with a specific goal, then figuring out the step needed before that goal, the one before that and so on. If the user consults the Gardening Consultant on how to start the task, the Gardening Consultant will nudge the user toward this strategy by suggesting that the user work with the table’s end result first, by naming a table column (Fig. 4). The Gardening Consultant then suggests the next-to-last step (Fig. 5), and so on.

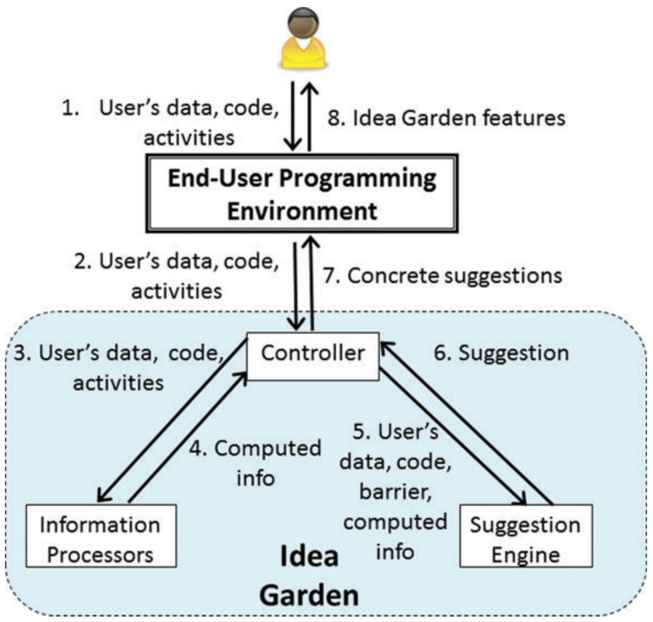
Finally, as per the content requirement of Table 1: CC3, the Idea Garden never tries to *entirely* solve users' problems with its suggestions. Rather, it tries to fill the conceptual part of the gap, and to nudge users into actively applying new programming knowledge and problem-solving strategies to complete the task at hand. For example, an Idea Garden suggestion usually describes how to solve parts of a *related* problem. This is by design: creativity theory posits that an essential part of arriving at good ideas is gaining the ability to elaborate or adapt existing ideas (Guilford, 1968). By doing so, the user engages in an act of 'association' which is a way of enhancing ideational fluency (Osborn, 1963). The hoped-for effect is that seeding users' efforts with starter ideas will encourage them to elaborate and adapt the suggested ideas to form new ideas toward solving parts of the problem at hand.

5.3. Behind the scenes: architecture and a walk-through of constructing a suggestion

The Idea Garden issues suggestions that provide 'relevant' examples with action items tailored to the user's context. To illustrate how the Idea Garden accomplishes this, we describe the architecture of the Idea Garden and the construction of an example of a Compute-value-with-web suggestion (Fig. 6) that the initial prototype can provide.

As mentioned in Section 3, the Idea Garden is an extension of a host environment and its host-independent architecture is depicted in Fig. 9. As Fig. 9 shows, the relationship between the Idea Garden and its host programming environment allows the Idea Garden to contextualize its suggestions and make them available to users of the host environment. On a high level, the process is as follows: the host environment, e.g. CoScripter, provides the Idea Garden with a stream of information about what the user is doing, including the user's code, data and recent activities. Using this contextual information, Idea Garden constructs context-appropriate suggestions for the user (primarily via off-the-shelf components). Finally, the Idea Garden makes the availability of new suggestions apparent in the host environment unobtrusively so that the user can view them at convenient times, if desired.

In this illustrative scenario imagine a user named Grace who, like the users in Study 1, must create a script that finds apartments near her campus. When Grace starts CoScripter, the Idea Garden automatically starts too, and it logs Grace's activities as she works on her script. Suppose that Grace searches Google for apartment websites that contain information about driving time to campus. Grace might click on several of these apartment websites, but perhaps none of them has driving information. So, on the apartment website that seems most relevant to her so far, she would select the apartment names and their addresses, then click the CoScripter button to import these addresses into a table. All of Grace's actions, as well as the data in her table and the code so far in her script (see Fig. 9, Edge 1) would be passed along to the Idea Garden Controller (Edge 2) from CoScripter.



1. User's data, code, activities
8. Idea Garden features
End-User Programming Environment
2. User's data, code, activities
7. Concrete suggestions
3. User's data, code, activities
Controller
6. Suggestion
4. Computed info
5. User's data, code, barrier, computed info
Information Processors
Suggestion Engine
Idea Garden

Figure 9. Architecture for the Idea Garden.

Using this information, in Edges 1–2, the Idea Garden's Controller would need to issue relevant suggestions. To accomplish this, the Idea Garden Controller passes the current tabular data to the Information Processors (Edge 3), which return additional information about the existing data (Edge 4). This information could include, for example, information about how many cells in each column are filled, or the inferred datatype of each column in the existing tabular data. The Idea Garden Controller would then send all of the available data and information about that data to the Suggestion Engine (Edge 5), which would be responsible for returning suggestions in template form back to the Controller (Edge 6). The Controller would fill in the templates to contextualize them before showing them to the user.

In the case of our illustrative scenario, the most useful information provided back by the Information Processors would be information about the datatypes of data in the table. To infer datatypes, our current prototype uses TopeDepot (Scaffidi, 2010), an environment-independent tool that infers data types, as one of its Information Processors. TopeDepot responds with a ranked list of possible data types for each column. The Controller uses the top guess. For our hypothetical user Grace, cell values like '2645 Hard Road Columbus, OH 43235' in a column, TopeDepot would likely report that the column contains 'address' data.

Having information about the columns in the data would make it possible for the Suggestion Engine to select template suggestions appropriate for a given situation. In particular, in this scenario, the Suggestion Engine would be configured so that when the user's table contains data of type 'address', then the Engine should issue a Compute-value-with-web suggestion (Fig. 6) along the lines of 'People often use the following websites when dealing with address…'.

An Idea Garden creator must configure the Suggestion Engine with rules for generating appropriate suggestions in different situations. We have found that this process was straightforward for the CoScripter prototype. First, we considered what common tasks people would want to perform with CoScripter (based on our review of the literature about what users were doing with CoScripter, e.g. Cypher *et al.*, 2010; Zang *et al.*, 2008), which not only helped identify their work processes but also the types of data that users would encounter. Secondly, we noted the barriers that people encountered (Section 4). Thirdly, we considered what state the user's data might possibly be in when the user encounters these barriers. For instance, in the case of not having an idea of how to even start (How-to-Start), the user likely would have an empty table. Not understanding how to compose existing modules and functions to create a program (Composition) would imply that the table contained some data of particular types that we had identified based on user tasks (above). Finally, we configured the Suggestion Engine to follow the rules that we had identified.

To present suggestions to Grace, the prototype contextualizes suggestions and integrates them with the programming environment. For presentation, the Controller calls host APIs to create widgets to hold the suggestion, and hence, create the concrete suggestion for Grace (Edge 7). The host then shows the availability of the concrete suggestion by displaying the Gardening Consultant icon near Grace's table columns. If Grace becomes curious and hovers over the cell, the concrete suggestion in Fig. 6 appears.

Recommendations for creating an Idea Garden for a specific host environment can be found in Chapter 8 of (Cao, 2013).

6. STUDY 2: THE IDEA GARDEN MEETS END USERS

To investigate the effectiveness of our approach with our target audience and to guide refinements of our design of the Idea Garden's suggestions, we conducted an empirical study with the following research question: *When and how will the Idea Garden help—or not help—end-user programmers overcome their barriers?*

6.1. Participants

We recruited 15 participants (undergraduate, non-CS students with little to no programming experience) to create a script using CoScripter, supported by a prototype of the Idea Garden. We excluded from our analysis the data from six of the participants because those six did not encounter barriers (and thus did not generate data relevant to our research question). Thus, we analyzed the data from a total of nine participants.

6.2. Procedure

The study procedure consisted of a tutorial, scripting task and semi-structured interview at the end of the session.

The study took place over the course of several months, so that we could continually evaluate and refine our prototype. With the first five of the nine participants, we paired a paper prototype of the Idea Garden with an executable version of CoScripter. We transitioned to a fully integrated, executable system for the other four participants. (We will indicate which prototype each participant used in the Results section by identifying participants of the paper Idea Garden as 'Paper' and those of the fully executable prototype as 'Exe'.) Each participant used a newer prototype than the previous participant as the Idea Garden features continued to evolve. Figure 10 shows the paper prototype, and the previous sections have included screenshots of the executable prototype.

The tutorial was hands-on and showed participants how to create two CoScripter scripts: one to look up information from a web page, and a more complex script that mashed up information from two web pages using the table.

The main task was to create a script for finding 2-bedroom apartments under a certain price and within a certain walking time from the university. Participants talked aloud as they worked. To make interactions with the paper part of the Paper Idea Garden no more costly than interactions with the computer part, participants who used the paper versions were not allowed to use the computer keyboard and mouse. Instead, they told us the actions they wanted, and a researcher then carried them out.

Participants had an hour to complete the task and were given the scripts created in the tutorial for reference. If any participant fixated for more than 5 min on an unsuccessful approach, we gave him/her a hint to encourage trying another approach (i.e. 'How would you normally find apartments?' or 'Maybe try a different website'). This allowed us to gather data from subsequent parts of the task.

Finally, a semi-structured interview asked the following question about each suggestion: 'What did you think about this suggestion?' If necessary, we followed up with additional questions for clarification. We videotaped the sessions and collected the final scripts.

We collected video and audio recordings of the sessions that captured how the participants approached the task and interacted with the Idea Garden throughout as well as the interviews. Using a method similar to what we used in Study 1, we identified episodes where a participant encountered a barrier as indicated by, e.g. the participant turning to the Idea Garden for help or verbalizing a need for help. Unlike Study 1 in which we focused on any barriers, we focused our analysis only on the episodes where the participants encountered the barriers that our Idea Garden features targeted.

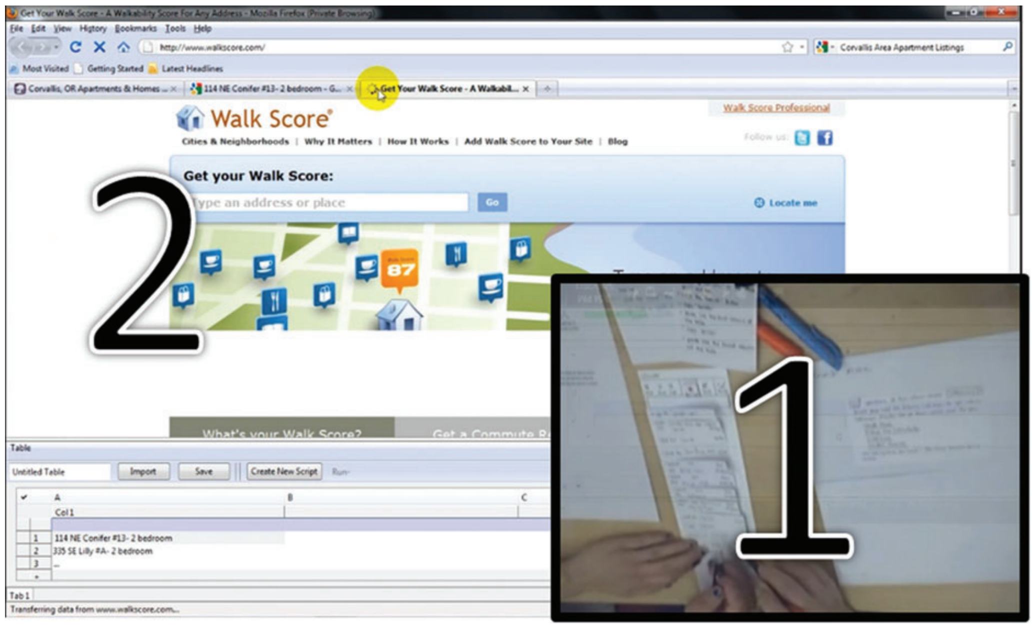


Figure 10. Example paper-prototype session with (1) a paper script and suggestions, and (2) a CoScripter table and webpage.

Table 3. Participants' interactions with suggestions and results.

Suggestion:	Start-with-a-column-name	Finder-page	Compute-value-with-web	Generalize-with-repeat	Steps complete
Barriers:	How-to-start		Composition	More-than-once	
Step numbers:	1	1	2	2,3	
Paper-F1	+	–			1
Paper-F2				+	1, 2
Paper-F3			+	+	1, 2
Paper-M1	–	–	–	+	1, 2
Paper-M2		–		+	1, 2
Exe-F1	–		–		1
Exe-F2	–		–	+	1, 2
Exe-F3	+	–	+	+	1, 2, 3
Exe-M1			+	+	1, 2

+: participant followed suggestion and made progress.
–: participant did not follow suggestion, or did not make progress from following it.

6.3. Results

To find out when and how the Idea Garden did and did not help our participants overcome their barriers, we counted the participants' interactions with the suggestions (i.e. they brought up the suggestion by hovering/clicking to see the content). These data showed that every participant interacted with 1–4 of the prototype's suggestions, for a total of 22 interactions (Table 3).

Using the videos, we then categorized each interaction with a progress criterion: Did the participant follow the suggestion and add something actually useful to their emerging mashup? According to this criterion, 12 interactions led to progress in accomplishing the task (interactions marked '+' in the table; interactions marked '–' did not satisfy the progress criterion). As Table 3 shows, eight participants made progress. These participants either were able to progress to the next step, or could make tangible progress within the current step, as illustrated by the examples and quotes in Section 6.3.1.

Participants' overall success in completing the final script is enumerated in the rightmost column of the table. The task required three steps. Step 1 was to get a list of apartments from a web page into the table, Step 2 was to use a web page to compute walking time for each apartment and Step 3 was to iterate over the addresses in the table rows and paste the walking times into another column. Steps 2 and 3 required participants to use the repeat command. As Table 3 shows, one participant completed all three steps, six completed the first two steps and two completed only one step.

6.3.1. *When and how the suggestions helped*

Helping with the How-to-Start Barrier. The How-to-Start barrier proved to be enough of a struggle that six participants turned to the Gardening Consultant. The suggestions targeting this barrier are Start-with-a-column-name and Finder-page. Finder-page was not helpful to anyone, but Start-with-a-column-name helped two of the participants. Specifically, Paper-F1 and Exe-F3 used Start-with-a-column-name as a springboard to getting apartment information into their tables. Recall that the Start-with-a-column-name encourages participants to use the problem-solving strategy of working backward. Both participants showed evidence of adopting this strategy.

For example, Paper-F1 immediately followed the *Start-with-a-column-name* suggestion after first encountering it; she named a blank column 'number of bedrooms'. Out of her own volition, she changed it to 'address', and then proceeded to naming the second column 'price' and the third column 'number of bedrooms'. At this point, she started asking herself what might be a possible website to retrieve housing information from:

Paper-F1 [min 14]: What is the website for searching [for a] house?

Not knowing which websites provided housing information, she made a website up:

Paper-F1 [min 14]: Maybe "houseForRent.com" or something?

Interestingly, even with this made-up website, she made progress. HouseForRent.com did indeed exist, and eventually led her to rent.com, where she found data that she used to populate her table. Thus, she succeeded by working backward, just as the suggestion was intended to promote.

Helping with the Composition Barrier. Participants encountered the Composition barrier when expanding scripts to pass data from the table into a new web page during Step 2, which was needed to compute the walking distance for each apartment. Three of the six participants who viewed Compute-value-with-web overcame this barrier. An interesting story was shared by the three participants (Paper-F3, Exe-F3 and Exe-M1). They first tried to accomplish Step 2 through the web page they used for Step 1, hoping to directly specify the walking distance from the university in their web queries, but this approach did not work because apartment search engines do not support that parameter.

Stymied at first, all three of these participants turned to the Gardening Consultant for a suggestion—and subsequently acted on the Gardening Consultant's idea of using websites to perform *calculations* (e.g. calculations that Bing Maps is willing to perform). After each of the three participants acted upon this idea, he/she began to make progress by effecting the flow of addresses from the apartment search pages to a map page, an application of the *dataflow* concept:

Exe-F3 [Interview]: "this [suggestion] was very helpful because this was what I used to go to, Bing Map, in order to find driving time and distance. So I thought it was very useful."

Generalizing to Overcome the More-than-Once barrier. The most successful suggestion was Generalize-with-repeat, which nudges the user into generalizing a single table manipulation across the rest of the rows by providing a script (albeit an imperfect one). Seven participants ran into the More-than-Once barrier and turned to this suggestion—and all seven benefited from it.

Six succeeded in editing the provided code to complete Step 2 and, in Exe-F3's case, Step 3 as well. Three participants even elaborated on the code—that is, by making an existing idea better by enhancing or expanding it. Such elaboration is a key aspect of both learning and creativity. For example, Paper-F2 experimented with *four* different ways of placing the suggested code within her own code, ultimately completing Step 2 by finding the correct placement.

Learning is characterized by the ability to transfer knowledge used in an early setting to a later setting, and two participants demonstrated that ability after having worked with the *Generalize-with-repeat* suggestion. Both Paper-F3 and Exe-F3 later created an additional repeat loop by themselves, without the help of the suggestion. In doing so, they demonstrated knowledge of the *iteration* concept as well as the *repeat-copy-paste* pattern.

Where did this knowledge come from? Our interpretation of why users overcame the barriers after interacting with the Idea Garden features is that they learned new strategies and gained new knowledge that the features aimed to convey. In these two clear cases, after they interacted with the Idea Garden, these users were able to solve similar problems on their own. Also, in a separate study, we directly measured effects of the Idea Garden on users' learning (Cao *et al.*, 2012). That study measured learning directly, and found that the Idea Garden helped nine out of ten users learn relevant problem-solving strategies, patterns and programming concepts.

6.3.2. *And when the suggestions did not help*

Not all suggestions were helpful. In seeking help for the *How-to-Start* barrier, Exe-F1 and Exe-F2 saw the *Start-with-a-column-name* suggestion but decided to ignore it. Fortunately, they were able to eventually overcome the barrier without assistance.

The suggestion with the lowest success was the *Finder-page*: every participant who read it decided to ignore it. All missed the relevance of the design pattern it was trying to convey to the task at hand:

Paper-M2 [Interview]: "I didn't need any restaurants and there was no 'apartments.com' that I was going to go to."

Users failing to see the pertinence of a suggestion also arose with the *Compute-value-with-web* suggestion:

Exe-F2 [Interview] "I wasn't sure how it was related to what I was doing… 'cause I wasn't looking for business ratings or jobs."

Exe-F1 also looked at the *Compute-value-with-web* suggestion; however, she looked at the suggestion only briefly and then dismissed it. The suggestion was designed to help with the second step of the task. And she even looked at the suggestion at the ideal moment: when she had just completed the first step (i.e. gathering an initial list of apartments). But she apparently did not see the suggestion's relevance.

In the above cases, we (the designers of these suggestions) knew that the suggestions were relevant to the task at hand. Obviously, in at least some cases, the suggestions failed to convey their relevance.

7. TOWARD IDEA GARDEN 2.0

The results of Study 2 suggest several possibilities for the Idea Garden's future.

7.1. Using context to support a user's problem frame

Schön's concept of a problem frame (Schön, 1983) may help to explain why some participants failed to see the relevance of certain suggestions. According to Schön, when facing a 'messy' problem that is not well defined, people often impose a *frame* on the problem based on their own interpretations of what it entails. Thus, a frame is a boundary within which people work to solve the messy problem. *Framing*, the process of setting the frame, is in essence the problem solver's perceived definition of the problem to be solved.

For example, Exe-F1's remark as she dismissed the *Compute-value-with-web* suggestion implies that she framed the problem differently from the suggestion content's focus:

Exe-F1 [min 48]: You are just saying, like, if it has addresses or zip codes that are close to it [viewing the Compute-value-with-web suggestion], but I'm still just trying to write the script.

In this case, the Gardening Consultant's suggestion may have failed to help Exe-F1 because the suggestion made the underlying assumption that the user, having gotten this far, lacked the concept of computing with a web page. This problem might be avoided by having multiple suggestions available for each user context, each based on a different assumption about the user's current frame. For example, a different assumption about Exe-F1's frame would be that she was trying to create the script using multiple web pages. Thus, a suggestion based on that assumption might include an example snippet of script (e.g. similar to the snippet in Fig. 7 suggestions) that uses more than one web page together to solve a problem.

7.2. Understanding user behavior with attention investment

Because relevance to context was a pervasive theme for both the helpful and unhelpful occurrences of our suggestions, it is useful to consider relevance from the perspective of the model of Attention Investment (Blackwell, 2002). According to this model, users' perceptions of the benefits, costs and risks of pursuing a particular path predict the probability of their following that path. Perceived benefit seems likely to align with perceived relevance.

For example, recall that the *Generalize-with-repeat* suggestion was our most successful suggestion: All the participants who saw it engaged with it and made progress in their task. One possible reason for its success is that it became viewable in a circumstance that matched the user's current context very well. The suggestion not only mentioned a step that all participants were trying to complete (generalizing from one row to all rows in the table), but the suggestion's snippet of script also included actions the user had just performed, helping to convey the suggestion's relevance.

Attention Investment's cost and risk factors may also explain why participants favored the *Generalize-with-repeat* feature. The participants' perception of the cost (effort) and risk required to copy and then fix the feature's code snippet may have seemed lower than the effort required to write the script from scratch.

7.3. Balancing 'correct' and (deliberately) 'imperfect' aspects of suggestions

Participants' perceptions of a suggestion's relevance may also have been influenced by the Idea Garden's deliberate use of incomplete and/or imperfect suggestions.

By design, most of the Gardening Consultant's suggestions have 'correct' and 'imperfect' parts. The parts that are not problem-specific are correct in that they apply regardless of whether the user is, say, looking up restaurants versus apartments. Examples of such correct parts include the portions that embody design patterns and problem-solving strategies, which appear in the *gist* of the suggestion (recall Fig. 8). The imperfect parts of suggestions are so specific that they are unlikely to be *exactly* what the user needs. For example, the Finder-page's suggestion to use restaurants.com (Fig. 5) cannot be used to find apartments.

Unfortunately, the imperfect part of a suggestion sometimes dominated participants' perceptions of the suggestion's relevance. For example, Exe-F2 said that the *Compute-value-with-web* suggestion (Fig. 6) was not relevant because she was not looking for the types of things to which the suggestion referred (i.e. people, places on a map and zip codes). Likewise, recall that Paper-M2 decided that the *Finder-page* suggestion was not relevant because 'it said restaurants' (instead of apartments). Here, both participants rejected the suggestions based on the specific examples, and did not see how the suggestions might be relevant to them.

The ability to apply a *schema* during problem solving has been shown to distinguish experts from novices (Sweller, 1988). A schema is a structure that allows problem solvers to recognize a problem state as belonging to a particular category of problem states that normally require particular moves. Experts possessing schemas are able to categorize problems according to those schemas, whereas novices without schemas tend to resort to surface structures when classifying problems. In our study, participants Exe-F2 and Paper-M2 (who, like all our participants, were novice programmers) seemed to lack schemas, focusing on surface-level details such as 'restaurant' or 'jobs', even though the schema (here, the notion of a Finder pattern) was explicitly given in suggestions (Fig. 5).

Studies of information foraging provide another explanation for why users fixate on details. They observed that for web users engaged in information seeking, specific wording has stronger 'scent' (ability to attract information seekers' attention) than general wording (Spool *et al.*, 2004). In our study, participant Paper-F3 made foraging decisions based on specific words in a suggestion:

Paper-F3 [min 27]: The reason why I chose 'Walkscore[.com]' was because it had the word 'walk' in it and I'm trying to find walking distance.

Thus, although the (deliberately) concrete suggestion aimed at encouraging problem solving by analogy, the participant was unable to see past the concrete details.

To mitigate the problem of users not seeing relevance of the concrete parts of the suggestions, the Idea Garden may need to help users set the right expectation. This suggests a need to make clear that these concrete examples are not meant to be used 'as-is'. Rather, the examples show a *way* of approaching part of a programming task that might be useful to the users. To give users more guidance, the Idea Garden might also suggest alternative actions for users to follow if they find an example irrelevant, e.g. 'You can search for a web site that computes the information you need'.

We have already begun to make some of these improvements. The improved Idea Garden is presented in Cao (2013) and has been evaluated in two follow-up studies focusing on the effects of the Idea Garden on users' learning transfer (Cao *et al.*, 2012, 2013). The studies showed that the Idea Garden not only helped users overcome programming barriers but also helped them learn the relevant programming knowledge and problem-solving strategies that they applied in solving new programming tasks. In addition to demonstrating the generalizability of the Idea Garden across different programming tasks, we showed the generalizability of its software architecture by implementing a second instantiation in the Gidget end-user programming environment (Cao, 2013).

8. CONCLUSION

In this paper, we have presented the Idea Garden, a novel approach for helping end-user programmers generate new ideas and problem-solve when they run into programming barriers. The Idea Garden's design was informed by our empirical observations of end-user mashup programmers from Study 1 and by theories from the literature, such as those on problem solving and MLT. The Idea Garden is different from mixed-initiative approaches like Microsoft's Clippy, because in the Idea Garden, all initiative and control belongs to the user, and the Idea Garden never interrupts. The Idea Garden also is not a replacement for online tutorials. Rather, it supplements such materials with scaffolding for problem-solving strategies and programming knowledge in the context of users' actual tasks. Finally, and most important, the Idea Garden is not an automatic problem solver. Instead, it steers users toward learning to solve programming problems themselves.

To explore and refine the Idea Garden approach, we prototyped it in the CoScripter programming environment. Our prototype targets three barriers identified by the end-user programming literature: *How-to-Start*, *Composition* and *More-than-Once*. To help users overcome these barriers, the features include suggestions that support (1) three problem-solving strategies, (2) two design patterns and (3) three programming concepts.

Study 2 provided empirical evidence that the Idea Garden helped end-user programmers apply several of the strategies/concepts targeted by the approach. In particular, we observed instances where participants interacted with the Idea Garden and subsequently demonstrated the ability to apply one or more of the following:

(i) the generalization and work backward problem-solving strategies,
(ii) the repeat-copy-paste design pattern and
(iii) the dataflow and iteration programming concepts.

Although not all the Idea Garden's suggestions worked equally well, eight out of nine participants benefited from at least one of them. Furthermore, our participants overcame instances of *all three* types of barriers targeted. Although there is much room for improvement in future work on the Idea Garden, these results suggest that Idea Garden to be a promising approach in helping end-user programmers incrementally build their skills, just in time, to better solve problems with their programs on their own:

Exe-F3: [The Idea Garden] definitely was useful… for someone who's utilizing the program [CoScripter]—maybe by themselves.

ACKNOWLEDGEMENTS

We thank William Curran for helping with the paper prototype study, and Forrest Bice and Hannah Adams for helping with the implementation of the prototype.

FUNDING

This work was supported in part by NSF grants IIS-0917366, CNS-1240957, and IIS-1314384.

REFERENCES

Ackerman, M.S. and McDonald, D.W.(1996) Answer Garden 2: Merging Organizational Memory with Collaborative Help. Proc. ACM Conf. on Computer Supported Cooperative Work, pp. 97–105. Boston, MA.

Alexander, C. (1979) The Timeless Way of Building. Oxford University Press, New York.

Andersen, R. and Mørch, A. (2009) Mutual development: a case study in customer-initiated software product development. In End-User Development, pp. 31–49. Springer, Heidelberg.

Bandura, A. (1977) Self-efficacy: toward a unifying theory of behavioral change. Psycholog. Rev., 84, 191–215.

Blackwell, A. (2002) First Steps in Programming: A Rationale for Attention Investment Models. Proc. IEEE Symposium on Human Centric Computing Languages and Environments, pp. 2–10. Arlington, VA.

Blackwell, A. and Hague, R. (2001) AutoHAN: An Architecture for Programming the Home. Proc. IEEE Symposium on Human-Centric Computing Languages and Environments, pp. 150–157. Stresa, Italy.

Bloom, B. and Broder, L. (1950) Problem-Solving Processes of College Students. Supplementary Educational Monographs. University of Chicago Press.

Brandt, J., Dontcheva, M., Weskamp, M. and Klemmer, S. (2010) Example-Centric Programming: Integrating Web Search into the Development Environment. Proc. 28th ACM Conf. on Human Factors in Computing Systems (CHI), pp. 513–522. Atlanta, GA.

Cao, J. (2013) Helping enduser programmers help themselves: the Idea Garden approach. PhD Thesis, Oregon State University. http://hdl.handle.net/1957/38561.

Cao, J., Riche, Y., Wiedenbeck, S., Burnett, M. and Grigoreanu, V. (2010a) End-User Mashup Programming: Through the Design Lens. Proc. 28th ACM Conf. on Human Factors in Computing Systems (CHI), pp. 1009–1018. Atlanta, GA.

Cao, J., Rector, K., Park, T., Fleming, S., Burnett, M. and Wiedenbeck, S. (2010b) A Debugging Perspective on End-User Mashup Programming. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 149–156. Madrid, Spain.

Cao, J., Fleming, S. and Burnett, M. (2011) An Exploration of Design Opportunities for 'Gardening' End-User Programmers' Ideas. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 35–42. Pittsburgh, PA.

Cao, J., Kwan, I., White, R., Fleming, S., Burnett, M. and Scaffidi, C. (2012) From Barriers to Learning in the Idea Garden: An Empirical Study. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 59–66. Innsbruck, Austria.

Cao, J., Kwan, I., Bahmani, F., Burnett, M., Fleming, S., Jordahl, J., Horvath, A. and Yang, S. (2013) End-User Programmers in Trouble: Can the Idea Garden Help Them to Help Themselves? Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 151–158. San Jose, CA.

Carroll, J. (1990) The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill. MIT Press, Cambridge, MA.

Carroll, J. (ed.) (1998) Minimalism Beyond the Nurnberg Funnel. MIT Press, Cambridge, MA.

Carroll, J. and Rosson, C. (1987) The Paradox of the Active User. In Interfacing Thought: Cognitive Aspects of Human–Computer Interaction, pp. 26–28. MIT Press, Cambridge, MA.

Compeau, D. and Higgins, C. (1995) Computer self-efficacy: development of a measure and initial test. MIS Q., 19, 189–211.

Costabile, M., Mussio, P., Provenza, L. and Piccinno, A. (2009) Supporting end Users to be Co-Designers of their Tools. In End-User Development, pp. 70–85. Springer, Heidelberg.

Cypher, A., Nichols, J., Dontcheva, M. and Lau, T. (2010) No Code Required: Giving Users Tools To Transform the Web. Morgan Kaufmann, Los Altos, CA.

Díaz, P., Aedo, I., Rosson, M., Carroll, J. (2010) A visual tool for using design patterns as pattern languages, ACM International Working Conference on Advanced Visual Interfaces (AVI), pp. 67–74.

Dorn, B. (2011) ScriptABLE: Supporting Informal Learning with Cases. ACM International Computing Education Research Conference (ICER), pp. 69–76. Providence, Rhode Island.

Fischer, G. (2009) End-User Development and Meta-Design: Foundations for Cultures of Participation. In End-User Development, pp. 3–14. Springer, Berlin.

Flavell, J. (1979) Metacognition and cognitive monitoring. Am. Psychol., 34, 906–911.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Gantt, M. and Nardi, B. (1992) Gardeners and Gurus: Patterns of Cooperation Among CAD Users. Proc. Conf. on Human Factors in Computing Systems (CHI), pp. 107–118. Monterey, CA.

Gross, P. and Kelleher, C. (2010) Non-programmers identifying functionality in unfamiliar code: strategies and barriers. J. Vis. Lang. Comput., 21, 263–276.

Gross, P., Herstand, M., Hodges, J. and Kelleher, C. (2010) A Code Reuse Interface for Non-Programmer Middle School Students. Proc. 15th Int. Conf. on Intelligent User Interfaces, pp. 219–228. Island of Madeira, Portugal.

Guilford, J. (1968) Intelligence, Creativity, and Their Educational Implications. RR Knapp, San Diego.

Guzdial, M. (2008) Education: paving the way for computational thinking. Commun. ACM, 51, 25–27.

Hundhausen, C.D., Farley, S.F. and Brown, J.L. (2009) Can direct manipulation lower the barriers to computer programming and promote transfer of training? An experimental study. ACM Trans. Comput.-Hum. Inter., 16, 1–40.

Jansson, D. and Smith, S. (1991) Design fixation. Des. Stud., 12, 3–11.

Kelleher, C. and Pausch, R. (2006) Lessons Learned From Designing a Programming System to Support Middle School Girls Creating Animated Stories. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 165–172. Brighton, UK.

Ko, A., Myers, B. and Aung, H. (2004) Six Learning Barriers in End-User Programming Systems. Proc. IEEE Symposium on Visual Languages and Human Centric Computing, pp. 199–206. Rome, Italy.

Lee, M. and Ko, A. (2011) Personifying Programming tool Feedback Improves Novice Programmers' Learning. Proc. 7th Int. Workshop on Computing Education Research, pp. 109–116. Providence, Rhode Island.

Lee, M., Ko, A., and Kwan, I. (2013). In-Game Assessments Increase Novice Programmers' Engagement and Level Completion Speed. Proc. ACM ICER, pp. 153–160. San Diego, CA.

Levine, M. (1994) Effective Problem Solving. Prentice Hall, Englewood Cliffs, NJ.

Lewis, S., Dontcheva, M. and Gerber, E. (2011) Affective Computational Priming and Creativity. Proc. ACM Conf. on Human Factors in Computing Systems (CHI), pp. 735–744. Vancouver, BC.

Lieberman, H. (2001) Your Wish Is My Command: Programming By Example. Morgan Kaufmann, Los Altos, CA.

Lin, J., Wong, J., Nichols, J., Cypher, A. and Lau, T. (2009) End-User Programming of Mashups with Vegemite. Proc. ACM Int. Conf. on Intelligent User Interfaces, pp. 97–106. Island of Madeira, Portugal.

Little, G., Lau, T., Cypher, A., Lin, J., Haber, E. and Kandogan, E. (2007) Koala: Capture, Share, Automate, Personalize Business Processes on the Web. Proc. ACM Conf. on Human Factors in Computing Systems (CHI), pp. 943–946. San Jose, CA.

Nass, C. and Moon, Y. (2000) Machines and mindlessness: social responses to computers. J. Soc. Issues, 56, 81–103.

Newman, M., Lin, J., Hong, J. and Landay, J. (2003) DENIM: an informal web site design tool inspired by observations of practice. Hum.-Comput. Inter., 18, 259–324.

McFarlane, D. (2002) Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. Hum.-Comput. Inter., 17, 63–139.

van der Meij, H. and Carroll, J. (1998). Principles and Heuristics for Designing Minimalist Instruction. In Carroll, J. (ed.), Minimalism Beyond the Nurnberg Funnel, pp. 19–53. MIT Press, Cambridge, MA.

Myers, B., Pane, J. and Ko, A. (2004) Natural programming languages and environments. Commun. ACM, 47, 47–52.

Oney, S. and Myers, B. (2009) FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 105–108. Corvallis, Oregon.

Osborn, A. (1963) Applied Imagination: Principles and Procedures of Creative Problem-Solving. Scribner, New York, NY.

Pane, J. and Myers, B. (2006) More Natural Programming Languages and Environments. In End User Development, pp. 31–50. Springer, Berlin.

Polya, G. (1973) How To Solve It: A New Aspect of Mathematical Method. Princeton University Press, Princeton, NJ.

Repenning, A. and Ioannidou, A. (2008) Broadening participation through scalable game design. ACM SIGCSE Bull., 40, 305–309.

Robertson, T., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J., Beckwith, L. and Phalgune, A. (2004) Impact of Interruption Style on End-User Debugging. Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI), pp. 287–294. Vienna, Austria.

Scaffidi, C. (2010) Sharing, finding and reusing end-user code for reformatting and validating data. J. Vis. Lang. Comput., 21, 230–245.

Scaffidi, C., Shaw, M. and Myers, B. (2005) Estimating the Numbers of End Users and end User Programmers. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 207–214. Dallas, Texas.

Scaffidi, C., Cypher, A., Elbaum, S., Koesnandar, A. and Myers, B. (2008) Using scenario-based requirements to direct research on web macro tools. J. Vis. Lang. Comput., 19, 485–498.

Scaffidi, C., Bogart, C., Burnett, M., Cypher, A., Myers, B. and Shaw, M. (2009) Predicting Reuse of End-User Web Macro Scripts. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 93–100. Corvallis, Oregon.

Schön, D. (1983) The Reflective Practitioner: How Professionals Think in Action. Basic Books, New York.

Seaman, C.H. (2008) Qualitative Methods. In Shull, F., Singer, J. and Sjøberg, D.I.K. (eds) Guide to Advanced Software Engineering. Springer, Berlin.

Simon, H.A. (1980). Problem Solving and Education. Problem Solving and Education: Issues in Teaching and Research, pp. 81–96. Lawrence Erlbaum, London.

Spool, J., Perfetti, C. and Brittan, D. (2004) Designing for the Scent of Information. User Interface Engineering, North Andover, MA.

Soloway, E. and Ehrlich, K. (1984) Empirical study of programming knowledge. IEEE Trans. Softw. Eng., 10, 595–609.

Sweller, J. (1988) Cognitive load during problem solving: Effects on learning. Cogn. Sci., 12, 257–285.

Wickelgren, W. (1974) How To Solve Problems: Elements of a Theory of Problems and Problem Solving. WH Freeman, San Francisco.

Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., and Rothermel, G. (2003) Harnessing Curiosity to Increase Correctness in End-User Programming. Proc. ACM Conf. on Human Factors in Computing Systems (CHI), pp. 305–312.

Wong, J. and Hong, J. I. (2007) Making Mashups with Marmite: Towards End-User Programming for the Web. Proc. ACM Conf. on Human Factors in Computing Systems, pp. 1435–1444. San Jose, CA.

Zang, N. and Rosson, M. (2009) Playing with Information: How End Users Think about and Integrate Dynamic Data. Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 85–92. Corvallis, Oregon.

Zang, N., Rosson, M. and Nasser, V. (2008) Mashups: Who? What? Why? Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)—Extended Abstracts, pp. 3171–3176. Florence, Italy.

APPENDIX

The computer self-efficacy questionnaire used in Study 1. Our questionnaire was based on Compeau and Higgins' questionnaire (Compeau and Higgins, 1995) and adapted to the task of script creation. For each question, Strongly Disagree corresponds to a score of 1, whereas Strongly Agree corresponds to a score of 5. We calculated a participant's self-efficacy score by taking the average of the ten scores.

The following questions ask you to indicate whether you could use an environment for creating scripts (that automate daily tasks you perform on the web) under a variety of conditions. For each of the conditions please indicate whether you think you would be able to complete the job using the environment.

Given a description of what a script should do, I could figure out how to create the script:

… if there was no one around to tell me what to do as I go	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if I had never seen a script like it before	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if I had only the software manuals for references	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if I had seen someone else using it before trying it myself	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if I could call someone for help if I got stuck	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if someone else had helped me get started	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if I had a lot of time to complete the task	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if I had just the built-in help facility for assistance	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if someone showed me how to do it first	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree
… if I had used similar environments before this one to do this same task	Strongly Disagree	Disagree	Neither Agree	Nor Disagree	Agree	Strongly Agree