AN ABSTRACT OF THE DISSERTATION OF

Jill Cao (Chen Cao) for the degree of Doctor of Philosophy in Computer Science presented on April 24, 2013.

Title: Helping End-User Programmers Help Themselves – The Idea Garden Approach

Abstract approved:

_____

Margaret M. Burnett

End-user programmers face many barriers in programming. Research has seen many programming environments that attempted to lower or remove the barriers but despite these efforts, empirical studies continue to report barriers users face. To investigate this issue, we took a theory-informed approach. Using theories from design, creativity, and problem solving as a lens, we examined end-user programmers' programming obstacles to derive design implications. Synthesizing the implications, we proposed an Idea Garden approach for creating problem-solving support in existing end-user programming environments aimed at *helping users help themselves*. This approach focuses on delivering problem-solving strategies and programming knowledge in the context of users' work to help them overcome barriers. We developed a proof-of-concept prototype of an Idea Garden for the CoScripter environment. Results from empirical studies of the prototype were encouraging: not only was the Idea Garden able to help users overcome barriers, learn relevant programming and strategies, but such learning persisted with users so that they were able to apply it toward problem-solving new tasks without further help from the Idea Garden. We conclude by providing recommendations to researchers who are interested in developing an Idea Garden for their end-user programming environments.

Helping End-User Programmers Help Themselves – The Idea Garden Approach

by
Jill Cao (Chen Cao)

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented April 24, 2013
Commencement June 2013

Doctor of Philosophy thesis of Jill Cao (Chen Cao) presented on April 24, 2013.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Jill Cao (Chen Cao), Author

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude toward my advisor, Margaret Burnett, who has continuously inspired me with her wisdom and her permanent positive attitude. You have made so many positive influences on me which have helped me grow both as a researcher and as a person. I cannot thank you enough.

I would like to thank Scott D. Fleming, Irwin Kwan, Christopher Scaffidi, Allen Cypher, Susan Wiedenbeck, and Yann Riche for their feedback and support. Your expertise and insights have helped me shape my work in many ways.

I have also had the pleasure to work closely with Faezeh Bahmani, Hannah Adams, Rachel White, Forrest Bice, Valentina Grigoreanu, Kyle Rector, and Thomas H. Park. Your efforts and input have been wonderful.

Mom and Dad, thank you for encouraging me to pursue my dream and always have my back.

To Allan, my love, thank you for making me happy every day and happier the next.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF APPENDICES

Helping End-User Programmers Help Themselves – The Idea Garden Approach

Chapter 1. Introduction

Today, there are hundreds and thousands of users building computer programs to facilitate their work or hobbies. According to Nardi, these users are called *end-user programmers*, people who program to complement a primary task (Nardi, 1993). One common example is an accountant creating spreadsheet formulas to keep track of budgets to avoid manual calculations. Another example is a web-savvy college student programming a script to automate the process of finding apartments close to campus. In fact, programming has become so widespread that according to the U. S. Bureau of Labor and Statistics, the number of people using spreadsheets and databases at work reached 55 million by the end of 2012—an order of magnitude greater than the number of professional programmers (Scaffidi, Shaw, & Myers, 2005).

1.1 Background and Motivation

Evidence shows that end-user programmers continue to be challenged by programming barriers they encounter. For example, Ko et al. identified six learning barriers faced by end-user programmers in using Visual Basic (Ko, Myers, & Aung, 2004). These barriers included design barriers (e.g., how to create a sorting algorithm), selection barriers (i.e., how to choose among multiple modules or APIs), use barriers (i.e., when an API is chosen, how to utilize its functionality), and coordination barriers (i.e., how to orchestrate multiple APIs). In the realm of spreadsheets, Chambers and Scaffidi found that spreadsheet users face barriers similar to those faced by Visual Basic learners as identified by Ko et al. (Chambers & Scaffidi, 2010). Kissinger et al. identified information needs of end users debugging spreadsheets highlighting information gaps in debugging strategies (Kissinger, et al., 2006). In animation, Gross and Kelleher's study found that users faced barriers in identifying functionality in unfamiliar code when attempting to reuse the code (Gross & Kelleher, 2009). In mashups, Zang and Rosson found that users had trouble understanding structured data when asked to create a mashup in Yahoo!Pipes (Zang & Rosson, 2009).

Perhaps it is not a surprise that end-user programmers have faced considerable barriers in programming. After all, end-user programmers are different from professional programmers in

at least two aspects: software engineering training and motivation (Ko, et al., 2011), both of which seem to contribute to end-user programmers' susceptibility to programming obstacles. Whereas professional programmers are expected to have formal education in computer science and/or software engineering, many end-user programmers lack such training. Signs of lack of formal training manifest in the code end users create. For example, in a study examining Yahoo!Pipes' code repository, 80% of the programs created by end-user programmers had deficiencies such as redundant modules and hard-coded values (instead of using variables) making the programs error-prone and difficult to maintain (Stolee & Elbaum, 2011).

The other difference is motivation. End-user programmers' motivation to code originates from their need to accomplish their jobs or hobbies as shown by studies such as (Dorn & Guzdial, 2010). Thus, to end-user programmers, programming is a means to some end rather than the end itself. For example, a high school physics teacher may program a Flash animation to help her students gain intuitions into the two-body problem. Her animation is merely a vehicle to convey scientific knowledge. In contrast, a professional programmer is keenly interested in the code he/she is working with because code is the important deliverable of their job. Perhaps due to end-user programmers' unique motivation, they are rarely interested in investing in up-front learning of programming knowledge which does not seem to have immediate benefits to their work at hand even if it could benefit them further down the road, a phenomenon termed by Carroll and Rosson as the "paradox of the active user" (Carroll & Rosson, 1987).

1.2 This Thesis

Inspired by theories from design, creativity, and problem solving, this thesis sets out to re-imagine how to go about helping end-user programmers overcome barriers using a theory-based approach. Because we believe that program creation by end users is essentially an act of design, ideation, and problem-solving, we drew from theories from design, creativity, and problem solving.

In design research, Schön's *reflection-in-action* (Schön, 1983) is an important theory that describes how designers approach design problems (Bayazit, 2004). On a high level, the reflection-in-action theory can be used to describe the overall process end-user programmers go through in order to create a program. This model describes design as an iterative process with three phases: *framing* (to understand and define) the problem, *acting* (to transform the

current situation to a better one or to learn more about the situation), and *reflecting* (to assess consequences and implications). The process is an iterative "conversation" between the designer and the design problem (Schön, 1983), with moves from framing to acting, to reflecting, and sometimes back to major reframing. This model can highlight barriers end users encounter in light of each phase of the model. For example, a barrier in framing the programming problem can lead to no framing output which may in turn result in unguided actions in the subsequent acting phase.

From the creativity literature, the most influential theory is arguably Guilford's theory centered on *ideation*, generation of ideas (Guilford, 1968). The ideation concept can help understand the barriers on a finer granularity. Evolved out of this work are three concepts that point to creativity: ideational fluency, flexibility, and elaboration. *Ideational fluency* refers to the rate of generating ideas related to the creative output. The creativity literature has long argued for quantity of ideas as an indicator for creativity, i.e., the more ideas a person produces, the higher chances of him/her arriving at creative ideas (Boden, 1994; Guilford, 1968). Empirical evidence supports the construct validity of using ideational output as a measure for quality of responses (Milgram, 1990). *Flexibility* is defined as generating different types of ideas (Runco, 2007; Guilford, 1968). Flexibility can be recognized when an individual moves from one ideational category to another (Runco, 2007). *Elaboration* is described as the ability to develop a basic idea or concept into a detailed version of it (Guilford, 1968). Instead of categorizing barriers according to the phase of design they occur in, the ideation concept can help us identify idea barriers which can arise during any of the design phases. Idea barriers may be used to recognize issues such as frugality of ideas a user is able to form, inflexibility in trying different ideas, and under-elaboration, which can give rise to the abandonment of potentially good ideas. These barriers may help us recognize barriers on a high level, i.e., barriers in the phase of framing, acting, and reflecting. For example, a dearth of ideas may point to the inability to form a new frame.

Finally, from the problem-solving literature, Simon believes that two types of skills are necessary for solving problems in a specific domain: *domain-specific knowledge* and general *problem-solving strategies*: "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 & Broder, 1950). The problem-solving literature completes our approach by two means. First, it can *explain* why the above barriers exist through the lack of two sets of knowledge: domain-specific knowledge, i.e., programming domain knowledge, and problem-solving strategies. Second, it can *advise* the design of solutions to help users overcome barriers, for example, by supporting both domain-specific knowledge and strategies.

Together, the design, creativity, and problem-solving theories provided a unified perspective for understanding end-user programmers' barriers and informing the design of programming environment support aimed at assisting end-user programmers in their effort to overcome these barriers.

We will explain each of these theories in detail in Section 2.2.

1.3 Thesis Statement

The thesis statement for this work is:

> *It is possible to gain from theories of design, creativity, and problem solving new insights into the barriers end-user programmers face. Further, it is possible to use these new insights to create environment features to help end-user programmers to help themselves to overcome these barriers.*

1.4 Methodology

To investigate the thesis statement, we incorporated theories into the interaction design methodology (Rogers, Sharp, & Preece, 2011) that starts with formative studies, followed by iterative design, and then a summative study. Our methodology is depicted in Figure 1.1.

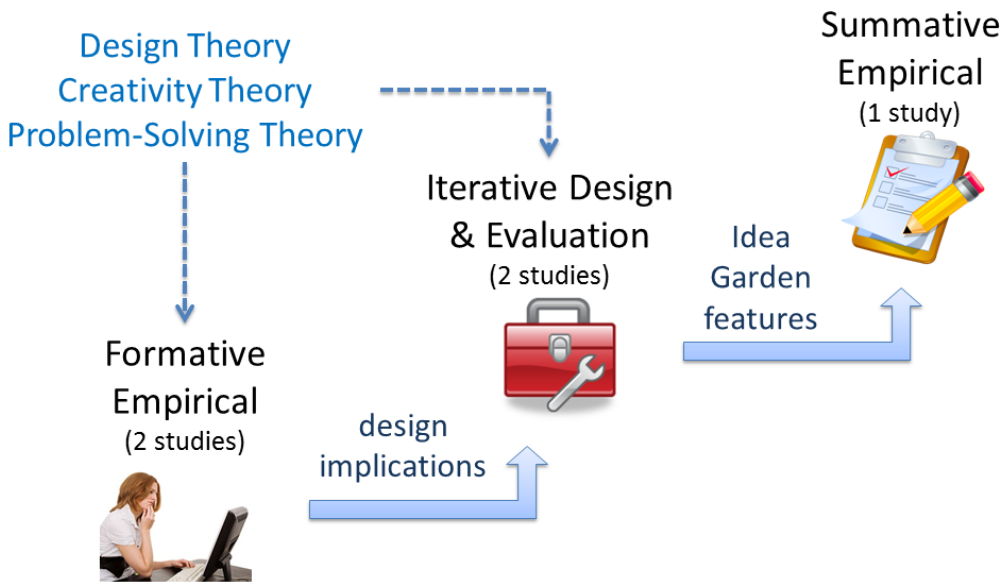


Figure 1.1 Methodology undertaken by this thesis. We started with formative empirical studies, did iterative design and evaluation, and finally a summative empirical study. Both the formative studies and the iterative design were informed by theories from design, creativity, and problem solving.

We began with two formative empirical studies. In the first formative study, we investigated how end-user programmers approach a programming task in Microsoft Popfly. We used the design and creativity theories mentioned above as a lens to interpret end-user programmers' programming behaviors and to understand the obstacles they faced. Based on the findings, we derived insights into how to design supports for end-user programming environments to help users overcome the obstacles.

In the second formative study, we conducted a similar study in IBM CoScripter and performed an analysis of the combined data from the new study and the first study using problem-solving theories. The goal was to generalize the results from the first study and to form solutions toward *how* to help users overcome barriers they encounter. This study was informed by the problem-solving theories.

The outcome of the two formative studies was a new approach which we term *Idea Gardening*. Idea Gardening is rooted in theories and literature from design, creativity, and problem solving, and is intended for supporting problem solving in end-user programming environments.

Taking the Idea Gardening approach, we iteratively designed and evaluated Idea Garden features in a programming environment aimed at nudging users into overcoming their programming barriers *by themselves*. The features were equipped with both problem-solving strategy knowledge and programming knowledge. In this iterative design process, we carried out two qualitative empirical studies: one aimed at whether the Idea Garden can help end-user programmers overcome barriers, and another aimed at whether end-user programmers were able to learn the relevant problem-solving strategies and programming knowledge.

Finally, we conducted a summative quantitative evaluation focusing on whether end-user programmers would be able to apply what they had learned from the Idea Garden in one programming task to a different programming task. The goal was to evaluate if the knowledge would stay with the end-user programmers to help them in their future tasks.

1.5 Contributions

The main contribution of this thesis is a new approach called *Idea Gardening* which is a way of designing problem-solving support for end-user programmers that is rooted in theories from design, creativity, and problem solving. This approach aims to provide guidance to designers of end-user programming environments who wish to create problem-solving support for their own environments' users. This support takes the form of an extension to an end-user programming environment. An Idea Garden extension consists of features aimed at helping end-user programmers to overcome hurdles they encounter in programming by helping them generate their *own* ideas. The Idea Garden is unique in the following ways:

- The Idea Garden *never tries to solve users' problems for them*. Instead, it provides users with just enough hints to help them make progress. For example, instead of fully functioning code, the Idea Garden provides partially functioning examples that require users' own problem solving to be incorporated into their programs.
- To enable users to help themselves, the Idea Garden features leverages both *problem-solving strategies* and *programming knowledge*, the two skill sets necessary for effective problem solving in the domain of programming, in nudging the users.
- The Idea Garden targets users who are interested in *doing* rather than *learning*. However, through a careful integration of instructional materials into users' own tasks,

the Idea Garden aims at *promoting* the *learning* of both problem-solving strategies and programming knowledge in users whose primary goal is not learning.

In addition to the main contribution which is the Idea Gardening approach itself, the work leading up to the Idea Gardening approach resulted in the following contributions:

- *Two formative empirical studies* investigating barriers end-user programmers encounter in two end-user mashup environments, namely Microsoft Popfly (Wikipedia, Microsoft Popfly, 2012) and IBM CoScripter (Lin, Wong, Nichols, Cypher, & Lau, 2009).
- *Barriers* and *insights* from the formative empirical studies from the perspective of the reflection-in-action theory from design, the ideation concept from creativity, and the two skill sets of problem-solving strategies and domain-specific knowledge from problem-solving literature.
- The *methodology* of leveraging design, creativity, and problem-solving theories and literature to understand end-user programmers' programming behaviors to gain insights into how to better support end users' programming efforts.
- *A proof-of-concept Idea Garden prototype* for the CoScripter environment.
- Empirical evidence through *two qualitative empirical studies* and a quantitative *summative study* demonstrating both the *feasibility* of the Idea Garden approach as well as its *capability* of conveying strategies and programming knowledge to end-user programmers.
- *Quantitative evidence* that knowledge users gained by interacting with the Idea Garden features during one programming task where learning was not the primary goal was *sustained*: it helped them in a subsequent programming task where the Idea Garden was no longer available.
- *Recommendations* for how to create an Idea Garden for existing end-user programming environments.

Chapter 2. Related Work and Theoretic Background

In this chapter, we describe empirical work aimed at understanding barriers end-user programmers encounter. We also explain the theories underlying our work. We defer coverage of approaches aiming at similar goals as the Idea Garden until Chapter 5 for better context.

2.1 Related Work (To Understand Barriers)

In order to design programming languages and environments that suit the needs and characteristics of end-user programmers, researchers have studied barriers end users face using various methods. One way to classify these approaches might be consider how much theoretic basis was present in these methods. We will discuss two broad categories of work related to this thesis: the grounded and the theory-informed.

Our work falls in the theory-informed category in that we aim to apply the design theory of reflection-in-action, the creativity theory, and problem-solving theories in understanding obstacles end-user programmers have and in informing features aimed at helping end-user programmers to overcome these obstacles. Below, we compare and contrast our work with existing literature.

2.1.1 Grounded Approaches

Some researchers have taken a "grounded" approach to study barriers end users encounter. Grounded theory is a research methodology that focuses on deriving theories directly from qualitative data (as opposed to developing hypotheses and then gathering and examining data for evidence that supports or refutes the hypotheses). The grounded theory approach emphasizes a set of steps for developing theories reliably through rigorous coding (the process of assigning properties to data) (Strauss & Corbin, 1998). While none of the work in end-user programming we mention below seems to have strictly followed the grounded theory methodology, they all borrowed methods from the grounded theory, e.g., by developing categories of barriers based on users' behavioral data. We will refer to these approaches as "grounded approaches". Examples of grounded approaches include (Pane, Ratanamahatana, & Myers, 2001), (Rode, Rosson, & Pérez-Quiñones, 2002), (Ko, Myers, & Aung, 2004), (Kissinger, et al., 2006), (Subrahmaniyan, Burnett, & Bogart, 2008), (Subrahmaniyan, et al.,

2008), (Kelleher, 2009), (Zang & Rosson, 2009), (Gross & Kelleher, 2009), (Dorn & Guzdial, 2010), and (Chambers & Scaffidi, 2010).

The most prominent difference between this body of work and our work is that our work is informed by others' theories whereas this body of work builds from the ground up. For example, Ko et al. studied barriers encountered by learners of Visual Basic in the context of a user interface design course (Ko, Myers, & Aung, 2004). The study used the self-reported descriptions of barriers provided by students to reveal six learning barriers faced by the students: design barriers ("I don't know what I want the computer to do"), selection barriers ("I think I know what I want the computer to do, but I don't know what to use"), use barriers ("think I know what to use, but I don't know how to use it"), coordination barriers ("I think I know what things to use, but I don't know how to make them work together"), and understanding barriers ("I thought I knew how to use this, but it didn't do what I expected") (Ko, Myers, & Aung, 2004). Our work is partially inspired by Ko et al.'s work but also brings in theories from design, creativity, and problem solving to help understand the barriers end-user programmers encounter.

Kelleher studied "pre-learners" of programming, middle school girls. She compared barriers these girls faced when programming in two versions of the Alice programming environment, Generic Alice and Storytelling Alice (Kelleher, 2009). From analyzing girls' behaviors, she found that while girls who used Generic Alice were affected by two kinds of motivational barriers: lack of interest in programming and lack of a sense of control. In contrast, girls who used the Storytelling Alice were not affected by these barriers. Instead, they experienced barriers similar to those faced by learners from Ko et al.'s study in (Ko, Myers, & Aung, 2004), such as barriers in determining what was possible with the system and barriers in finding programming tools to realize their goals (similar to Ko et al.'s selection barriers) suggesting that they had moved beyond the phase of needing motivation (in the context of the study) to become truly engaged with programming. Our work does not focus on motivational barriers. Rather, we focus on barriers that end-user programmers encounter after they have decided to do some programming.

Dorn and Guzdial studied web designers as informal learners of programming to characterize their programming knowledge and their learning strategies (Dorn & Guzdial, 2010). Through an interview with card sorting activities, they found that while web designers were familiar

with common concepts in programming such as input/output and variables, they were less unfamiliar with advanced concepts such as infinite loops, exception handling, and functional decomposition. The study suggests that what might be preventing the web designers from fully understanding these concepts might be the difficulties in learning these concepts (some advanced concepts are hard to grasp) and the lack of necessity in learning (the most useful concepts are learned first). Their work primarily focused on barriers in learning programming-related concepts. Our work is different in that we study design-, creativity- and problem-solving-related barriers. Although our work does touch upon barriers in learning a few particular programming concepts, they are not the main focus; we study them only in the context of how to problem-solve about programs.

Rode et al. also studied end users as web developers (Rode, Rosson, & Pérez-Quiñones, 2002). Instead of concentrating on learning to code in an existing programming environment, the study focused on end users' mental models of how web sites work. The problems identified included lack of knowledge of control flow, inability to describe how functionalities were implemented behind the scenes (e.g., how a web site responds to a search query), and using technical terms without being specific and precise. The study also found end users thinking in terms of sets rather than in terms of iteration (e.g. show all instead of show each) which agreed with the finding from Pane et al.'s study on children (Pane, Ratanamahatana, & Myers, 2001). Whereas their work focused on end-user programmers' understanding and mental models of technical aspects of web development, our work aims to understand end-user programmers' ways of approaching programming tasks through the lens of design, creativity, and problem-solving theories.

Wong and Hong studied end users as creators of a type of emerging web application called mashups (Wong & Hong, 2007). Mashups combine data from multiple web sources to form a unified view (Zang & Rosson, 2008). Wong and Hong found that in creating mashups, end-user programmers often had a hard time deciding what operations to choose (similar to Ko et al.'s selection barriers), especially when creating a new dataflow. In addition, end users were ineffective at forming ideas to test their theories about why their mashups went wrong (similar to Ko et al.'s understanding barriers). Last but not least, some users encountered insurmountable barriers in trying to understand the concept of dataflow and were unable to

complete the task due to this barrier. The focus of the work was to identify and remove usability barriers in a mashup environment. Its findings are reminiscent of Ko et al.'s findings.

Zang and Rosson also studied barriers in mashup creation (Zang & Rosson, 2009). In a lab study of a dozen end user programming mashups in Yahoo!Pipes, Zang and Rosson identified users' difficulties in understanding structured data (e.g., in XML form) which often led to incorrect guesses. Resonating Ko et al.'s finding with Visual Basic learners, Zang and Rosson also found the problem of failure to predict the function of unfamiliar modules, a selection barrier. For example, participant failed to recognize "Loop" as an appropriate module for their mashup. Many used "Filter" or "Union" instead. This result highlights a problem with the language's "role expressiveness", the ability of a notation to express what it is meant to do (somewhat similar to the idea of affordance) (Green & Petre, 1996). This work used the Cognitive Dimensions as heuristics for analyzing the Yahoo!Pipes language's usability and studied barriers from the perspective of the usability of the language. Our work focuses on end-user programmers' design and problem-solving processes rather than the language alone.

The above work focused on barriers end-user programmers experience in program *creation*, Subrahmaniyan et al.'s study focused on program *debugging*, in particular, spreadsheet debugging (Subrahmaniyan, et al., 2008). This work also took a grounded approach and discovered different debugging strategies used by male and female end-user programmers. Their approach relied on a content analysis of participants' self-reported descriptions of their strategies, analysis of electronic logs of users' interactions with the spreadsheet environment, and a machine learning algorithm that analyzed the electronic logs for patterns. One obstacle faced by the female end-user programmers was that the environment lacked support for one of their preferred strategies, specification checking. Both this work and ours focuses on users' problem-solving obstacles but their work focused solely on the debugging aspect of programming whereas ours focuses mostly on the creation aspect (which of course includes some debugging of emerging programs).

Chambers and Scaffidi conducted a field study that focused on barriers end users encountered in using Excel spreadsheets (Chambers & Scaffidi, 2010). Using a grounded approach, they synthesized data from a spreadsheet corpus, discussion threads of an online forum, and interviews. The main findings highlighted barriers in "problem setting" (knowing what to do but not knowing how to set up the problem.), "feature finding" (problem in finding specific

features through UI), and "config[uration]" (not knowing how to set up the Excel environment). Our work is most related to "problem setting" but also follows up on problem solving.

Gross and Kelleher studied the strategies end-user programmers adopt in searching for code to reuse and the obstacles they encountered (Gross & Kelleher, 2009). Their work studied users in a lab setting and identified several barriers in users' ways of successfully identifying functionalities to reuse. For example, users struggled to associate code with output and found it difficult to recreate the execution flow of the code. Their work focused on reuse but ours focuses on creation.

2.1.2 Theory-Informed Approaches

The opposite of grounded approaches is the theory-informed approach which aims to understand data using applicable existing theories or hypotheses derived from the theories. Examples of this approach include (Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006), (Wilson, et al., 2003), (Subrahmaniyan, Burnett, & Bogart, 2008), (Seals, Rosson, Carroll, Lewis, & Colson, 2002), and (Hundhausen, Farley, & Brown, 2009).

Our work is primarily inspired by works by those from the theory-informed category. For example, Beckwith et al. used theories identified in psychology, marketing, and education to discover and understand gender-related barriers in spreadsheet debugging (Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006). One theory from psychology was the self-efficacy theory (Bandura, 1977) that predicts that people with low self-efficacy theory would be less inclined to adopt a new approach when facing a barrier. When pattern-matching this theory to data, Beckwith et al. found that females' actions were consistent with what the theory predicted: the females with low self-efficacy were less likely than those with high self-efficacy to approach new debugging features, an example of a "new approach", in the environment (Beckwith, et al., 2006). However, the theory did not hold true for the males. Males, regardless of their self-efficacy, were equally likely to approach features unfamiliar to them. As such, the application of gender-related theories, e.g., the self-efficacy theory, helped researchers to discover and understand gender-related barriers in end-user programming. Inspired by Beckwith et al.'s approach, we take a similar theory-informed approach, but instead of focusing on gender-related barriers in debugging, we focus on barriers in program creation where we expect

theories from design, creativity, and problem solving to be useful in understanding users' behaviors.

Also in the spreadsheet debugging domain is the work from Wilson et al. that applied curiosity theory (Lowenstein, 1994) to inform the design of features to entice end-user programmers to assertions (Wilson, et al., 2003). Specifically, using the concept of an "information gap", a surprise that violates a user's expectation which could lead to curiosity and a search for explanations, Wilson et al. design an assertion feature that intentionally generated values that fell outside of users' expectations, e.g., -1 for work hours instead of from a range between 0 to 8 hours. This approach was successful in encouraging users' engagement with assertions and had positive impacts on users' debugging. Our work is not based on curiosity theory.

Subrahmaniyan et al. adopted a theory-based approach that incorporated the Attention Investment Model (Blackwell, 2002), the Technology Acceptance Model (Davis, 1989), and Minimalist Learning Theory (Carroll, 1998) to understand obstacles an end-user programmer encountered during a free trial period of a spreadsheet visualization tool (Subrahmaniyan, Burnett, & Bogart, 2008). For example, Attention Investment Model predicts a user's likelihood of pursuing an action based on perceived costs, risks, and benefits associated with the action. According to the model, perceived benefit was a positive motivator of the participant's decision, which applied to the participant's situation: the only barriers she persisted long enough to overcome were the ones she perceived to be of direct benefit to her task. Our work draws from the Attention Investment Model and Minimalist Learning Theory, but we use them to guide our design of our features.

Seals et al. conducted an empirical evaluation of a programming tutorial for a cross-generation collaboration in designing and building a simulated community topic using a visual language called Stagecast (Seals, Rosson, Carroll, Lewis, & Colson, 2002). The tutorials were designed based on the principles of Minimalist Learning Theory and the same theory was applied in understanding participants' reactions to the tutorial. For example, one principle of the Minimalist Learning Theory is to support recovery from errors which was accomplished by the tips in the tutorials the participants followed. Our work is different in the theories we chose to adopt, i.e., creativity, design, and problem-solving theories. In addition, we focus on individuals' problem solving rather than that of pairs.

Hundhausen et al. conducted an empirical evaluation to investigate whether starting programming novices in a language with visualizations and direction manipulation features would lead to positive transfer-of-learning to common textual languages (Hundhausen, Farley, & Brown, 2009). Their work identified positive transfer in novice programmers' learning as evidenced by the finding that those who had access to the visualization and direction manipulation features and then used the text-only language outperformed those who had no access throughout. Their work utilized dual coding theory (Paivio, 1983) to explain why the visualizations helped. The theory posits that (1) images and words are encoded differently in the brain, (2) connections can form between imagery and verbal encodings of a concept, and (3) it is easier for a person to remember a concept that is dually coded (i.e. having both imagery and verbal encodings) and has connections between both kinds of encodings. The visualization features facilitated the imagery encoding and the formation of the connections between the imagery and verbal encodings. Our work does not seek to understand the effect of visualizations on transfer-of-learning in programming. Instead, we focus on using creativity, design, and problem-solving theories to understand end-user programmers' barriers.

2.1.3 Summary

As mentioned in the beginning of Section 2.1, the grounded and theory-informed classification is just one way to categorize existing work on end users' programming barriers. Table 2.1 below brings in several additional categories to help classify the literature from different perspectives. The table explicitly positions the work in this thesis in relation to this literature.

Table 2.1 Related Work

Grounded vs. Theory-Informed

Grounded
(Pane, Ratanamahatana, & Myers, 2001), (Rode, Rosson, & Pérez-Quiñones, 2002), (Ko, Myers, & Aung, 2004), (Kissinger, et al., 2006), (Subrahmaniyan, Burnett, & Bogart, 2008), (Subrahmaniyan, et al., 2008), (Kelleher, 2009), (Zang & Rosson, 2009), (Gross & Kelleher, 2009), (Dorn & Guzdial, 2010), (Chambers & Scaffidi, 2010)
Theory-informed
(Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006), (Wilson, et al., 2003), (Beckwith, Innman, Rector, & Burnett, 2007), (Subrahmaniyan, Burnett, & Bogart, 2008), (Seals, Rosson, Carroll, Lewis, & Colson, 2002), (Hundhausen, Farley, & Brown, 2009), and **THIS THESIS**

Programming Tasks

Creation
(Pane, Ratanamahatana, & Myers, 2001), (Rode, Rosson, & Pérez-Quiñones, 2002), (Ko, Myers, & Aung, 2004), (Kelleher, 2009), (Zang & Rosson, 2009), (Dorn & Guzdial, 2010), (Chambers & Scaffidi, 2010), (Seals, Rosson, Carroll, Lewis, & Colson, 2002), (Hundhausen, Farley, & Brown, 2009), **THIS THESIS**
Debugging
(Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006), (Beckwith, Innman, Rector, & Burnett, 2007), (Kissinger, et al., 2006), (Subrahmaniyan, Burnett, & Bogart, 2008), (Subrahmaniyan, et al., 2008), **THIS THESIS**
Reuse
(Gross & Kelleher, 2009)

Barrier Types

Motivation
(Kelleher, 2009)
Learning
(Ko, Myers, & Aung, 2004), (Subrahmaniyan, Burnett, & Bogart, 2008), (Kelleher, 2009), (Dorn & Guzdial, 2010), (Hundhausen, Farley, & Brown, 2009), **THIS THESIS**
Programming concepts
(Dorn & Guzdial, 2010)**,** (Pane, Ratanamahatana, & Myers, 2001)**,** (Rode, Rosson, & Pérez-Quiñones, 2002)**,** (Wong & Hong, 2007)**, THIS THESIS**
Problem-solving/Strategies
(Beckwith, et al., 2006), (Kissinger, et al., 2006), (Subrahmaniyan, et al., 2008), (Gross & Kelleher, 2009), **THIS THESIS**
Mental models
(Rode, Rosson, & Pérez-Quiñones, 2002), (Zang & Rosson, 2009), (Kulesza, Stumpf, Burnett, & Kwan, 2012)
Environment features
(Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006), (Kissinger, et al., 2006), (Beckwith, Innman, Rector, & Burnett, 2007), (Zang & Rosson, 2009), (Chambers & Scaffidi, 2010), (Subrahmaniyan, Burnett, & Bogart, 2008)
Collaboration
(Seals, Rosson, Carroll, Lewis, & Colson, 2002)

Application Settings

Animation/Simulation
(Pane, Ratanamahatana, & Myers, 2001), (Kelleher, 2009), (Gross & Kelleher, 2009), (Seals, Rosson, Carroll, Lewis, & Colson, 2002)
Mashups
(Wong & Hong, 2007), (Zang & Rosson, 2009), **THIS THESIS**
Spreadsheet
(Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006), (Kissinger, et al., 2006), (Beckwith, Innman, Rector, & Burnett, 2007), (Subrahmaniyan, Burnett, & Bogart, 2008), (Subrahmaniyan, et al., 2008), (Chambers & Scaffidi, 2010), (Subrahmaniyan, Burnett, & Bogart, 2008)

Web
(Rode, Rosson, & Pérez-Quiñones, 2002), (Dorn & Guzdial, 2010)
General purpose (e.g., Visual Basic)
(Ko, Myers, & Aung, 2004), (Hundhausen, Farley, & Brown, 2009)

2.2 Theoretic Background

In this section, we explain theories and literature underpinning this thesis.

2.2.1 Design Theories and Reflection-in-Action

As mentioned in 0, Schön's reflection-in-action is an important design theory that describes designers' ways of approaching design problems. It includes three phases: framing, acting, and reflecting (Schön, 1983).

2.2.1.1 Why is the Reflection-in-Action Model Applicable to End-User Programming?

In design research, the reflection-in-action theory has been applied in numerous empirical studies aiming to understand designers' design activities (Cross, 2006). Results from these studies show that the reflection-in-action theory can help highlight the difficulties designers face. For example, a study of team design activity by student industrial designers showed that some designer teams encountered difficulties in problem framing (Valkenburg & Dorst, 1998). The unsuccessful design team had trouble coming up with new frames when their designs failed. They relied on one single frame for a design task compared with five frames used by the successful team for the same task.

We believe that the reflection-in-action theory is applicable to understanding end-user programmers' behaviors for at least two reasons. The first reason is that we believe programming by end-user programmers essentially amounts to a broad notion of *design*. In software engineering, researchers and professionals alike have argued for a broad notion of design. As early as 1992, Reeves argued that programming was a design activity (Reeves, 1992). He believed that *everything* from coding, testing, debugging to what professionals typically considered software design was all part of design. For example, he regarded testing and debugging as part of design because he regarded them as the equivalent of design validation and refinement of other engineering disciplines. As a result, he argued for coding early on as it would expose problematic areas of a design that were neglected at the start.

Krutchen holds a similar view that software design is a wider concept than what many software engineers think (Krutchen, 2005). With a view similar to Reeves, he considers a wide spectrum of software development activities ranging from requirement elicitation and capture, programming, and testing to be part of design. He demonstrated his point by casting software design into the Function-Behavior-Structure (FBS) model (Gero & Kannengiesser, 2004), a model for explaining the nature of design and for analyzing its process as design occurs in many engineering disciplines. In particular, he mapped the processes and products of the Rational Unified Process (Krutchen, 2005), a well-established software engineering process model, to the design processes and artifacts in the FBS model.

We hold a similar view to the views of Reeves and Krutchen. We believe that what is seen as *just coding* from the view of traditional software engineering in fact is peppered at the micro level with design decisions and, as such, much of it can be viewed as *designing*. We believe that this is true not only of professional programmers but also of end-user programmers. For example, some end-user programmers might not decompose a problem top-down, instead taking an opportunistic approach. Instead of *planning* a solution in advance of implementing it, end users may start coding early on, grabbing opportunities whenever they arise: essentially doing coding integrated with bottom-up design. Studies such as those presented in (Hartmann, Doorley, & Klemmer, 2008) and (Brandt, Guo, Lewenstein, & Klemmer, 2008) have discovered a similar phenomenon called "opportunistic programming" where programmers chose to avoid upfront design and delved straight into prototyping. This lack of advance design planning is reminiscent of "debugging into existence" (Rode & Rosson, 2003), i.e., eschewing analysis in advance and incrementally developing a small part of the system then iteratively using the debugger to make design decisions while refining and correcting problems.

A second reason why we believe that the reflection-in-action theory is applicable to end-user programmers is that we believe that when end-user programmers are coding, they are trying to solve ill-defined problems, the same type of problems that designers are faced with. Reflection-in-action is suited for analyzing ill-defined problems such as design problems and social problems (Rittel & Webber, 1984). It is important to note that whether a problem is considered to be ill-defined or not depends, to a large extent, on the problem-solver's knowledge, experience, and/or point of view of the problem (as can be inferred from (Hayes,

1978)). For example, an experienced programmer may see the problem of creating a sorting algorithm as well-defined whereas a novice may view the same problem as ill-defined due to a lack of experience and strategies for approaching the problem.

Similar to novice programmers, we believe that end users are often faced with ill-defined design problems in programming. In fact, empirical studies of end-user programmers have reported design-like barriers in end users. Ko et al.'s work on learning barriers of Visual Basic learners reported "design barriers" where the learners could not specify what they would like the program to do (Ko, Myers, & Aung, 2004). These barriers are inherent in programming problems; they are unrelated to how solutions to these problems might be represented, e.g., with words or diagrams. Examples of design barriers included sorting and concurrency. Reminiscent of the "design barriers", Chambers and Scaffidi identified a "Problem setup" barrier that end users of Microsoft Excel faced (Chambers & Scaffidi, 2010). Chambers and Scaffidi described the barrier as "Often the user will know what they want to do [e.g., what the end result of the program should be], but wants to know how to set up the problem."

For these two reasons, the reflection-in-action theory seems potentially very suitable. We believe that using the reflection-in-action theory to understand end-user programmers' programming activities and the barriers they are facing may lead to insights into how to help them overcome those barriers.

2.2.1.2 Design as Reflective Practice and Schön's Reflection-in-Action Model

The reflection-in-action theory was born out of Schön's seminal work on reflective practice (Schön, 1983). According to Schön's perspective, design is a process where the designer learns and makes sense of the design situation through experience (Cross, 2007). In this process, the designer forms a solution to the problem based on her definition of the problem. Upon seeing how the situation responds to the solution, she may rethink her definition of the problem which will in turn affect how she formulates subsequent solutions. Therefore, the problem definition and the solution *co-evolve* (Dorst & Cross, 2001). Likewise, Bazjanac views design as a learning process in which the solution evolves through many cycles of problem formulation, solution evaluation, and documentation (Bazjanac, 1974). Numerous empirical studies aimed at understanding how designers work have taken the approach of design as a reflective practice (Cross, 2006).

The reflection-in-action theory adds details to the notion of design as a reflective practice. It is a descriptive theory that illustrates expert practitioners' way of approaching ill-defined problems such as design problems (Schön, 1983). It describes the process in which the practitioner thinks and acts in action in order to arrive at a viable solution for a problem. This theory is based on the fact that the practitioner thinks about what she does while she is doing it.

According to this theory, the process of solving an ill-defined problem is broken into three phases. Framing involves understanding and defining the problem. This is also known as "setting the problem". Acting aims to transform the current situation to a better one, or to learn more about the situation. This involves experimentation that produces a solution or new understanding of the situation that can inform further moves. Reflecting looks back on actions to assess their consequences and implications. Surprise at the consequences of actions may make the practitioner surface her understanding of the situation and to develop a new framing of the situation (reframing). It may also lead her to act some more. Therefore, the process of reflection-in-action is iterative (Schön, 1983), with moves from framing to acting to reflecting, and sometimes back to major reframing as moves within existing frames may lead to new understanding of the problem calling for reframing. Figure 2.1 shows the relationship between the different steps.

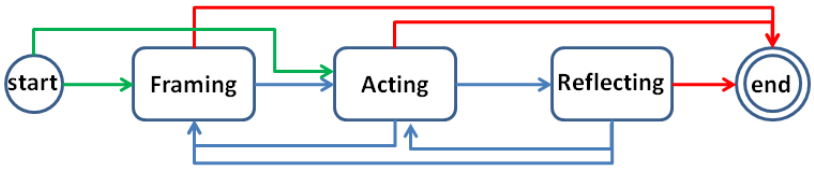


Figure 2.1 The reflection-in-action theory.

As described above, the practitioner may traverse the three steps in a sequential fashion, starting from framing (Start→Framing), then from framing to acting (Framing→Acting), and finally from Acting to Reflecting (Acting→Reflecting).

While Schön's original account on the theory does not explicitly mention starting from acting (Start→Acting), we believe it is possible. When Schön first brought up this theory, his goal was to describe how expert practitioners solved problems. One distinction between experts

and novices is their ability to frame problems. Experts are better able to frame problems because of their past experience, a "repertoire" that they may use towards understanding the problem at hand and acting on it. When solving a problem, they impose a frame on the situation by seeing it as something already present in their repertoire and act accordingly (Start→Framing). However, it is possible for novices who lack such a repertoire to begin their reflection-in-action by going straight to acting without having properly framed the problem (Start→Acting). For example, novice programmers have been found to design their programs bottom-up without having an overall grasp of the problem (Jeffries, Turner, Polson, & Atwood, 1981) (Rist, 1991).

An alternative interpretation of the shortcut from Start to Acting may have been suggested by Schön himself. He suggests that frames may be implicit in the practitioner's actions. She may not surface it to her conscious as she does things. In this case, skipping the framing step does not necessarily indicate the absence of a frame.

Acting may lead to more framing (Acting→ Framing). This happens when the practitioner feels a need to frame the problem rather than moving on to reflect on what she has done.

Reflecting may lead to more acting (Reflecting→Acting). For example, when reflecting confirmed an action, the practitioner may continue pursuing the same line of actions.

Reflecting may also lead to framing (Reflecting→Framing). For instance, when an action is disconfirmed, the practitioner may question about the theories or assumptions underlying her actions, which leads to more framing.

Finally, the practitioner may choose to stop working on the problem in any of the three steps of Framing, Acting, or Reflecting (Framing→Stop, Acting→Stop, Reflecting→Stop).

In essence, reflection-in-action allows the practitioner to reshape what she is working on while she is working on it.

2.2.2 Creativity Theories and Ideation

Guilford advocated the role of ideation in creative problem solving (Guilford, 1968). As we mentioned in Section 1.2, he proposed three central concepts in creativity that focused on ideation: fluency, flexibility, and elaboration.

2.2.2.1 Why Are the Ideation Concepts Applicable to End-User Programming?

We believe that concepts from the creativity literature can be useful in understanding how end-user programmers engage in creative acts. When end-user programmers sit down to craft a piece of code, they engage in *personal creativity*, the creations of things new to oneself (Boden, 1994). In contrast to historical creativity which refers to acts resulting in groundbreaking ideas or inventions in a specific domain (such as the discovery of penicillin and the invention of the Internet), personal creativity concentrates in creations by a single creator (Boden, 1994). However, the road to personal creativity might be bumpy, especially for end-user programmers who lack the necessary programming skills to succeed at programming. Therefore, we believe it would be beneficial to bring in creativity theories such as those centered on Guilford' ideation concept to understand the barriers end-user programmers run into. Doing so may open up new opportunities to improve programming environments in new ways to nurture end users' personal creativity.

2.2.2.2 Creativity's Concept of Ideation: Fluency, Flexibility, and Elaboration

As we have already mentioned, Guilford's three concepts are related to ideation. Fluency suggests that in order to arrive at good solutions, a problem solver needs to produce a plethora of ideas in the first place. In agreement with Guilford's concept, Terry and Mynatt showed that one important strategy for acting upon hard problems is to produce many candidate solutions, evaluate them, and select the best ones (Terry & Mynatt, 2002). Similarly, design education places much emphasis on this approach of coming up with a lot of ideas in the face of ill-defined design problems (Do & Gross, 2007). In fact, empirical evidence supports the construct validity of using ideational output as a measure for quality of responses (Milgram, 1990).

Guilford describes the concept of flexibility as a person's ability to form different types of ideas (Runco, 2007; Guilford, 1968). According to creativity literature, flexibility can be detected when an individual's ideational output moves from one category to another (Runco, 2007). For example, a person who can name uses of a brick different from its original use (as a type of construction material), such as to use it as a door stopper, as a piece of chalk, or as a souvenir, is far more flexible, and therefore, more creative than a person who is stuck with the idea of using a brick to build a wall or a house.

Elaboration is defined by Guildford as the ability to develop a sketchy idea into one that is fully-detailed (Guilford, 1968). Elaboration allows for the addition of important details to an idea making it rich and intricate.

2.2.2.3 Encouraging Personal Creativity

From a practical stand-point, Osborn has proposed two widely recognized techniques, association and brainstorming, to help people form new ideas (Osborn, 1953). Association can be accomplished by following any of the three laws established by the ancient Greeks: contiguity, similarity, and contrast (Osborn, 1953). Contiguity is to relate one object to another object that is close to it. For example, seeing a baby's shoes would remind a person of the baby. Similarity is to relate an object to an object that is similar to it. For example, a picture of a lion reminds a person of his cat. Contrast is to think of an object that is opposite of the object at hand. For example, a mouse reminds a person of an elephant.

While the laws of association focuses on where ideas might come from, i.e., through congruity, similarity, and contrast, the brainstorming technique focuses on the process of coming up with a lot of ideas. Two principles are central to the brainstorming technique which Osborn refers to as a way for "deliberate idea finding": (1) Deferment of judgment – a person should not start evaluating ideas until he/she has developed a good list of ideas, and (2) quantity breeds quality (which agrees with Guilford's ideational fluency concept).

While Guilford and Osborn's work focused on the products of creative endeavors, i.e., ideas, other researchers have concentrated on the creative person and researched individual traits linked to creativity. Amabile found that creative individuals were often highly motivated and did not easy give up in face of a challenge (Amabile, 1996). Sawyer reported studies on creative individuals that had identified traits such as metaphorical thinking, flexible decision making, tolerance of ambiguity, willingness to take risks, and self-confidence (Sawyer, 2006). These traits overlap with, to a large extent, traits of successful designers. For example, in interview studies of top designers, Cross identified similar traits of tolerance of ambiguity and willingness to take risks (Cross, 2006).

Some researchers have suggested ways in which computers might assist people in solving problems creatively. According to Lubart, computers can be involved in creative work in four ways to foster creativity: (1) by managing creative work, (2) by facilitating communication

between people working together on creative projects, (3) by supporting creativity enhancement techniques, and (4) by working cooperatively with people to help them generate ideas (Lubart, 2005). Shneindermann proposed four traits of a creativity-conducive computer environment (Shneidermann, 2007), some of which overlap with Lubart's ideas: supporting exploratory search, enabling collaboration, history keeping (keeping track of idea exploration), allowing low threshholds, high ceilings, and wide walls (easy for novices to enter and ambitious enough for experts). Some tools have begun to take these traits. For example, Parallel Pies (Terry & Mynatt, 2004), a system for image manipulation, provides support for users to experiment with multiple manipulations (e.g., sharpening an imaging in magnitudes) simultaneously and compare their results side by side. Another example is DENIM (Newman, Lin, Hong, & Landay, 2003), a web design tool that allows the designer to express rough ideas without having to commit to detailed designs of each page. Therefore, both Parallel Pies and DEMIN facilitate exploratory search.

2.2.3 Problem-Solving Theories

Problem solving is a cognitive process aimed at transforming a given situation into a goal situation when the solution to achieve the transformation is unknown to the problem-solver (Mayer, 1990). Simon proposed that two types of knowledge, problem-solving strategies and domain knowledge, are essential to problem solving (Simon, 1980).

2.2.3.1 Why Are Problem-Solving Theories Applicable to End-User Programming?

When an end-user programmer encounters a barrier, he/she has to problem-solve in order to overcome that barrier. Thus, Simon's theory seems pertinent in two useful ways. First, the theory can help pin-point what kinds of knowledge end users are lacking when they run into barriers. For example, some barriers might result from a lack of programming domain-knowledge whereas others might be caused by a poor choice of problem-solving strategies. Second, the theory can inform what programming environments need to support in order to help user overcome their barriers, e.g., by facilitating the acquisition of necessary programming knowledge and enabling adoption of new problem-solving strategies as end-user programmers are working on their self-directly programming tasks.

2.2.3.2 Problem-Solving Theories

Simon believes that two sets of skills are essential for a problem-solver to be able to solve problems successfully in any given domain: general problem-solving strategies and domain knowledge (Simon, 1980). He compares domain 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 & Broder, 1950).

Problem-solving literature has reported general problem-solving strategies that may be applied across different problem domains, e.g., (Polya, 1973). Polya's book titled "How to Solve It?" explains an array of problem-solving strategies with a focus on their application in mathematical problem solving (Polya, 1973). The strategies, or "heuristics", in Polya's words included analogy (find a problem analogous to the problem at hand), generalization (find a problem more general than the problem at hand), specialization (find a problem more specialized than the problem at hand), decomposing and recombining (i.e., divide-and-conquer), and working backwards (start with the goal and work backwards toward the givens). Wickelgren introduced seven problem-solving techniques some of which overlapped with Polya's "heuristics", for example, working backwards (Wickelgren, 1974). Similar to Polya, Levine encouraged analogies. He also explained and encourages the use of "go to the extremes" (solving an extreme case of a problem) and "simply the problem" (solving a simplified version of a problem) (Levine, 1994) . Osborn concentrated on creative problem solving and advocated the laws of association - the analogy strategy (Osborn, 1953). Apart from strategies, domain knowledge is essential to problem solving. Simon believed that "expertness" (or expertise) is not possible without domain knowledge (Simon, 1980). For example, it is not possible to become a physicist without learning physics. Newell and Simon have used production systems, i.e., databases of condition - action pairs, to represent experts' domain knowledge in the computer (Newell & Simon, 1972). Creativity literature agrees that domain knowledge is necessary for solving problems creatively. For example, in the Creative Problem-Solving Process (Osborn, 1953) which includes the steps of fact-finding, idea-

finding, and solution-finding, "fact finding" is related to domain knowledge, e.g., knowing what construct to use to implement iteration is a fact in the domain of programming.

Mayer's model of problem-solving process can be understood from Simon's perspective. According to Mayer, problem solving is a procedure with two phases: a *representation phase* and a *solution phase* (Mayer, 1990). In the representation phase, there are two steps: problem translation (converting a problem statement into a mental representation) and problem integration (constructing a mental model of the situation presented in the problem statement). The problem translation step requires factual and linguistic knowledge whereas the problem integration step requires schematic knowledge, knowledge about problem types. In the solution phase, there are two steps as well: solution planning (devising a problem plan) and solution execution (carrying out the plan). Solution planning requires strategic knowledge. Solution execution requires procedural knowledge. We believe that strategic knowledge maps to Simon's general problem-solving skill set, and we argue that factual knowledge, schematic knowledge, and procedural knowledge are related to domain knowledge.

Chapter 3. Formative Study 1: Understanding End-User Programming from the Lens of Design and Creativity

This chapter presents a study whose goal was to use the reflection-in-action theory from design and the concept of ideation from creativity to understand how end-user programmers approach programming tasks and what barriers they encountered in the Microsoft Popfly mashup environment. Mashups are web applications that use and combine data, presentation, or functionality from two or more sources to create new services (Wikipedia, Mashup (web_application_hybrid), 2012). We chose web mashups because as more and more users are becoming increasingly reliant on the web both for life and work, end-user mashups present a unique opportunity for ordinary end users to take advantage of data and services that already exist on the Internet to transform the web to fit their own individual needs.

Therefore, this study addressed the following research questions:

> *RQ1: Can we understand end-user programmers' behaviors using the reflection-in-action design theory and the ideation concept from creativity?*
>
> *RQ2: What implications for an Idea Garden can we derive?*

3.1 Empirical Study

3.1.1 Participants

This study included four female and six male college students from a wide variety of majors (e.g., biology, nutrition science). None were computer science students or had taken computer science courses beyond the elementary level. One female and four males had past programming experience either in high school, college, or both (one male had one course; the rest had two). All participants were comfortable with web browsing.

3.1.2 Procedure

We used the think-aloud approach, conducting the study with one participant at a time. Participants first filled out a background questionnaire and worked on a hands-on tutorial (see Section 3.1.4 Tutorials) in which they were allowed to ask questions. They then completed a self-efficacy questionnaire adapted from (Compeau & Higgins, 1995) to the specific task of

end-user mashup creation. We collected self-efficacy scores because self-efficacy had previously been found to impact end users' approaches to programming tasks (Beckwith, et al., 2005) (Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006) (Grigoreanu, et al., 2008). Participants then practiced the think-aloud procedure before proceeding to the main task. When participants stopped making progress, the researcher administered an additional mini-tutorial to help them (see Section 3.1.4 Tutorials).

The data we collected included a video capture of participants' interactions with the environment (including their facial expressions), and participants' final mashups.

3.1.3 Environment

Participants used the Microsoft Popfly environment to create mashups. In Popfly, users build mashups using basic programming constructs called blocks. Each block performs a set of operations such as data retrieval and data display. Each operation takes input parameters to allow customization. Blocks are connected to form a network in which the output of a block can be used as input for adjacent blocks. Figure 3.1 shows a mashup example in which the Flickr block sends a list of images about "beaches" with their geographical coordinates to the VirtualEarth block (Figure 3.1: top and middle) to display them on a map (Figure 3.1: bottom). In Popfly, blocks are listed in different categories, which users can search. Additionally, users may share their mashups with others for reuse and modification. Shared mashups can be retrieved using a textual search.

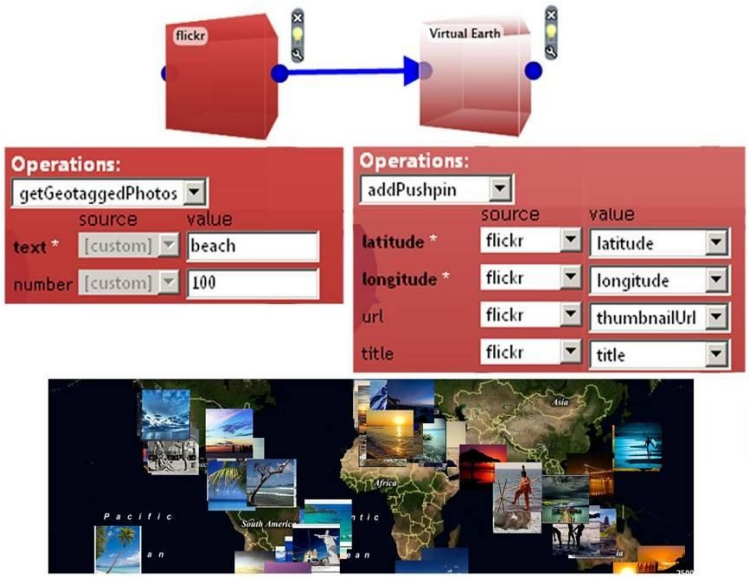


Figure 3.1 The pre-task tutorial example mashup in Popfly Mashup Creator. Top: the blocks. Middle: some of the blocks' settings. Bottom: results generated by pressing the Run button (not shown).

3.1.4 Tutorials

The pre-task 20-minute tutorial provided participants with an introduction to mashups, and included two live examples of mashups before a short hands-on session. The hands-on session familiarized users with Popfly's basic features, how to search and modify other people's mashups, and the Help feature. Figure 3.1 shows the mashup participants created during the tutorial.

During the task, participants who had not made progress for 15 consecutive minutes received an additional 5-minute tutorial to help them regain productivity. The tutorial consisted of creating two mini mashups. This tutorial was delivered if a participant demonstrated difficulty in generating new ideas to approach the task. Although the mid-task tutorial may have influenced participants' behaviors, we compare its effect to encountering a well-chosen example. The mid-task tutorial was given to half of the participants (two males and three females).

3.1.5 Task

The task involved creating a mashup about movies shown in a city. Paper, pens and sticky notes were provided. The mashup required the following pieces of information: (1) a list of local theaters, (2) movies shown at each theater along with information, e.g., running and show times, (3) a picture, and (4) a news story for each movie. Prior to the study, we refined the experimental setup and the task with pilot runs.

3.2 Analysis Method

We coded the study's transcript with two code sets (Table 3.1): *reflection-in-action* (Schön, 1983) commonly used for studying design activities (Bayazit, 2004) and *ideations* that we designed based on creativity literature (Guilford, 1968).

Table 3.1 Reflection-in-Action and Ideation Code Sets

Code	Example
Reflection-in-action (mode switches)	
Framing	*It looks like I have to have multiple VirtualEarth.*
Acting	*[Adds another VirtualEarth block]*
Reflecting	*So it gives me the theaters, and the movies themselves.*
Ideations (instances)	
Ideas for blocks	Expansion: *[Adds LocalMovies to the workspace]*
	Contraction: *[Removes block LocalMovies]*
Ideas for which blocks to connect	Expansion: *So I need to connect LocalMovies to VirtualEarth*
	Contraction: *[Removes link from LocalMovies to Flickr]*
Ideas for block dependencies	Contraction then expansion: *[Changes value for title from Local Movies' Theater Name to Local Movies' Movie Name]*
Within-block ideas	Expansion: *[Types THEATER CITY STATE in title field]*

For reflection-in-action, we used one code for each of the three phases in the reflection-in-action theory. *Framing* described events in which participants tried to understand and define the problem, either by generating a hypothesis to explore, or by gathering information to narrow down the design space. *Acting* described events in which participants started or changed their mashups. *Reflecting* described events in which participants evaluated their actions. Table 3.1 shows examples of each.

We built upon Guilford's notion of "ideational fluency" to create the ideations codes. As mentioned in Section 2.2.2, ideational fluency refers to rate of generating ideas (Guilford, 1968). To account for this notion, we coded expansions and contractions of the participant's working set of major ideas. *Expansion* describes a new idea to solve the problem, or the elaboration of an existing idea. In contrast, *contraction* is the abandonment of an existing idea. In our analysis, we only coded expansion/contraction if there was unambiguous evidence of an idea addition/deletion through their verbalization or action. As a result, this code set mainly expresses how ideation processes were reflected by actions carried out in the workspace.

For each code set, two researchers coded small portions of the transcripts independently and compared inter-coder agreement until they reached an 80% agreement covering at least 20% of the transcripts. Researchers then split up the remaining work and coded independently.

3.3 Result 1: Reflection-in-Action Cycles

To provide context for the results, we first provide the participants' success levels in achieving the given task. In particular, participants' IDs are ordered by their success levels as measured by the number of requirements they achieved during the task (listed in parentheses): F1(4), F2(2.5), F3(2), F4(2); M1(3.5), M2(3.5), M3(3), M4(3), M5(2.5), and M6(2). Half points indicate partially fulfilled requirements, e.g., not all movies showing a picture.

To get an overview of participants' behaviors, we graphed the result of the reflection-in-action code set over time. As Figure 3.2 shows, participants made extensive use of all three phases, iterating tightly through the reflection-in-action cycles. In particular, we identified three common patterns: *stair-step*, *w*, and *restart*. The stair-step pattern refers to a succession of consecutive episodes of framing, acting and reflecting. The *w* pattern refers to participants switching more than one time from framing to acting and back or from acting to reflecting and back. The other pattern, which occurred occasionally, was the restart pattern, in which reflection led to a return to the framing stage. An example of each pattern is illustrated in Figure 3.2. Notably absent was any kind of waterfall-like pattern that would have featured a fairly long period of framing alone first, then a fairly long sequence of acting without returning to framing, then a sequence of reflecting. This indicates that our participants used a highly iterative development style—not one characterized by lack of design, but rather one peppered with numerous instances of "micro design".

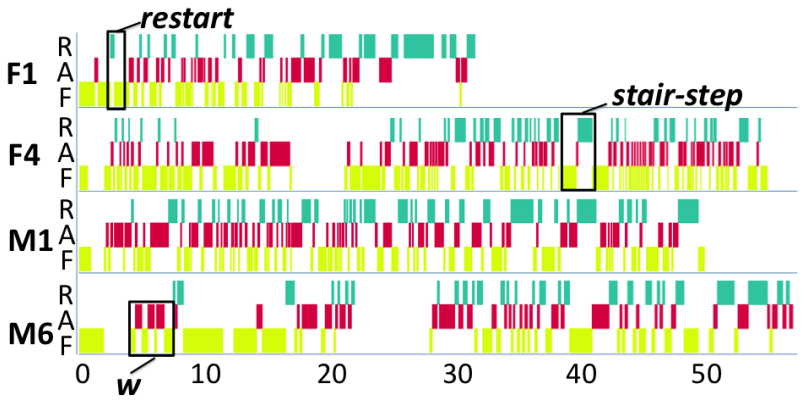


Figure 3.2 Reflection-in-action cycles with four participants.

3.4 Result 2: Framing the Problem

"Create a mashup to…"—but how? In order to begin, one needs to have a grasp of the problem. The notion of framing captures efforts to understand and define the problem (Schön, 1983). We discovered two issues participants encountered in the framing stage. First, successful participants' framing efforts often produced actionable ideas to guide their actions whereas the unsuccessful ones' often did not. Second, participants' inclination to reframe in the face of failure differed. We discuss each of these issues next.

3.4.1 Good framing output guides effective actions

When analyzing our participants' framing episodes, we noticed that the successful episodes were able to suggest actionable ideas as to how to proceed with the task, whereas the unsuccessful often failed to do so. This difference drew our attention to the importance of framing's output, i.e., ideas that guided actions. In fact, visiting the framing phase, whether the participant exited with or without output, was often critical to the success of what came next.

For instance, M1's framing usually produced output in the form of actionable ideas, and his subsequent actions made direct use of those ideas. As an example, earlier in his task, he had set up Flickr to get pictures of movies. Then he discovered a problem in minute #11—his mashup did not return pictures. He re-entered the framing phase briefly in minute #11, producing the hypothesis that the pictures' sorting criterion might be wrong. He translated this idea to action immediately (minute #12):

M1: "Theater address may not be right." [Changes the sorting criterion in Flickr] "Sort by movie names"

By contrast, F4, the least successful participant, often failed to produce actionable ideas from her framing. For example, numerous times she filtered out possible ideas before even trying to follow up on them, such as at minute #4:

F4: "Flickr. Settings." [Looks at getGeoTaggedPhotos which is the default operation and getPhotos. Leaves the default selected.] "Ok I'm doing this wrong…"

F4's low self-efficacy (3.3, vs. an average of 3.48 for females and 3.8 for all) may have hindered the production of such output, perhaps because it turned her focus toward her own capabilities and away from solving the problem itself. When her framing did not produce outputs, F4 had no inputs for the action phase. She flailed, choosing actions to try at random, often repeating ideas that she had already tried multiple times:

F4: …"Didn't work…Click to get mashing ideas" [Reads mashing suggestions:] "GeoNames, Flickr…" [Hovers over Phonebook. Picks GeoNames. Hovering, reads:] "get latitude and longitude" <which is the default>

"Oh I keep on doing that."

With the participants like F4 who suffered from unsuccessful framing, we found framing episodes with no output ideas, whose output ideas were too vague to act upon or simply repeated an idea that had failed earlier.

But why did participants leave the framing phase without output? A lack of repertoires may be the culprit. According to Schön, experts have repertoires of past approaches. They bring these repertoires to a new situation by "imposing" a previously useful frame on it, testing the fit by seeing if their actions in the new situation contradict the reused frame. Without such repertoires, end users' framing efforts are prone to result in no ideas for further actions.

Implications: We hypothesize that the lack of repertoires of programming problems and known solutions to them in end users might be causing the dearth of framing output ideas. One promising avenue to assist them in framing would be providing users with repertoires of

this kind. Examples can serve a basis for a repertoire. Examples are common in end-user programming environments, and were available in Popfly—but when examples were available in this environment, attempts to learn from examples failed (17 out of 21 among all participants). The problem was that participants were unable to find the right examples or to distill useful information from them. This suggests the need for better support for helping users find and make use of the examples they need to address the problems they are having.

Another option might be to facilitate framing through case-based reasoning (Kolodner J. , 1993). A case is different from an example in the sense that it provides a narrative that describes the process an expert adopted to design the solution to a problem (Dorn, 2011) in addition to the solution itself. Because of this rich contextual information on how an expert solution was devised, a well-designed case can help learners "interpret, reflect on, and apply experiences . . . in such a way that valuable learning takes place" ((Kolodner, Owensby, & Guzdial, 2004) p. 829 as cited in (Dorn, 2011)). Dorn argues that when a case structure is well-designed, it provides an example for learners in how to organize knowledge and experiences (Dorn, 2011), the kind of knowledge that a repertoire is composed of.

Tools to assist end-user mashup programmers to refine their understanding of the problem and possible solution ideas, e.g., through examples or cases, could help prevent end-user programmers from coming away from their framing efforts empty-handed.

3.4.2 If an idea fails, reframe and get a new one

Using the ideations codes, we noticed that some participants shared the same ideas but the degrees to which they were attached to those ideas varied greatly. Some participants refused to discard unworkable ideas, and we viewed that as inflexibility. As mentioned before, flexibility, the ability to produce a variety of ideas, is critical to creative output. One way to achieve this is through what Schön called reframing; that is to change one's definition of the problem to approach it from a different angle, which allows for the discovery of very different solution ideas.

F4 exhibited an example of inflexibility in her refusal to reframe. She had the idea in minute #9 that she needed a map when in fact using a map was not a viable solution to the task. Other than a brief detour at minute #40, she stayed with that idea throughout the session, trying to

get movie information and pictures to show up on a map. When her idea failed, instead of reframing or looking for other alternatives, F4 turned to a "get mashing ideas" tool in Popfly that lists blocks that could communicate with blocks already in the workspace. This produced actionable ideas (the suggested blocks), but these ideas came from the environment, not from her head. There was no evidence that she reflected on what had gone wrong with her previous attempts, nor attempted in reframing to rethink the problem. Instead, she simply repeated actions she had tried before, with no progress in the mashup itself or in evolving her understanding of the problem or potential solutions.

On the contrary, flexibility in reframing did not seem to be difficult for the more successful participants; they seemed to recognize the time to abandon nonproductive ideas. M1 is an interesting contrast to F4, because he started with exactly the same idea as F4, i.e., that pictures needed to be placed on a map, and then tried to use exactly the same blocks as F4, which occurred in minute #6.

But unlike F4, when he did not succeed with that idea, he abandoned it. After only two attempts to get pictures to show up on a map failed, he reentered the framing phase to look for other possible approaches, at which point he came across the Local Information tab that led to Local Movies. By minute #12, he had already taken that idea into the acting phase to pursue the Local Movies idea.

The uncertainty in the reframing stage of reflection-in-action was difficult for even the most successful participant. For example, F1 said, "I don't know what I'm doing". It especially challenges people with low self-efficacy like F4, because according to self-efficacy theory, low self-efficacy often leads to low flexibility (Bandura, 1977). Like F4, low-self-efficacy people may attribute failures to their own lack of abilities, thereby pursuing poor ideas too long.

Implications: "What-if" features might help with inflexibility and unwillingness to reframe for low self-efficacy users. For example, a tool that would allow users to make assumptions about what a block will output might enable users to explore assumptions in multiple ways. One way might be to focus on testing the assumption. A second way might be to focus on companion blocks compatible with that assumption. A third way might be to focus on competing blocks supporting the same assumption.

3.5 Result 3: Acting upon Ideas

Acting upon ideas can be regarded as "just" implementation. Even so, its interwoven relationship with design decisions sheds insights on the way ideas progressed in our participants' mashups as well as obstacles to such progression.

In transferring ideas to action, one obvious reason our participants took these actions was to follow up on ideas or hypotheses generated in the framing stage. A second reason was to produce a specific outcome as distinguished from those generated by hypotheses or a goal of exploration. A third reason was exploratory—participants acted to explore and to see what would happen in order to understand the situation better. These goals for acting are consistent with the reflection-in-action theory (Schön, 1983), but in addition we identified barriers participants faced reflected by the lack of support of acting in the environment, namely the lack of support for tinkering, elaboration, and parallel explorations of ideas.

3.5.1 Explore and tinker… not so effectively

Schön characterizes exploratory actions as "probing, playful activity by which we get a feel for things" (Schön, 1983). Research in education (Rowe, 1978) and end-user programming (Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006) has pointed to the benefits of tinkering. We took note of tinkering behaviors by looking at the output from framing and the types of ideations people had. We noticed that participants sometimes left the framing phase without concrete ideas to act on. In those occasions, participants tinkered, generating fodder for reflection, and sometimes new ideas or hypotheses that might lead to later developments. For example, M3 tinkered with the options of the Local Movies block without prior expectations as to what the changes would bring about. By tinkering he discovered the usefulness of the Local Movies block, i.e., the ability to deliver theater information, so he retained the block in his mashup and built other ideas around that.

M3: "I'm just trying to figure out how to get the program to run to show movies around CITY but I can't figure it out...

I'll just keep clicking around 'till I get it… Try a different operation to see if it works… So far I've found out the theaters within CITY…"

M2 and M4 tinkered excessively with blocks' connections, reflected in part by the large number of which blocks to connect ideas (see the dark gray shade in Figure 3.3; M2 and M4's portions are surrounded by rectangles). Although tinkering sometimes led to useful outcomes, more often it did not:

M2: [Links Local Movies to Yahoo Images and MSN News, which feed to Block Inspector. Runs. Nothing shows] "So, let's try all these in series."[Does that. Runs. Nothing shows] "Nope. So, I was on the right track before."

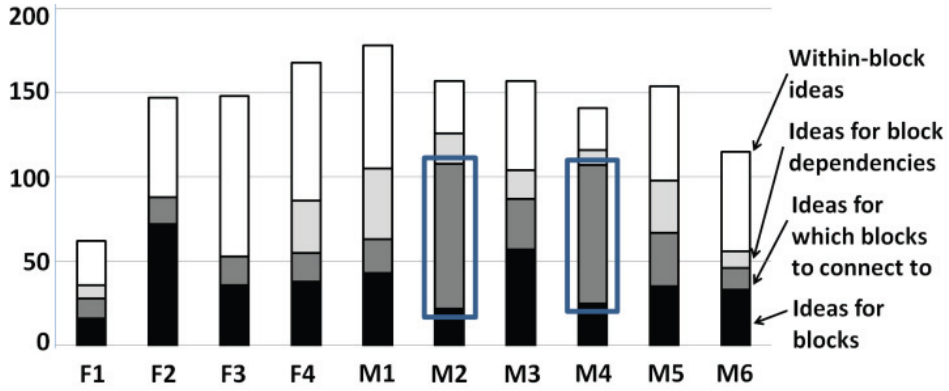


Figure 3.3 Ideation counts: number of all types of ideations.

As these examples illustrate, tinkering did not consistently occur with reflection, impeding participants' understanding of the design options available to them in the form of blocks supplied by the environment or how blocks may work together. Partially to blame is the cost of carrying action to reflection in the environment. Popfly's runtime view (Figure 3.1 bottom), the primary facilitator for reflection, is separate from the implementation interface (Figure 3.1 top and middle). This separation made it difficult for participants to cross-reference mashups with their output.

Moreover, participants were only allowed to view the runtime results for the entire mashup as opposed to those originating from tinkering with a portion of the mashup. Because of these attributes, reflection tended to slow down ability to act.

Implications: Literature in end-user programming has shown that tinkering with reflection can be helpful, but tinkering without reflection has been associated with negative outcomes

(Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006). In our environment, a barrier to carrying action to reflection was the cost of running. On the other hand, previous tinkering research has shown that the cost of running can also be too low, encouraging some end users (usually males) to tinker without bothering to reflect (Beckwith, Burnett, Grigoreanu, & Wiedenbeck, 2006). Thus, in order to encourage tinkering productively, the cost of crossing the bridge from action to reflection needs to be carefully considered, so that it is neither too high nor too low. Grigoreanu et al. has shown that it is possible to influence tinkering behaviors through feature design (Grigoreanu, et al., 2008).

3.5.2 Elaborate, but with Moderation

Elaboration is an important component of creativity. By analyzing the ideation codes, we discovered both good and poor elaboration behaviors. For example, F1 was successful at elaborating her ideas systematically. This was depicted by her organized engagement with all types of ideas. Her ideation processes often followed a pattern: picking a candidate block (ideas for blocks), examining its options (within block ideas), connecting it to other blocks (ideas for which blocks to connect), and adjusting settings of blocks to account for the inclusion of the new block (ideas for block dependencies). As a result of adequate elaboration, she was able to distinguish good ideas from poor, and act effectively toward solving the problem.

In contrast, elaboration was problematic for most participants. Two major issues were: lack of elaboration and excessive elaboration. Lack of elaboration was pinpointed by excessive addition and removal of blocks in the workspace. Two participants, F2 and M3, were particularly affected by this problem (see black in Figure 3.3 for F2 and M3). M3 encountered the difficulty of not knowing what block to use for a desired behavior multiple times. Because of this, he excessively added and removed blocks leading to a failure to elaborate on potentially successful ideas based on those blocks.

M3: "So I don't really know what blocks to use… Cinema-TopTen came up before - I don't know if it's useful or what it does but I'll try it… There's nothing… I'll see what Cigarettes is 'cause that seems interesting…"

Some participants demonstrated the opposite behavior, attempting to elaborate excessively but failed to gain benefits from doing so. In particular, in trying to refine her existing ideas, F4 were often times unsure of how to use a block or how to make blocks work together, but regardless of her difficulties, she persisted. These difficulties prevented her from being able to elaborate effectively on her ideas and in turn led to difficulties in interpreting the output from her program. The following example shows that when she was uncertain about how to use a block, she randomly fiddled with the block's settings, and thus failed to elaborate effectively.

F4: "Do I need to change the source for all of these <parameters for operation>?" [Sighs. Changes the settings back and forth. Runs] "Why doesn't it show? I don't know what I'm doing wrong." [Keeps on trying without success]

Implications: Both phenomena of under- and overelaboration highlight mashup environments' lack of support for various levels of design from the abstract to the concrete.

We suggest that under-elaboration lies in the possibility that participants may have perceived elaboration as more costly than simply choosing another idea. The reason might be that the environment encouraged detailed implementation too early. Similarly, we argue that over-elaboration is linked to the same issue with the environment. For example, in order to test a tentative idea that a block might be useful, rather than being able to make the high-level assumption that it is useful and proceed with the rest of the design, the user had to go all the way, specifying various settings for the block and integrating it with the rest of the mashup in order to test that idea. In cases where an environment solely provides detailed implementation mechanisms, there is a risk of users to be lured into "trying to make it better" rather than thinking about the bigger picture. This phenomenon is particularly important in design (Simpson & Viller, 2004) and has led to an important body of research on supporting sketching in computerized design practice (e.g., (Do & Gross, 2007)). Similar effort has been made to support "sketching phases" in user interface development (e.g., (Landay & Myers, 2001)). However, these systems have been targeted to professional designers rather than casual end-user programmers.

3.5.3 Backtracking: explore ideas in parallel and what else?

Backtracking refers to instances in which participants returned to a previous state of the mashup after exploring other ideas. We identified these instances by diagramming ideations in the workspace in a time-wise fashion. All of our participants backtracked multiple times. There were three types of backtracking: to pursue alternative ideas, to revert back to a more successful state, and accidentally re-entering a previous state. First, some participants were trying to experiment with multiple idea alternatives, but as with almost all programming environments, there was no support for this. For example, within only three minutes, M3 backtracked to the same state three times (Figure 3.4). With each trial, he experimented with a block for retrieving/displaying pictures that he hadn't used before. M3's experimentation would have been less time-consuming if he could have done his experiments in parallel and compared results side by side. The second way participants used backtracking was to retreat from a path, getting back to a "safe" state. Once back in a working state, the participant usually went back to framing, to think of other ideas to try. But this tended to be error-prone, because sometimes participants had trouble recalling the exact details of that state.

M2: "So how was this working before?"

M5: "I'm gonna save more often now, like if I screwed up I could still get something to come up...."

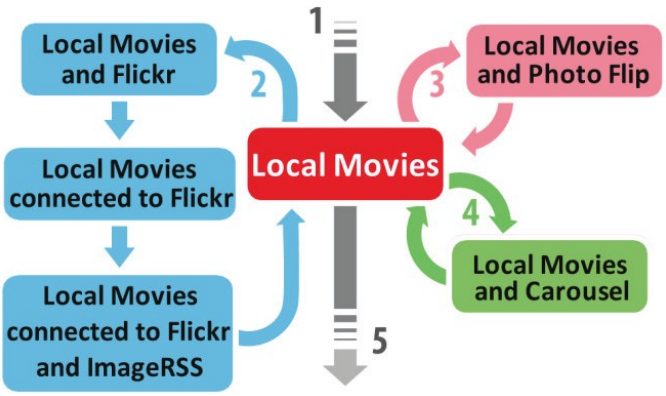


Figure 3.4 M3's backtracking over three minutes. Each box represents a different state of his mashup. The numbers show the order in which M3 (re-)entered and exited LocalMovies.

The third way participants backtracked was by accident, trying to find their way without meaning to return to previous states. F4 did this. Although she did not intend to backtrack, she sometimes recognized a state when she stumbled into it again.

F4: [Adds GeoNames. Hovers over it. Reads description] *"Get latitude and longitude. Oh I keep on doing that…"*

Implications: Similar to designers (Myers, Park, Nakano, Mueller, & Ko, 2008), end users would benefit from the ability to explore ideas in parallel. Systems featuring this capability have been implemented for professionals, e.g., (Terry & Mynatt, 2002). However, such systems remain largely absent from end-user programming, with a notable exception in (Hartmann, Yu, Allison, Yang, & and Klemmer, 2008). Additionally, programming environments should support end users' need to return to an earlier salient step in their design, for instance by permitting them to bookmark their exploration. Our participants relied on their memory to do this, which was error-prone. Finally, since backtracking is detectable, it might be possible to gently map the user's journey through state space, to avoid the wasted effort of returning to a state multiple times by accident.

3.6 Result 4: Reflecting upon Acting

Analyzing participants' reflection phases, their actions in between reflections, and the barriers they had in understanding their programs' output, we identified two salient issues with the support of reflection in Popfly. First, some participants carried out a large number of actions before they reflected and thus missed out on the opportunity to identify the impact of each action. Second, once the mashups' results were shown, participants lost the ability to refer back to the program's logic (as the Edit interface was separate from the Run view), and hence they could not efficiently debug. Not surprisingly, nearly all participants experienced difficulty knowing why the program did what it did.

Actions were reflected by ideations in the workspace. For example, having an *idea for block* meant adding a block to the mashup. Thus, upon visually exploring the occurrences of the ideations codes, we noticed that F4 underwent many actions before reflecting on them. For each participant, we then calculated the number of actions carried before an evaluation of the mashup, i.e., running it. Figure 3.5 provides a visualization of the number of ongoing actions between runs. F4 clearly stands out, as she made many changes (46 actions) to her mashup before the first run which happened at minute #26. To a lesser extent, F2 and M5 demonstrated a similar pattern (minute #33 and minute #13 respectively).

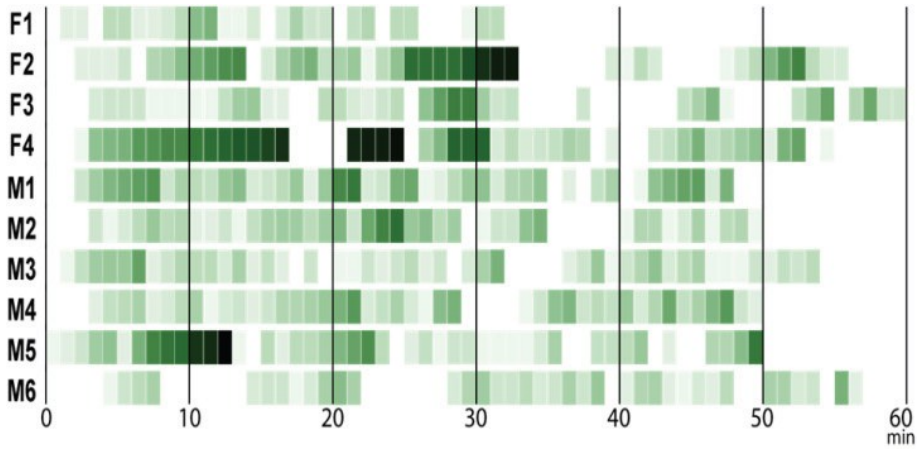


Figure 3.5 Accumulation of idea actions (expansions and contractions) before run. Color gets darker as idea actions accumulate. Color is reset to very light whenever participant hits "Run". (White spaces are pauses in which no actions take place.) For example, F4 accumulated 46 idea expansions and contractions before she first hit "Run" in minute #26.

These participants had difficulty evaluating which action caused the changes in results. M5 eventually recognized that this strategy was not working for him, realizing that his program had become so complex, he could not debug it. At that point, he started removing blocks, which rapidly turned his progress around.

M5: "Simplicity"[Runs. Theater and movie info shows up.]

"Oh, ok. There we go. I was getting way too complicated."

"It works well to run the program at each step."

In contrast, the most successful participant, F1, reflected upon her actions frequently. In this example, she did only two actions (adding a block and selecting an operation for it) before she reflected by running her mashup:

F1: "Ooh, Local Movies" [Adds it. Looks into block's settings]

"getTheaters <AndMovies>"<default operation>

"So I just hit run" [looks at results] "Theaters. Ok so I have a really long list of movies. And show times."

Additionally, we noticed that five participants had on occasions no actions carried out between runs. One possible reason was that the environment separated the mashup's output from its logic so when running the mashup, its logic was no longer available for reference. Thus, participants had to memorize one screen before switching to the other. In fact, three participants took notes on the outputs before going back to editing the mashups. Memorization is taxing, and the cost of running could also have deterred participants from frequent runs to enable reflection.

Implications: These phenomena suggest two implications. First, there is a need for mashup environments to not only reduce effort of running per se but also the effort of marrying the runtime output with the program's logic itself. The environment could for instance provide micro-evaluations of local portions of the mashup during the implementation phase (e.g., what is the output of this particular block if I set it up like this). Moreover, the environment could provide references between the outcomes of the mashup and its logic, by highlighting

relationships between them. However, solutions to these problems are inherently difficult for mashups, since their performances also rely on remote providers of information that may not always be responsive.

Second, while professional programmers and even novice computer science students get a lot of practice honing their problem-solving strategies for debugging such as isolating variables, end-user programmers may not have developed debugging strategies like these. Tools for debugging by end-user programmers could provide hints for debugging strategies, as demonstrated in spreadsheet software (Grigoreanu, et al., 2008).

3.7 Summary

In this study, we have presented a design- and creativity-theory based approach to investigating end users' programming behaviors. We demonstrated the usefulness of this approach by applying the reflection-in-action theory from design and the ideation notion from creativity literature to the think-aloud protocols from ten participants creating web mashups. As summarized in Table 3.2, the results revealed ample opportunities for an Idea Garden to better support end-user programming as a design and a creativity activity.

Therefore, this study delivers the following contributions:

(1) The methodology of applying a theory-based design perspective to programming,

(2) Evidence of the usefulness of using this approach through insights gained, and

(3) The insights themselves into end-user programmers' problem-solving barriers and attempts, with implications for design of an Idea Garden for end-user programming environments. Implications included support for framing output ideas, for effective reflection, and for exploration of multiple alternative ideas in parallel.

Table 3.2 Barriers from the perspective of reflection-in-action, implications, and candidate Idea Garden features.

Barriers	Implications	Example Idea Garden Features
(**Framing**) No actionable ideas as output from framing efforts.	System needs to provide users with ideas to start with if they do not have any of their own, e.g., through examples or cases.	(Idea Garden prototype) Features that help the user find the right examples or cases and make use of them. (See Section 5.3.)

(**Framing**) Reluctance to reframe due to low self-efficacy.	"What-if" features might help with inflexibility and unwillingness to reframe for low self-efficacy users.	(Idea Garden vision) A tool that would allow users to make assumptions about what a block will output might enable users to explore assumptions in multiple ways. For example, one way might be to focus on testing the assumption. A second way might be to focus on companion blocks compatible with that assumption.
(**Acting**) Tinkering without reflecting due to high cost of reflecting.	For tinkering to benefit from reflecting, the cost of tinkering and the cost of reflecting need to be balanced.	(Idea Garden vision) Features should make it easy for users to evaluate the output of their programs by lowering the cost of reflecting. This hopefully will lead to more reflecting while tinkering and thus improve the quality of tinkering.
(**Acting**) Under- and over-elaboration.	Tools should support various levels of design ranging from the concrete to the abstract.	(Idea Garden vision) Features that allows the user to create a program at different levels of abstraction and evaluate design for each level, and facilitates the navigation between the levels.
(**Acting**) Backtracking to explore multiple ideas.	Tools should allow users to explore multiple ideas at once,	(Idea Garden vision) Features that make it possible for users to pursue multiple ideas in parallel and compare them against each other.
(**Acting**) Backtracking to revert to a more successful state.	Tools should keep track of where the users' program has been to make it easy for users to go back to previous states of their programs.	(Idea Garden vision) Features that records where the user has been in his/her program, allows the user to tag whether a state is desirable or not, and to easily revisit any previous state of their program.
(**Acting**) Backtracking by accident.	Tools should make users aware of the fact that they are entering the same state for multiple times.	(Idea Garden vision) When a user's program is the same as some previous version of it, the system should let the user know. In addition, the system should also notify the user when he/she has spent much time in a state.
(**Reflecting**) Accumulating many ideas before reflecting due to the high cost of reflecting	The environment should lower the cost of reflecting.	(Idea Garden vision) See "Tinkering…" In addition, features could allow "micro" evaluation of part of the program to provide opportunity for immediate feedback and to avoid having to evaluate the entire program.
(**Reflecting**) Difficulty in referencing the code when looking at output and vice versa.	Tools should marry the output with the code for easy reference.	(Idea Garden vision) Features that allow the user to see the code next to the output from the code.
(**Reflecting**) Difficulty in understanding the program's behaviors.	Tools should help users understand the output from their code.	(Idea Garden vision) Features could help users make use of strategies to make sense of the output from their code.

Chapter 4. Formative Study 2: Understanding End-User Programming from the Lens of Problem-Solving

In Chapter 3, we presented barriers participants encountered and implications for the design of potential features for an Idea Garden discovered through the lens of the reflection-in-action theory from design and the ideation notion from creativity. A pervasive theme among the barriers we identified was the lack of problem-solving ideas when participants faced a barrier. For example, some participants attempted to reframe the programming problem multiple times, only to end up struggling with barely any new ideas at all.

Therefore, in this chapter, we consider this issue from another perspective, namely, problem solving. In particular, we conducted a study in a second end-user mashup environment, IBM CoScripter (Lin, Wong, Nichols, Cypher, & Lau, 2009). We analyzed the new study's data using the problem-solving literature as a lens to understand end users' programming behaviors. Also, to promote the generality of our results, we triangulate the results with a re-analysis of the data from the Popfly study focusing on problem solving. This analysis targeted the following research questions:

> *RQ 1: Why do users experience such great difficulties in problem-solving their way to generate new ideas?*
>
> *RQ 2: What design opportunities for an Idea Garden are revealed by applying problem-solving theories to the results of RQ 1?*

4.1 Empirical Study

We obtained the data by asking end users to perform programming tasks while we observed. For the Popfly environment, we had already collected such data for another purpose (Cao, Riche, Wiedenbeck, Burnett, & Grigoreanu, 2010) and were able to reanalyze the data set, as we describe in detail below. For the CoScripter environment, we had not yet conducted such a study and needed to collect a new data set. Our results (Section 4.2) are based on our qualitative analysis of both data sets together.

4.1.1 Environments

We have described the Popfly environment in Section 3.1.3.

CoScripter is an end-user programming-by-demonstration environment for web scripting and mashup building (Lin, Wong, Nichols, Cypher, & Lau, 2009). In CoScripter (Figure 4.1), users demonstrate to the system how they would carry out a task on the Web (e.g., by filling out a form online to reserve a shuttle ticket to the airport). The system watches and translates users' actions into an editable script (Figure 4.1: script), which the user can execute at a later time to perform the same task again.

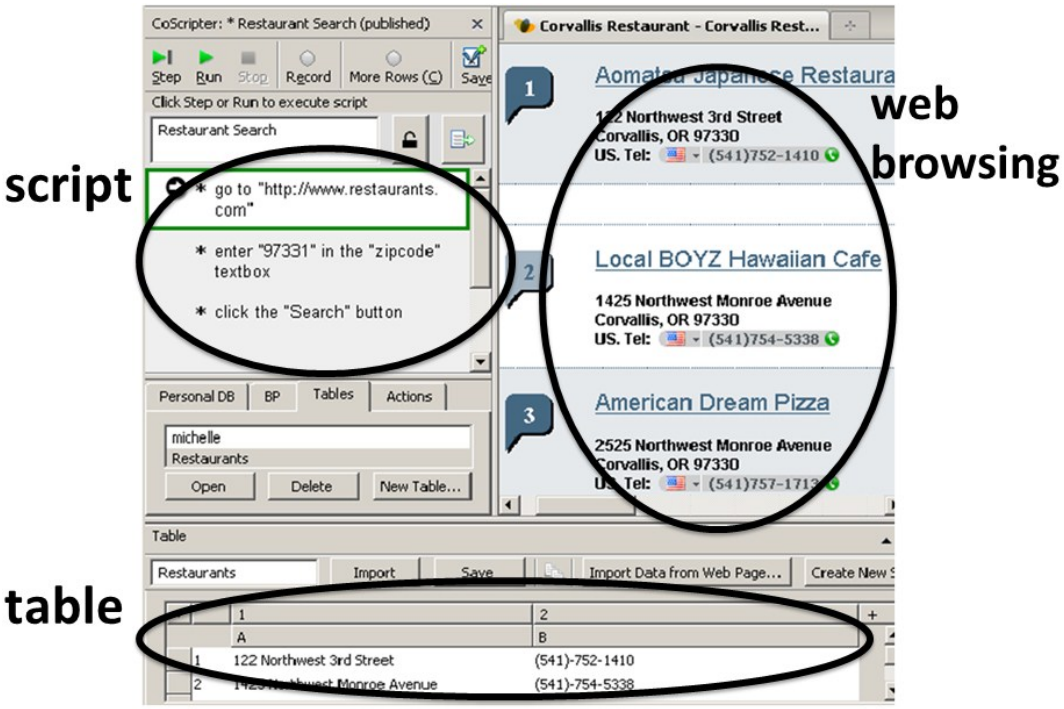


Figure 4.1 The CoScripter environment with three main parts, i.e., the script area, the table area and the web browsing area.

CoScripter enables mashup programming via a table feature (Figure 4.1: table). Users can create scripts that automatically copy data between the table and existing web pages. This enables the user to combine data from multiple web pages in the table and to flow text from one web page to form fields of another.

4.1.2 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*).

4.1.3 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 rent for under $800 per month and are within a 30-minute drive of the Oregon State University campus; this required combining data from a search site, such as Craigslist or Apartments.com, with data from a maps site, such as Google Maps. The Popfly participants' task was to create a mashup that integrated movie-related information, such as movies currently showing at local theaters, theater information, and news stories about each movie.

4.1.4 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-minute hands-on tutorial that familiarized them with the mashup environment. They then completed a self-efficacy questionnaire (Compeau & Higgins, 1995) adapted to the task. Participants then practiced "thinking aloud" before proceeding to the main task. Finally, they had 50 minutes 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 fifteen minutes, the researcher administered an extra five-minute 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 three minutes, the researcher prompted them with hints, such as suggesting that they try a different website or try using the table; all these participants received at least one hint.

We collected audio and video of the participants as they worked on the main task as well as video capturing their screen activity. We analyzed the audio, video and screen-capture data using qualitative analysis methods based on grounded theory (Strauss & Corbin, 1998). Because our goal was to understand problems encountered, we focused most specifically on when participants struggled. In particular, our analysis focused on episodes where participants encountered barriers such as when a participant verbalized a need for help, turned to the programming environment's help facility, asked the researcher for help, or when the researcher offered assistance observing a participant struggle. Our analysis revealed different patterns of participant responses to barriers, demonstration of a lack in problem-solving strategies, and programming knowledge. We triangulated findings across participants and environments to identify the most pronounced trends, which we report below.

4.2 Results and Implications

As we mentioned Section 2.2.3, Simon believes that both domain-specific knowledge and general problem-solving strategies are necessary to solving problems successfully in a domain (Schön, 1983). Thus, we organize our results along these two skill sets, and triangulate with applicable theories related to problem solving.

4.2.1 Problem-Solving Strategies

A strategy, according to Webster Dictionary, is a careful plan or method for achieving a specific goal. According to literature on problem solving (e.g., (Jonassen, 2000)), the adoption of problem-solving strategies are affected by metacognitive skills, beliefs, and expertise. We discuss the first two of these here, and since expertise maps to Simon's concept of domain-specific knowledge, which we discuss in the next subsection.

Metacognition. Simply put, metacognition is thinking about thinking (Flavell, 1979), for example, a person's conscious awareness and monitoring of his/her own thoughts, understanding, or learning. Literature on problem solving shows when a person is stuck on a

problem, it is important to step back and analyze what he/she has been doing (Wickelgren, 1974).

Some participants exhibited very little metacognition while struggling to solve problems, and they largely failed to overcome barriers. For example, CoScripter-F2 showed no sign of metacognition regarding her problem-solving strategies. For example, 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 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]*

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."*

Self-efficacy. As to beliefs, one kind of belief that can affect the adoption of problem-solving strategies is self-efficacy, a person's confidence in their ability to succeed at a specific kind of task (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 consider switching from a straight Google search to using the VegeTable to help with her task until the researcher prompted her:

> *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]! "*

Design opportunities for supporting problem-solving strategies *What* the tool features should consist of is also an open question, particularly considering that some users might rightfully have low self-efficacy because they have a small repertoire of problem-solving strategies or poor knowledge: it is not enough for a prompt to simply say, "Break the problem into subproblems!" because some users might not know that strategy. Therefore, support for strategy use should also help users to understand, effectively choose among, and follow through on strategies in the context of a current programming task.

Bloom and Broder asserted that the "habits of problem solving, like other habits, could be altered by appropriate training and practice" (Bloom & Broder, 1950). This leads to the possibility that suggesting an appropriate problem-solving strategy at moments of difficulty could nudge end-user programmers' program problem-solving skills up enough to enable them to form new ideas themselves.

To more precisely pin down how tool features could prompt and support use of problem-solving strategies, we constructed a list of problem-solving strategies that a tool could be aimed at supporting. We identified these strategies by examining relevant problem-solving and creativity literature (Wickelgren, 1974; Polya, 1973; Levine, 1994; Osborn, 1953), then identifying five strategies that were cited in multiple sources and that covered a breadth of situations.

Working Backward: The problem solver starts with the goals of the problem in an attempt to reason back to the givens of the problem (Wickelgren, 1974; Polya, 1973). One way to bring this strategy to end-user programmers could be to ask them to start by envisioning the output of the desired program, and then work backwards from there to figure out what could lead to the output. A tool could further aid this reasoning by helping programmers to understand how specific outputs are related to specific inputs, such as through effective explanations and/or visualizations. Along the way, it could explain the associated input/output concepts for those users who lack the necessary foundational knowledge.

Divide and Conquer: The problem solver breaks the original problem into parts, solves each part individually, and combines these solutions (Soloway, 1986; Wickelgren, 1974). In end-user programming environments, encouraging a user to tackle part of the problem might provide insights on a potential overall solution, as well as increase self-efficacy once the user solves a subproblem. A tool could also assist by providing features for recognizing subproblems, tracking solutions to subproblems, and synthesizing these into a whole

Analogy: The problem solver relates the problem at hand to problems previously solved in the past (Osborn, 1953) (Polya, 1973). Polya mentions two ways a problem solver may leverage a solved problem in solving an unsolved problem (Polya, 1973): (1) use the solved problem's results, and (2) use the method for solving the solved problem in solving the unsolved problem. A programming tool could support this strategy by providing a collection of "Starter Ideas" that the user could extend—thereby also directly helping to address the lack of ideas noted for some participants, above. (Templates and solution "seeding" (Fischer, 2009) are essentially specific implementations of this approach.) The tool could also provide a catalog of ideas that are *similar to* or *in contrast to* one another, which could be helpful because producing new ideas often depends upon mixing different, often contrasting ideas (Osborn, 1953).

Generalization: The problem solver starts by considering one instance of the problem, solving that instance, and then adapting the solution to encompass larger sets of instances (Polya, 1973; Wickelgren, 1974). Trained computer scientists will recognize induction/recursion as useful examples of Generalization. An end-user programming tool could support this strategy by providing features for the user to solve a single instance, then providing language features for trying the solution on other problem instances. The tool could also explain associated

programming concepts; for example, if applying the solution to multiple instances involves iterating over them or passing the instances' data to a function, then the tool could explain the concepts of iteration and functions. In addition, to support the strategy, it could summarize or visualize which instances are not yet solved to identify directions in which the solution needs to be further generalized.

Sleep on It: The problem solver sets aside a difficult problem and comes back to it later, possibly with a fresh perspective (Osborn, 1953; Polya, 1973; Levine, 1994). Bringing this strategy to end-user programmers may be as straightforward as encouraging a stymied user to simply set aside the difficult subproblem and focus on other subproblems for a while. The tool could provide a feature for remembering to return later on.

These and another strategies point toward opportunities for enhancing tools, e.g., our Idea Garden, to more effectively enable users to overcome barriers during end-user programming.

4.2.2 Programming Domain Knowledge

Problem-solving strategies cannot be applied without basic knowledge in the domain where those strategies are to be applied (Simon, 1980). End users are by definition domain experts, and the goal of end-user programming is to bring their domain expertise directly to the 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 programming domain (e.g., concepts of input and output, how to go about debugging), or in the language the user is trying to use (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, we might not expect issues of programming expertise to arise. Interestingly, however, issues of programming knowledge arose many times and at multiple levels.

Language construct. At the language construct level, CoScripter-M1 had trouble figuring out where the "repeat" command should go when he wanted his script to loop through the rows in his table, a skill learned early by successful students of computer science:

CoScripter-M1: "So I got my results [in the table]. I guess you can repeat it then."

[Adds "repeat" to the beginning of his script, which tells the script to repeat every line instead of just table computations]

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

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

Likewise, several participants from the CoScripter study failed to grasp the ideas of using multiple webpages, 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.

Program design ideas. 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 below), running out of ideas to try very early (CoScripter-M2 below), or choosing a starting idea that leads down a wrong path (Popfly-F4 below).

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

CoScripter-M2: [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]

The above three examples have in common a shortage of ideas that would be available to those with more expertise in the domain of programming. The notion of *ideational fluency* from the creativity literature suggests that scarcity of ideas is a problem-solving disadvantage, and that the more ideas a user has, the greater chances of him/her arriving at useful and creative solutions to a given problem (Osborn, 1953).

Design opportunity for supporting programming domain knowledge.

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

Relating to the first example (CoScripter-F2 above), when a user has no idea to start with, there is a clear opportunity to help them gain momentum with Starter Ideas, a concept similar to Fischer's "seeding" (Fischer, 2009). The possibilities for such ideas could include strategy ideas (e.g., "try starting with a sketch of the output" or "look at all the blocks that produce maps") or very specific ideas ("a lot of people use the Google map page in problems involving addresses").

Now consider CoScripter-M2 and Popfly-F4, who had ideas but ran into trouble and did not know how to move on. In creativity literature, Osborn suggests that producing new ideas depends upon "mixing" ideas, often catalyzed by associations through three "laws": contiguity, similarity, and contrast (Osborn, 1953).

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

An example of contrasting ideas that a support system could suggest would be using a table widget instead of using a photo widget to display pictures, the result of which would be quite

different. A more drastically different idea would be to leave out the display widget entirely to see what happens.

Contrasting ideas may encourage users to think outside the box. In creative design literature, "design fixation" (Jansson & Smith, 1991) refers to the undesirable situation where the designer becomes overly focused on one idea, missing out on other opportunities. Indeed, in (Cao, Riche, Wiedenbeck, Burnett, & Grigoreanu, 2010), we found some users reluctant to relinquish ideas that were not working for them. This hindered users' willingness to redefine the design problem to approach it from a different perspective, termed "reframing" in design literature, a critical step in design problem solving (Schön, 1983). This also resulted in "over-elaboration" (Cao, Riche, Wiedenbeck, Burnett, & Grigoreanu, 2010), continually elaborating on an idea that cannot ever work. Contrasting ideas may help to avert some of these problems.

4.3 Summary

In summary, by using the problem-solving literature as a lens to understand end users' programming behaviors, we found a lack of problem-solving strategies and programming-domain knowledge to be prominent in users' struggles to generate new ideas to overcome programming barriers. Among our results were reasons *why* our participants encountered "ideas barriers" including a lack of metacognition, low self-efficacy, and little knowledge of program design patterns. Based on these results, throughout Section 4.2, we have proposed design implications for an Idea Garden to support both problem-solving strategies as well as programming knowledge. For example, we suggested that for users who struggle to get started on a program, the Idea Garden could offer the user a starting idea to "seed" their efforts. Table 4.1 tabulates all results and implications.

In summary, this work makes the following contributions:

(1) A demonstration of how to apply theories from problem-solving literature to understand ideation barriers in end-user programmers' problem-solving processes.

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

Table 4.1 Problem-solving barriers, design opportunities, and candidate Idea Garden features

Barriers	Design Opportunities	Example Idea Garden Features
(**Problem-Solving Strategies - strategies**) Reluctance to switch strategies.	A tool could remind the user of another approach, or teach users about new strategies they have not thought of.	(Idea Garden prototype) A feature that gently nudges the user into adopting a new strategy.
(**Problem-Solving Strategies - metacognition**) Missed to inspect one's approach	Environments could support metacognition, e.g., by cuing the users to think about their approach rather than the problem itself.	(Idea Garden prototype) A feature that recognizes what approaches the user has tried and cues him/her to think about their approach when it sees them spending much time on one approach.
(**Problem-Solving Strategies – self-efficacy**) Little support for low self-efficacy users	The environment could support users with different levels of self-efficacy, especially those with low self-efficacy.	(Idea Garden vision) The Idea Garden should assume a non-authoritarian tone when making suggestions to the user.
(**Programming-Domain Knowledge – Programming constructs**) Misunderstanding of constructs or no know about constructs.	The environment could support the learning of constructs in the context of users' work.	(Idea Garden prototype) Features that suggest programming constructs to the user at the right moment, e.g., a feature might suggest the "repeat" construct to the user when he/she has operated on one row of their table.
(**Programming-Domain Knowledge – Design patterns**) No or little knowledge of design patterns.	The environment could support the application of patterns in the context of users' work.	(Idea Garden prototype) Features that suggest programming patterns to the user at the right moment, e.g., a feature might suggest the "repeat-copy-paste" pattern the user when he/she has operated on one row of their table.
(**Programming-Domain Knowledge – Program design ideas**) No starter ideas.	The environment could help users get started.	(Idea Garden prototype) A feature that provides users with starter ideas that users can elaborate on.
(**Programming-Domain Knowledge – Program design ideas**) Running out of ideas early on.	The environment could help users come up with new ideas.	(Idea Garden prototype) A feature that provides users with additional ideas when they run out of ideas. For example, the additional ideas can take the form of contrasting ideas or similar ideas.

Chapter 5. An Initial Idea Garden

Motivated by the findings of the two formative studies (as described in Chapter 3 and Chapter 4), we devised the Idea Garden concept.

The Idea Garden concept's goal is to help end users generate new ideas and problem-solve when they are working on a self-directed programming task. This goal contrasts with those of existing approaches (described in Section 5.1): we do not seek to change the programming language to make it simpler or more natural, do not seek to automatically eliminate or solve problems for the user, and do not seek to delegate programming responsibilities to someone other than the end users themselves. Rather, we seek to help end-user programmers to generate their *own* ideas to overcome programming barriers. Our strategy for doing so is to provide scaffolding for problem-solving strategies and programming-domain knowledge such as design patterns and programming concepts.

In the rest of the chapter, we first contrast the Idea Garden concept with existing approaches (Section 5.1). We then explain its conceptual architecture (Section 5.2) and describe a proof-of-concept prototype of it for the CoScripter environment (Section 5.3). Finally, we explain the design principles behind the prototype (Section 5.4).

5.1 Related Work (Aiming to Make End-User Programming Easier) and How the Idea Garden Differs from It

A central goal in the design of end-user programming environments has historically been to attempt 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, Pane, & Ko, 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 & Myers, 2006). An empirical study demonstrated that this language significantly increased users' ability to complete programming problems (Pane & Myers, 2006). Another system, Vegemite (an add-on to CoScripter (Little, et al., 2007)), aims to make

mashup-programming easy by providing table data structures akin to spreadsheets that remove the need for explicit iteration constructs (Lin, Wong, Nichols, Cypher, & Lau, 2009). In an early study, users said it would be easy for them to create new mashups with the system (Lin, Wong, Nichols, Cypher, & Lau, 2009). Still other programming environments, such as Alice (Kelleher & Pausch, 2006) and AutoHAN (Blackwell & 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.

Other approaches attempt to automatically eliminate programming barriers for users. AgentSheets (Repenning & Ioannidou, 2008) and Koala/CoScripter (Little, et al., 2007; Lin, Wong, Nichols, Cypher, & Lau, 2009) are based on programming by demonstration (Lieberman, 2001; Cypher, Dontcheva, Lau, & J. Nichols, 2010), an approach where end users demonstrate an activity from which the system automatically generates a program. Some 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. Chickenfoot aims at automatically correcting errors, enabling the user to enter a mostly correct program from which the tools automatically infer the desired behavior (Miller, et al., 2010). Marmite provides templates that describe classes of nearly finished programs that users complete by simply filling in the blanks (e.g., (Wong & Hong, 2007)). A similar approach is for tools to provide examples from a set of recommendations that users can copy, paste, and tweak, e.g., (Hartmann, MacDougall, Brandt, & S. Klemmer, 2010). In fact, some tools offer a menu of code snippets that can be automatically generated on demand, e.g., Intel Mash Maker (Ennals, Brewer, Garofalakis, Shadle, & Gandhi, 2007). User studies in the papers cited above demonstrate that by simplifying the programming task, these and similar tools help people to complete programming tasks more quickly with fewer errors.

Some approaches instead delegate some of the programming responsibilities to other people. For example, meta-design (Andersen & Mørch, 2009; Costabile, Mussio, Parasiliti Provenza, & Piccinno, 2009; Fischer, 2009) 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. In some large organizations, an

expert end-user programmer, called a gardener, serves to ease or eliminate programming among the organization's end-user community (Gantt & 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 & 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, Dontcheva, Weskamp, & Klemmer, 2010), is an Adobe Flex Builder plug-in that semi-automatically gleans task-specific example programs and related information from the web, 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, Herstand, Hodges, & Kelleher, 2010; Scaffidi, et al., 2009)).

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. Succinctly put, 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 is little work aimed at helping professional interface designers generate and develop ideas. For example, DENIM (Newman, Lin, Hong, & Landay, 2003) is a system that allows designers to sketch web sites at a high level and thus helps the designers develop their design ideas. As another example, Díaz 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 limitation of the suggested patterns (Díaz, Aedo, Rosson, & Carroll, 2010). Our approach was inspired in part by this line of work. In particular, we seek to help end-user programmers generate new ideas and problem solve during their programming tasks.

5.2 The Idea Garden's Conceptual Architecture

The Idea Garden is intended to *extend* existing end-user programming environments. The existing environment thus serves as a "host" to the Idea Garden features.

The Idea Garden prepares context-based suggestions and makes them available to users of end-user programming environments (e.g., CoScripter or Popfly). The host environment provides the Idea Garden with a stream of information that includes the user's code, data, and recent activities. With this information, the Idea Garden infers as much as it can about the user's current context (primarily via off-the-shelf components) in order to construct appropriate suggestions for the user. It waits for opportunities to unobtrusively make the availability of new suggestions apparent in the host environment so that the user can view them at convenient times, if desired. Figure 5.1 depicts this conceptual architecture.

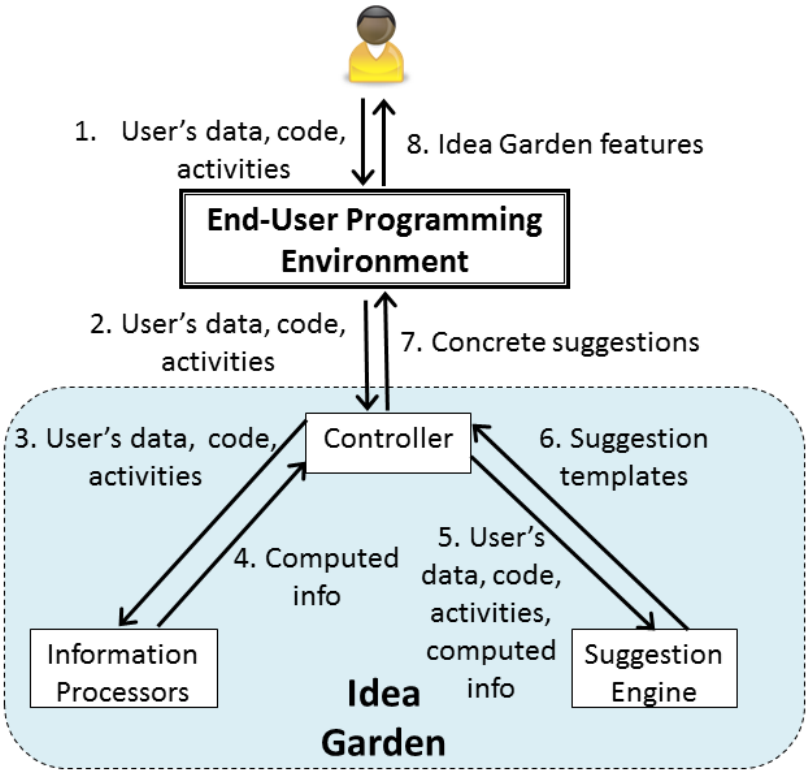


Figure 5.1 Conceptual architecture for the Idea Garden.

As the figure suggests, applying our approach requires a component, the "Controller" in Figure 5.1, which retrieves users' code, data, and activities from the environment. The

Controller then passes that information to Information Processors to compute additional information, which in turn passes the complete collection of information to a Suggestion Engine. The Suggestion Engine then tells the Controller what suggestions and/or widgets to insert back into the host environment.

5.3 An Initial Idea Garden Prototype for CoScripter

In an 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 of our formative studies and in prior end-user programming literature (e.g., (Ko, Myers, & Aung, 2004)).

5.3.1 Interacting with the Idea Garden Prototype

Our Idea Garden prototype as it appears with CoScripter as the host communicates with users via the Gardener (Figure 5.2: The Gardener). The Gardener understands the user's problems in CoScripter about as much as a garden shop owner understands problems in a customer's garden: a lot in general, but not that much about that particular customer's soil, neighboring plants, resident insects, etc.

For example, when the user starts CoScripter, the Gardener icon in Figure 5.2 appears. If the user clicks, the Gardener offers suggestions for how to proceed, such as the one in Figure 5.3. Electing to follow the suggestion will get the user started using a CoScripter table. Figure 5.4, Figure 5.5, and Figure 5.6 show other suggestions the Gardener can make.

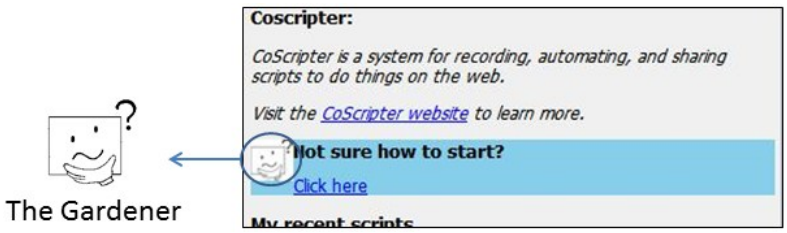


Figure 5.2 The Idea Garden inserts suggestions (highlighted text) directly into the CoScripter user interface.

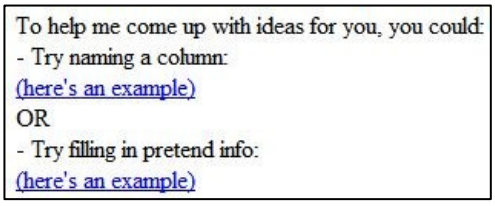


Figure 5.3 Start-with-a-column-name: This suggestion targets the *How-to-Start* barrier. The user can view it by clicking a special "ideas cell" in the CoScripter table area (not shown). The idea is to nudge the user into the beginning step of working backward from the ultimate goal; the next suggestion, Figure 6, encourages the next step.

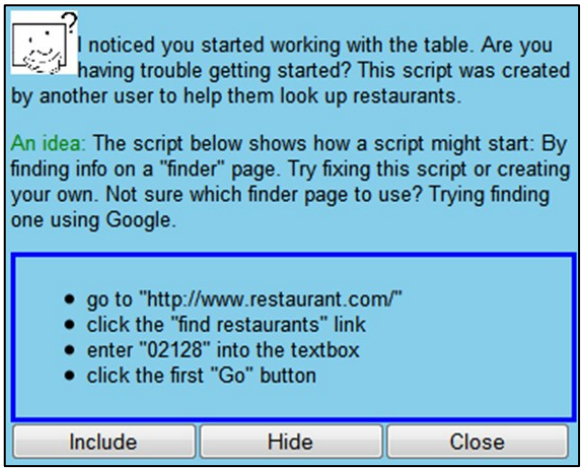


Figure 5.4 Finder-page: If the user names a column with empty cells and clicks or hovers on the Gardener icon in the script, the Gardener suggests examples of a design pattern, which we call the *Finder pattern,* to populate the column.

Figure 5.5 Second-webpage: 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).

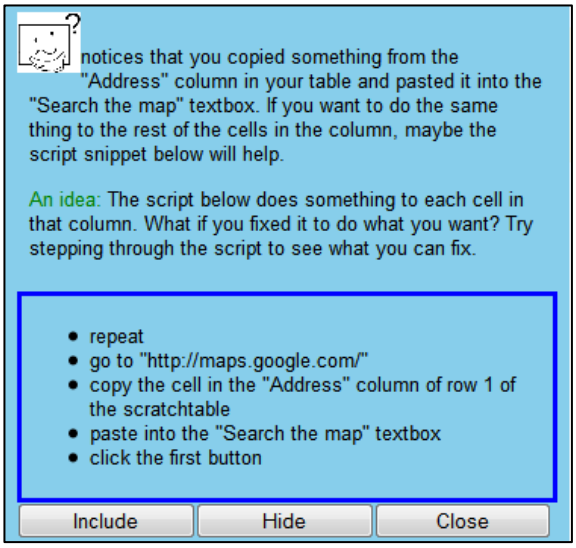


Figure 5.6 Generalize-with-repeat: When a user has copied and pasted one cell of a column into a web page to retrieve a value, the last recorded line of the script displays the Gardener icon. The user can click it to view the suggestion targeting the *More-than-Once* barrier. The script example in this suggestion is incomplete so that the user must actively engage with it by editing code.

Minimalist Learning Theory (Carroll & Rosson, 1987) which targets the design of instructional materials for "active" computer users in the midst of some task of their own

emphasizes the importance of the linkage of the materials to the actual system where the user is working, so the Gardener augments the host environment's usual display (here, CoScripter). However, the Gardener does not pop up uninvited, like the infamous Clippy of Microsoft Office would. Instead, the Gardener icon follows the Surprise-Explain-Reward approach (Wilson, et al., 2003), which includes the negotiated style of interruptions (Robertson, et al., 2004): it decorates the environment's user interface (either in unused whitespace or by extending the containing window), only appearing or disappearing when the relevant CoScripter window refreshes after a user operation, or when the user invites the Gardener by hovering or clicking on the icon; thus, it never interrupts the user. Thus, as per Surprise-Explain-Reward, the indicator is meant to entice users in need of more information to pursue the explanatory material it hides, and the explanatory material is meant to conjure an image of the benefit of following the particular suggestion.

5.3.2 Suggestion Types and Structure

The Idea Garden prototype supports four kinds of suggestions, each of which is structured as shown in Figure 5.7. The top section of Table 5.1 shows which suggestions target which barriers. The lower sections of this table indicate of the prototype's suggestions target each programming concept, design pattern, and problem-solving strategy. Future prototypes could explore other concepts, patterns and strategies.

Figure 5.7 Suggestion structure. (1) The Gardener "wonders" about context and (2) comments on this context to provide rationale for concrete examples in (4). (3) The Gardener summarizes the gist/essence of the idea and (4) suggests concrete examples as action items. (Start-with-a-column-name (Figure 5.3) does not include (1) and (2) because user has already told the system their context by the time he/she invokes this suggestion.)

Table 5.1 The Idea Garden prototype for CoScripter provides suggestions that target three barriers. We designed suggestions with the intent of helping users with three programming concepts, two design patterns, and three problem-solving strategies. See Figure 5.2 through Figure 5.6 for screenshots.

		Suggestions			
		Start-with-a-column-name (Figure 5.3)	*Finder-page* (Figure 5.4)	*Second-webpage* (Figure 5.5)	*Generalize-with-repeat* (Figure 5.6)
Barriers targeted:	*How-to-Start*	X	X		
	Composition			X	
	More-than-Once				X
Programming concepts reflected in suggestions					
Data Extraction	Select a slice of structured data from a web page and putting it into the table		X		
Dataflow	Flow 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	Use a web page to find information (as opposed to computing information)		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 the end goal, then figure out the last step to the goal, second to the last step, and so on until the givens are reached.	X	X		
Analogy	Find solution to similar problem, then adapt it for problem at hand		X	X	
Generalization	Solve one instance of a problem and generalize the solution to all instances in the problem				X

Programming concepts. 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., 2010; Cao, Fleming, & Burnett, 2011; Ko, Myers, & Aung, 2004; Zang & Rosson, 2009). In the context of CoScripter, these concepts arise in the need for retrieving data from web pages, passing data between tables and web pages, and iterating through all the rows of a table.

Design patterns. The phrase *design pattern* is popularly used to refer to common ways of organizing object-oriented code to accomplish particular goals (e.g., as in (Gamma, Helm, Johnson, & Vlissides, 1995)). We apply the more general sense of the term design patterns (e.g., as in (Alexander A. , 1979)) to our CoScripter context. Thus, our design patterns are common ways that users structure their scripts. Prior research has not identified a full range of patterns commonly used by CoScripter users, but we have personally found the two patterns shown in Table 5.1 to be helpful when creating CoScripter mashups. Therefore, we have chosen these two for our initial prototype.

Problem-solving strategies. The Gardener's suggestions attempt to help users adopt problem-solving strategies from the problem-solving literature (Polya, 1973; Wickelgren, 1974; Levine, 1994). 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 Gardener on how to start the task, the Gardener 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 (Figure 5.3). The Gardener then suggests the next-to-last step (Figure 5.4), and so on.

The Gardener never tries to *entirely* solve users' problems. 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, it 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, 1953). 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.

5.3.3 Behind the Scenes

Inference in the Idea Garden about the user's context is in part environment-dependent, acting on environment-specific clues about what the user is doing, and in part independent of the underlying host environment, leveraging context-independent inference mechanisms. As an example of the latter, one of our prototypes draws upon the TopeDepot (Scaffidi, 2010), an environment-independent tool that infers data types. We use it to infer the types of data the user has entered into a CoScripter table column. Our prototype sends the TopeDepot a list of strings and/or a label that describes those strings (e.g., a column label), and the TopeDepot responds with a sorted list of possible data types. Our prototype discards all guesses, except for the top one. For example, given the cell value "Merrill Lynch" in a column with the heading "company," TopeDepot might, as its top guess, classify the column as containing company-name data.

TopeDepot sometimes fails to correctly guess the data type—but this imperfection aligns well with our goal of promoting problem solving by analogy. For example, incorrectly guessing that a person name is a company name and suggesting web pages for using company names will (hopefully) encourage the user to reason by analogy to adapt the idea for use with person names.

5.4 The Idea Garden's Design Principles

To guide the design of the Idea Garden features, we built upon the principles of Minimalist Learning Theory (Carroll & Rosson, 1987). Minimalist Learning theory targets the design of instructional materials for "active" computer users in the midst of some task of their own, and emphasizes the importance of task-focused activities. This theory is especially suited to the design of the Idea Garden features because the Idea Garden's goal is to help end users generate new ideas while they are working on their own, self-directed programming tasks.

Minimalist Learning Theory 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. The Idea Garden follows these design principles in the following ways.

5.4.1 Principle 1: Self-directed reasoning

Our approach contrasts with existing tools that attempt to solve users' problems automatically. Instead, the Idea Garden suggests strategy alternatives and provides (intentionally) incomplete or flawed suggestions, all of which require the user to actively reason and problem solve in order to make substantive progress on the task at hand.

As briefly mentioned in Section 5.3.1, the Idea Garden also 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, the Idea Garden uses negotiated-style interruptions, which inform users of pending messages but do not force the users to acknowledge the messages (McFarlane D. , 2002). Such negotiated-style interruptions leave the initiative to the user. Perhaps because of this, they 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).

5.4.2 Principles 2 and 3: Meaningful, self-contained, and realistic activities

In the Idea Garden, all suggestions are contextualized and task-oriented. Each suggestion is by definition tied to the task that the user has already chosen to initiate, thereby giving suggestions additional meaning, value, and realism.

5.4.3 Principle 4: Closely linked to the actual system

The Idea Garden is linked to a host environment by layering itself on top of the environment. The Idea Garden can layer itself to any end-user programming 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). Many programming environments,

including Excel and CoScripter, satisfy these constraints, providing pluggable architectures into which Idea Garden features can be added.

5.4.4 Principle 5: Error detection and recovery

Because our approach is intended as a layer added onto existing end-user programming environments, users can continue to take advantage of all assistance each host programming environment provides to detect and recover from errors.

Finally, we do not expect the Idea Garden to replace existing help systems or tutorials. Rather, the Idea Garden supplements such traditional materials, which will still be necessary for general training on the end-user programming environment.

5.5 Idea Garden Defined

We have explained the goals and principles behind the Idea Garden concept, and in this subsection, we formalize the Idea Garden.

We will refer to instances of the Idea Garden concept as an Idea Garden. An Idea Garden is:

- An extension to an existing programming environment
- that follows Principles 1-5 from the previous subsection, and
- includes all the elements and properties enumerated in Table 5.2.

Table 5.2 Elements and Properties of an Idea Garden.

Type of element/property	The elements and properties	Motivation
Feature Content	Problem-solving strategies (See Table 5.1 for example strategies.)	Suggested by our empirical results (Chapter 4) and Simon's problem-solving theory.
	Programming knowledge (See Table 5.1 for example programming knowledge.) This knowledge comes in the form of concepts and design patterns.	
	Intentionally flawed examples (e.g., Part 4 in Figure 5.7)	To avoid solving users' problems for them.
Feature Form	Suggestion template (Figure 5.7)	The suggestion template prescribes the content of the features in a host-independent way. (An Idea Garden implementor then fills in the template with host-dependent details.).
Interaction Style	Interruption style: negotiated	The negotiated interruption style works better than the immediate style for users working with programming problems (Robertson, et al., 2004).
	Personality: non-authoritative	The Idea Garden should not appear to be an automatic problem-solver or an authoritative figure that can dictate what the user should do. Users are in charge of their own task and interaction with the Idea Garden.

5.6 Summary

In this chapter, 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 Chapter 3 and Chapter 4, and by theories from the literature, such as those on problem solving and Minimalist Learning Theory. The Idea Garden is different from prior 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 domain-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.

Chapter 6. Two Qualitative Empirical Studies

To investigate the effectiveness of the Idea Garden approach and to guide refinements of our design of the Idea Garden prototype, we conducted two qualitative empirical studies. In Study 1, we sought to understand when and how the Idea Garden (the prototype presented in Chapter 5) would help (or not help) end-user programmers to overcome programming barriers. In Study 2, we improved the Idea Garden prototype based on the feedback from Study 1. The goal was to learn whether users would learn relevant problem-solving strategies and programming knowledge in addition to whether the Idea Garden would help users overcome barriers and accomplish their task.

6.1 Study 1: Overcoming Barriers – When and How?

In Chapter 5, we presented a prototype of the Idea Garden that aimed to address three barriers (*How-to-Start*, *Composition*, and *More-than-Once*) with four features (*Start-with-a-column-name*, *Finder-page*, *Second-webpage*, and *Generalize-with-repeat*).The goal of our first evaluation of the prototype was to answer the following research question:

> *RQ: When and how will the Idea Garden help—or not help—end-user programmers overcome their barriers?*

6.1.1 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.

6.1.2 Procedure

The study procedure consisted of a tutorial, a scripting task, and a 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 6.1 shows the paper prototype, and Section 5.3.1 has screenshots of the executable prototype.

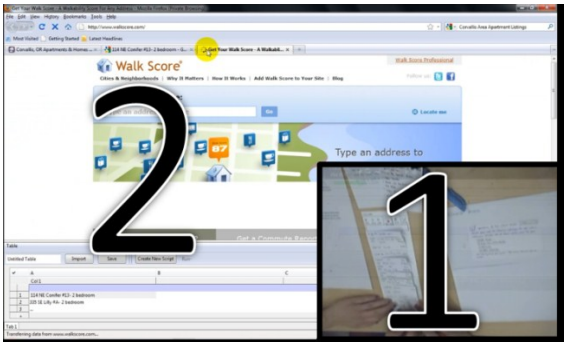


Figure 6.1 Example paper-prototype session with (1) a paper script and suggestions, and (2) a CoScripter table and webpage.

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 five minutes 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"), but did not count these

situations as the participant making progress. This allowed us to gather data from subsequent parts of the task.

At the end of the task, our 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 Chapter 4, we identified episodes where a participant encountered a barrier as indicated by, e.g., the participant turning to the Idea Garden for help or simply verbalizing a need for help, but focused our analysis on the episodes where the participants encountered the barriers that our Idea Garden features targeted.

6.2 Study 1 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 to 4 of the prototype’s suggestions, for a total of 22 interactions (Table 6.1).

Table 6.1 Participants' feature interactions and results.
+: participant followed suggestion and made progress.
-: participant did not follow suggestion, or did not make progress from following it

Suggestion:	*Start-with-a-column-name*	*Finder-page*	*Second-webpage*	*Generalize-with-repeat*	**Steps completed**
Barriers:	How-to-Start		Composition	More-than-Once	
Step #:	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

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 6.1 shows, 8 participants benefited from at least one interaction with a suggestion.

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 needed the `repeat` command. As Table 6.1 shows, one participant completed all three steps, six completed the first two steps, and two completed only one step.

6.2.1 When and how the suggestions helped

6.2.1.1 Helping with the How-to-Start Barrier

The *How-to-Start* barrier that we previously identified in Chapter 4 proved to be enough of a struggle that six participants turned to the Gardener. 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.

6.2.1.2 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 *Second-webpage* overcame this barrier.

All three participants (Paper-F3, Exe-F3, and Exe-M1) followed the following sequence. They first tried to accomplish Step 2 through the web page they used for Step 1, hoping to directly specify 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 Gardener for a suggestion—and subsequently acted on the Gardener's idea of using web sites 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 flowing 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."*

6.2.1.3 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 (flawed) script. 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 to be presented in Section 6.5 through Section 6.7, we directly measure 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.

6.2.2 … 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 *Second-webpage* 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 *Second-webpage* 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.

6.3 Study 1 Discussion

The results of the study suggested several possibilities for the Idea Garden's future.

6.3.1 Understanding Relevance

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.

In contrast the success of the *Generalize-with-repeat* suggestion, the *Second-webpage* suggestion failed to entice several participants into useful actions. For example, Exe-F1's remark as she dismissed the *Second-webpage* suggestion implies that she had a different goal from what the suggestion's content had conveyed to her: she wanted to "write the script" whereas the suggestion appeared to be asking her to work with data she already had in her table in a way that was irrelevant to her task, e.g., finding zip codes based on the addresses in her table.

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

In this case, the mismatch between the participant's goal and what the participant perceived as the suggestion's goal may have prevented the participant from benefiting from the feature although the knowledge contained in the feature, e.g., the dataflow concept, was exactly what the participant needed to know in order to make progress.

Again, Attention Investment's notion of perceived benefits can be used to explain Exe-F1's behavior. She saw the suggestion as irrelevant and of low benefit to pursue. Thus, she turned away from the suggestion.

We will refer to this problem as Problem #1:

Problem 1: *Irrelevance*

6.3.2 Balancing "Correct" and (Deliberately) "Flawed" 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 flawed suggestions.

By design, most of the Gardener's suggestions have "correct" and "flawed" 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 Figure 5.7). The flawed 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 of restaurants.com (Figure 5.4) cannot be used to find apartments.

Unfortunately, the flawed part of a suggestion sometimes dominated participants' perceptions of the suggestion's relevance. For example, Exe-F2 said that the *Second-webpage* suggestion (Figure 5.5) 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 the suggestion (Figure 5.4).

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, Profetti, & Britain, 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.

We will refer to this problem as Problem #2:

> ***Problem #2:*** *Balancing "correct" and "flawed" aspects of suggestions.*

6.4 Improving the Idea Garden Features (Round 1)

Based on these findings, we improved the Idea Garden in the ways we described next. Each improvement aims to address Problem #1 (Section 6.3.1), Problem #2 (Section 6.3.2) or both.

Context-insensitive Suggestions (Problem #1): Previously, all features were context-sensitive; they became available only at a time determined to be opportune by the Idea Garden. The contents of the context-sensitive features reflected user's code and/or data. For example, the Second-webpage feature's content is determined by the type of data stored in users' table. If a table column holds "addresses", the corresponding Second-webpage will show examples of

using addresses to compute addition information, such as using two addresses to calculate distance.

We added a context-insensitive version of the Second-webpage feature, namely Second-webpage that is accessible through a "Help" button located at the top of the browser tab and is always available (Figure 6.2). A reason for introducing this feature is to alleviate the problem that some users found the context-sensitive Second-webpage's suggestion, to be irrelevant due to its flawed examples. Provided that users recognize that a context-insensitive feature is unrelated their current contexts, users are more likely to follow up on the feature even upon seeing an example that does not seem to solve their immediate problems. Another reason for having a context-insensitive feature is to provide help that is always to the user available even when the context for viewing a context-sensitive feature is not available.

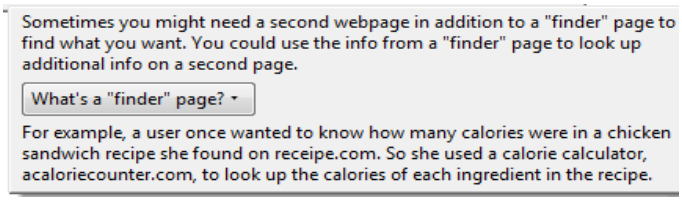


Figure 6.2 The context-free Second-webpage feature

Lead-in Sentence Infers User's Goal (Problem #1): Previously, the lead-in sentence, that is, the first sentence, of a context-sensitive feature, made a guess about the user's context, e.g., Does this column contain "address"? (Second-webpage for the "address" data type). A potential problem is that the Idea Garden's guess about user's current context can be incorrect, e.g., it might say "name" instead of "address", leading the user to abandon the suggestion immediately.

To help solve this problem, instead of fixing the Idea Garden's guess about user's context (e.g., instead of replacing "name" with "address"), we changed the lead-in sentence of the context-sensitive features (Second-webpage and Generalize-with-Repeat) to hint at the user's goal based on what he/she has just done (Figure 6.3's and Figure 6.4's "after" part). For example, for Second-webpage, the lead-in sentence infers user's goal as "… find more info based on what you have in this column ("address")?". Although the Idea Garden's guess about

whether the table is storing “name” or “address” may be incorrect, users may still pursue the feature as long as the feature matches users’ goal which is to find the missing information. Therefore, by aligning the suggestion’s lead-in sentence with users’ goal, we hope to increase users’ likelihood of reading the whole suggestion for possible actions to carry out for their goal.

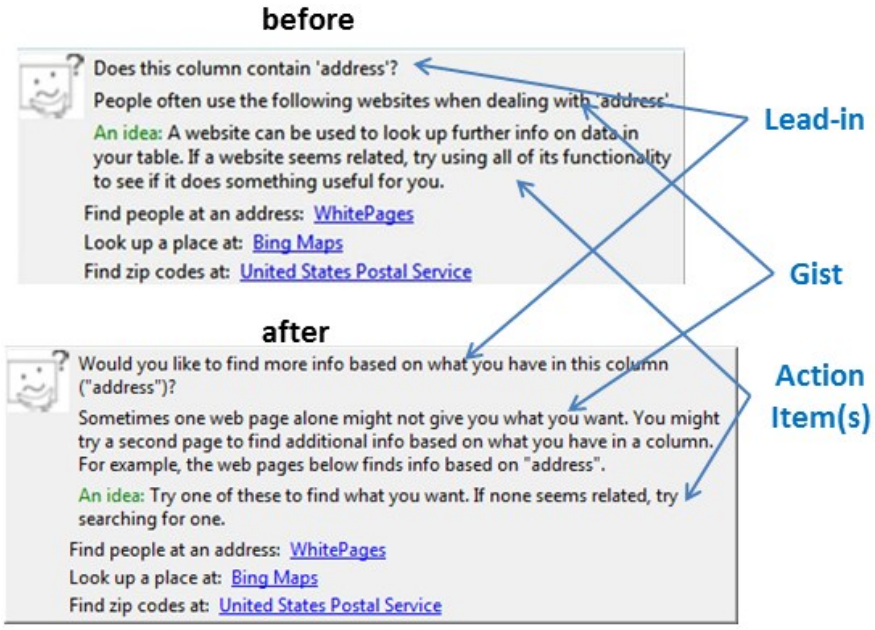


Figure 6.3 Context-sensitive Second-webpage: before and after

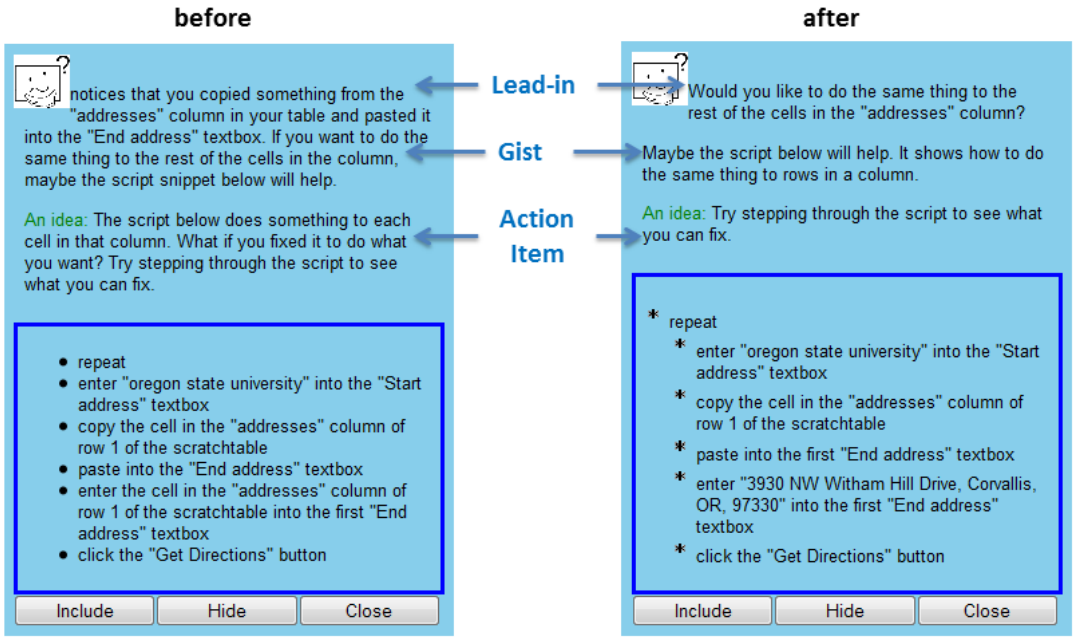


Figure 6.4 Generalize-with-repeat: before and after

Clarified Gist to Explain Solution (Problem #2): Previously, the gist portion aimed to explain the essence of the features but we felt that it was often vague. For example, in context-sensitive Second-webpage, the gist was "People often use the following websites when dealing with 'addresses'" which did not explain why the user should take a look at the websites. We improved the clarity of the gist of the features (context-sensitive Second-webpage (Figure 6.3's "after" part) and Generalize-with-Repeat (Figure 6.4's "after" part)) to explain a solution for user's goal and to allude to the example(s) to come. For context-sensitive Second-webpage, the gist became "Sometimes one webpage alone might not give you what you want. You might try a second webpage to find additional info based on what you have in a column. For example, the webpage below finds info based on 'address'.".

Providing Additional Action Items (Problem #1, #2): Previously, the context-sensitive Second-webpage feature had only one action item, namely to try an example webpage if it seemed related. A problem we found was that some users immediately turn away because they found the examples to be unrelated to their task. Thus, we added an "alternative" action, namely, "If none seems related, try searching for one." (Figure 6.3's "after" part) This change was intended to make clear that the example websites are not meant to be used "as-is".

Improved Analogy Wording (Problem #2): The fact that several participants from Study 2 failed to benefit from the examples in the context-sensitive Second-webpage feature may be indicating a problem with applying the analogy strategy. In particular, instead of making an analogy to relate the example websites to their task, some participants made no analogies and thus deemed the websites irrelevant to their task.

To help users make analogies, we used the structure-mapping theory (Gentner, 1983) to guide the wording of the gist that has a role of explaining the mapping. According to the structure-mapping theory, to make an analogy, a problem-solver needs to map a group of objects in Domain A to a group of objects in Domain B. The crux of the mapping is that when going from Domain A to Domain B, the objects change but the relationships between the objects are preserved. For example, the solar system is analogous to an atom because although they have different components, e.g., the sun vs. the nucleus or planets vs. electrons (objects), both are central force systems (relationship between the objects).

We applied the structure-mapping theory to the context-sensitive Second-webpage suggestion in the following way. We identified two domains: domain of the suggestion and the domain of the user. In the suggestion domain, the objects are input info (e.g., address), web page (e.g., Bing Maps), and output info (e.g., distance). The relationship is an input-webpage-output relationship. In order to make analogy based on the suggestion, the user needs to see beyond the concrete objects (i.e., address, Bing Maps, and distance) to realize the relationship among the objects (i.e., the input-webpage-output relationship). To help the user achieve that, we made explicit the relationship for the user in the gist part of the suggestion (i.e., "You might try a second webpage [webpage] to find additional info [output] based on what you have in a column [input]."). We also made clear that the examples from the suggestion's domain are "examples" and thus, are not meant to be used without changes (i.e., "For example, the webpage below finds info based on 'address'.").

Webpage-as-Component Pattern: In addition to the improvements mentioned so far which target Problem #1 and/or Problem #2, we incorporated the Webpage-as-Component pattern (the idea of using certain webpages, such as Google Maps, to calculate values from a given input) into the context-sensitive Second-webpage and context-free Second-webpage feature. This change is reflected by the bold text in Table 6.2's "Design patterns…" section.

Table 6.2 The improved Idea Garden prototype for CoScripter – Round 1. Shaded cells: new additions. Crossed-out: participants from Study 2 (to be presented in Section 6.5 through Section 6.7) did not encounter the barriers and/or did not use the features.

		Suggestions				
		~~*Start-with-a-column-name (Figure 5.3)*~~	~~*Finder-page (Figure 5.4)*~~	*Second-webpage (context-free, Figure 6.2)*	*Second Webpage (context-sensitive, Figure 6.3's after)*	*Generalize-with-repeat (Figure 6.4's after)*
Barriers targeted:	~~*How-to-Start*~~	~~X~~	~~X~~			
	Composition			X	X	
	More-than-Once					X

Programming concepts reflected in suggestions

~~Data Extraction~~	~~The select a slice of structured data from a web page and putting it into the table~~		~~X~~			
Dataflow	Flow data between web page and table, or between web pages			X	X	
Iteration	Loop through rows of table, operating on each row					X

Design patterns reflected in suggestions

~~Finder~~	~~Use a web page to find information (as opposed to computing information)~~		~~X~~			
Webpage-as-Component	Use a web page to compute information (as opposed to finding information)			X	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 the end goal, then figure out the last step to the goal, second to the last step, and so on until the givens are reached.~~	~~X~~	~~X~~			
Analogy	Find solution to similar problem, then adapt it for problem at hand		~~X~~	X	X	
Generalization	Solve one instance of a problem and generalize the solution to all instances in the problem					X

6.5 Study 2: Overcoming Barriers and Learning

To evaluate how effective our new features were, we conducted a qualitative think-aloud study whose goal was to investigate whether the Idea Garden helps end users to overcome barriers in their programming tasks, and if so, whether any learning happens along the way. Therefore, our research questions were:

> *RQ1: Does the Idea Garden help users complete programming tasks?*
>
> *RQ2: Does the Idea Garden help users overcome barriers?*
>
> *RQ3: Do users who use the Idea Garden when they encounter barriers then learn relevant programming concepts, patterns, and/or problem-solving strategies?*

6.5.1 Participants

We used emails and flyers to recruit 15 university students and recent graduates with majors other than electrical engineering or computer science. None had ever programmed before. Of the 15, 10 used the Idea Garden's features (6 females, 4 males; gender is indicated as F or M in IDs below).

6.5.2 Procedure

We gave each participant a background questionnaire that included a modified version of the standard computer self-efficacy test (Compeau & Higgins, 1995). Although one of our research questions focuses on learning, we chose not to use a pre-test of programming knowledge because doing so could have biased behavior—for example, causing users to focus on features that seemed related to the pre-test. Instead, we used temporal evidence (described later) to detect learning.

Because the Idea Garden is intended for users who have used an environment before (but then get stuck), we gave a 25-minute hands-on tutorial then walked participants through how to create three scripts: one to look up information from a webpage, one to pull data from a webpage into a table, and one to push data from the table to a webpage. We taught some concepts and patterns, but no strategies. Specifically, we taught the Dataflow and Iteration concepts, and had participants do all of the Repeat-Copy-Paste pattern and part of the

Webpage-as-Component pattern. We did not teach strategies per se, although it may have been possible for some participants to infer strategies from their tutorial work. The tutorial's goal was to give participants a basic level of CoScripter ability and practice, but we did not include the Idea Garden in the tutorial. Our intent was that participants should already know enough about CoScripter to start a task, but to use the Idea Garden (if they so chose) without having used it before.

Next, participants began their main task: to create a script for finding 2-bedroom apartments under $1300 within a 10-minute drive of the Ohio State University campus. The task had three implicit subtasks: (1) import a list of apartments and their addresses from a webpage into the table, (2) iterate over the addresses to compute driving time to Ohio State University, and (3) copy each driving time back to the table. Participants were vulnerable to the Composition barrier during subtasks 1 and 2 and to the More-than-Once barrier during subtasks 2 and 3. The Idea Garden was active during the task, and users could refer back to the tutorial.

If a participant became so stuck on a barrier that no further progress seemed possible, the researcher pointed out Idea Garden features or, if that did not help, eventually suggested an action to overcome the barrier. These hints enabled the participant to make progress, so that we could gather data on later subtasks. To support analysis, participants were instructed to talk aloud as they worked. We recorded audio, screen captures, and video of participants.

Once participants finished the task or exceeded 55 minutes, we used a structured interview to assess knowledge of each programming concept, pattern, and problem-solving strategy in the Idea Garden's features (listed in Table 6.2 from Section 6.4). Bloom's taxonomy identifies six levels of knowledge: remembering, understanding, applying, analyzing, evaluating, and creating knowledge (Anderson, et al., 2000). Because the task was short, our interview targeted the three lowest levels of this taxonomy. When structuring questions, we focused on learning transferability to another task or context (Bransford, Brown, & Cocking, 2000); specifically, we presented participants with the hypothetical task of creating a script to look up the best price for a novel sold online, then asked how to perform subtasks in that context. To further evaluate understanding, we used multiple choice and fill-in-the-blank questions, with follow-up questions asking participants to interpret various portions of scripts and to predict script behaviors.

6.5.3 Analysis Method

Two researchers independently used qualitative thematic coding to analyze the videos. They coded (1) barriers encountered based on whether participants struggled in specified ways and (2) progress made including barriers overcome (Table 6.3).

They also graded interviews' multiple choice and fill-in-the blank answers using an answer key. Finally, participant interviews' task answers were categorized using the concept, pattern, and strategy names in Table 6.2. To ensure reliability, we used a standard inter-rater agreement exercise (Seaman, 1999) in which, first, the independent analyses of the two coders achieved 80% agreement (Jaccard similarity) on 30% of the data, and then one of the coders alone completed the analysis.

Table 6.3 Coding scheme for whether (1) a participant encountered a barrier, and (2) Participant made progress.

1. Barrier	**Action/Vocalization**
Composition Barrier	Not knowing to combine *multiple* web pages
	Not knowing that a page *can calculate* driving time
	Not knowing *how* to use a page to calculate
	Not using the table as intermediate storage for webpage data that must be sent to a second webpage
More-than-Once Barrier	Not realizing operations on table rows could be generalized using the `repeat` command
	Uncertainty about how to use or the misuse of the `repeat` command, e.g. wrong placement or syntax
	Misunderstanding of the scope of the `repeat` command, e.g., thinking it could work on elements of a webpage
	Revisiting tutorial script for help with the `repeat` command
2. Progress	**Observed Behavior**
No movement	Behavior remained unchanged
Movement	Behavior changed but did not move closer to completing task
Positive Movement	Behavior changed and moved closer to completing task
Barrier Overcame	Behavior changed to overcame the barrier

6.6 Study 2 Results

6.6.1 RQ1: Idea Garden's Help with the Task

Almost all participants who turned to the Idea Garden for help with a barrier were able to complete at least part of the task. For the 10 participants who turned to the Idea Garden when they encountered a barrier, four went on to complete all three subtasks, and four more completed at least one subtask. Table 6.4 shows the details.

Table 6.4 Participant's progress in the task using Idea Garden.

● Barrier, then participant used Idea Garden to overcome barrier.
● Barrier, then participant used Idea Garden to change approach (but did not overcome barrier).
○ Barrier, then participant used Idea Garden, but did not act on it.
✔ Participant completed subtask.
✘: Participant started but did not finish the subtask.
R : Researcher solved the subtask.
✔R: Researcher completed part, participant completed the rest.
✘R: Participant started but did not finish the subtask and relied on researcher to solve the part of the subtask he started.

		F1	F2	F3	M1	M2	F4	M3	F5	M4	F6
Sub-task 1	Used Idea Garden	●		●	○	○	●	●			
	Task Success	✔	✔	R	R	R	✔	✔	✔	✔	✔
Sub-task 2	Used Idea Garden	●	●	●	○	●		○	●	●	●
	Task Success	✔	✔	✔R	✘R	✔	✔	✔	✔	✔	✔
Sub-task 3	Used Idea Garden								●	●	●
	Task Success		✘	✘			✔	✘	✔	✔	✔
Task Success Total		2/2	2/3	0.5/3	0/2	1/2	3/3	2/3	3/3	3/3	3/3

6.6.2 RQ2: Idea Garden's Help with the Barriers

As shown in Table 6.2 in Section 6.4, the context-sensitive Second-webpage and the context-free Second-webpage features target the Composition barrier, whereas the Generalize-with-

repeat feature targets the More-than- Once barrier. We will consider these features in the context of the barriers they target.

6.6.2.1 Overcoming the Composition Barrier

Table 6.5 summarizes, for the six participants who encountered the Composition barrier and then interacted with the Idea Garden, their feature usage and how successful they were in overcoming the barrier. As the table shows, half the participants who encountered the barrier were able to overcome it.

Table 6.5 The six participants who encountered the Composition barrier and their feature usage.

● participant used the feature and overcame a barrier.
● participant observed the feature, made movement.
○ participant observed the feature, but made no movement.

Feature	Participants who encountered Composition barrier					
	F1	F3	M1	M2	F4	M3
Context-sensitive Second Webpage	●	○○		○	●	
Context-free Second-Webpage		○●	○	○○		●

With the split in participants' success in overcoming g the Composition barrier, we analyzed whether these participants exhibited behavior consistent with an understanding of the Webpage-as-Component pattern, Analogy strategy, and Dataflow concept that the context-sensitive Second-webpage and context-free Second-webpage aimed to convey. Below, we present episodes that shed light on whether participants picked up this knowledge or not, and why.

Applying the Analogy strategy. F4's interactions with the context-sensitive Second-webpage feature show how she successfully used the Analogy strategy to figure out how to compute driving time. Figure 6.5 illustrates her interactions with CoScripter and the Idea Garden. The Analogy strategy rests on the ability to effectively map concrete examples to the task at hand. At first F4 tried to find a web page that showed the needed data (driving times), rather than computing the values with a calculator web page (Google Maps). She could not find a suitable site, but added a "distance" column header to her table anyway. This new column head

included an indicator for the context-sensitive Second-webpage feature. The tooltip suggested websites that accepted the data in her column (distance) as input to calculate new information. These websites included a gas calculator and a running time calculator, neither of which was what F4 needed. After inspecting each site, she searched for an analogous website that was appropriate for her task, a "driving distance calculator." Her search led her to a page that she used to finish Subtask 1.

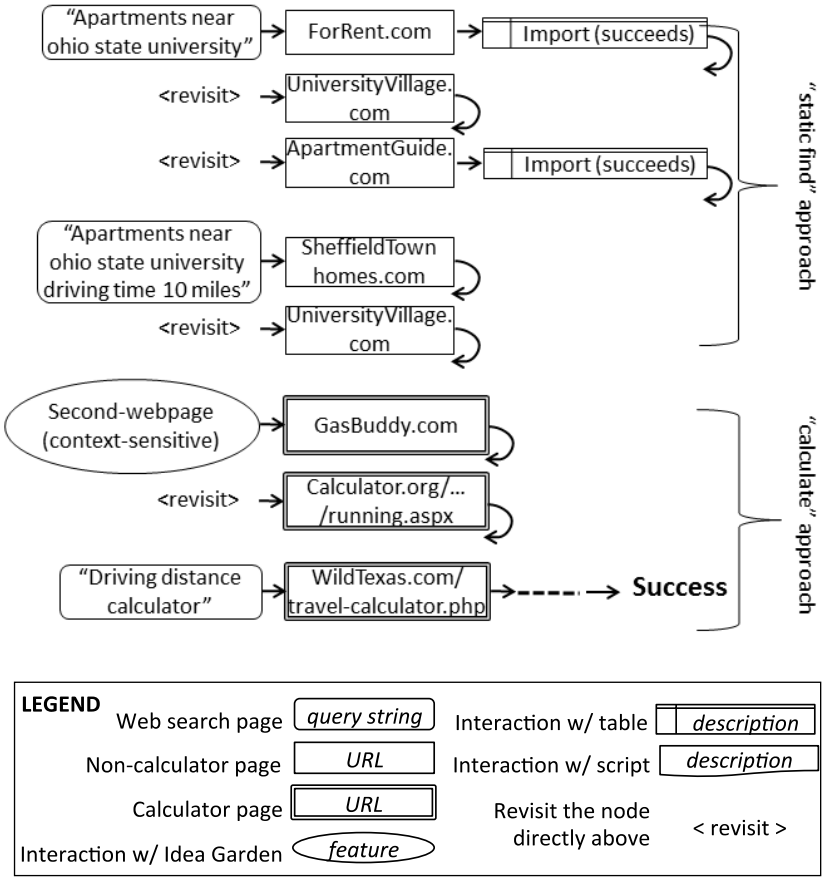


Figure 6.5 An episode from F4's sequence of interactions with CoScripter and the Idea Garden. (Curved arrows indicate transition to the next line).

In contrast to F4, neither F3 nor M2 sought websites analogous to the ones provided by the context-sensitive Second-webpage feature. Both F3 and M2 dismissed the context-sensitive Second-webpage and went back to their prior approach of looking for driving time from an apartment-listing page. Similarly, M1 interacted with the context-free Second-webpage feature, which showed an example of using a calculator webpage to compute calories of a recipe;

however, he appeared not to even read the example, and thus, he lost the opportunity to apply an analogy.

Applying the Webpage-as-Component pattern and the Dataflow concept. Three participants used the Idea Garden's suggestion directly to apply the Webpage-as-Component pattern and the Dataflow concept. In F4's episode (above), she applied the Webpage-as-Component pattern, using a travel calculator page as a component. In leveraging the page, she also applied the Dataflow concept, pushing addresses into the component and pulling driving times out. Similarly, F1 used Bing Maps, which was suggested by the context-sensitive Second-webpage feature, to compute driving times, thus applying the Webpage-as-Component pattern and Dataflow concept. For M3, the context-free Second-webpage feature brought to his attention that he could send existing data to a calculator page to get driving time, thus he too applied the Webpage-as-Component pattern and Dataflow concept.

In contrast, F3, M1, and M2 exhibited no evidence of understanding or applying the Webpage-as-Component Pattern and the Dataflow concept. Even after the researcher directed these participants to an appropriate component page, Google Maps, they continued to struggle. For example, M2 used Google Maps to find apartments rather than to compute driving times. The researcher provided each participant with one more hint: use "Get Directions". But the hint led to more confusion. For example, M1 said: "I'm just looking for the location of one place, not directions from me to it".

6.6.2.2 Overcoming the More-than-Once Barrier

The More-than-Once barrier is the target of the Generalize-with-repeat feature—and 7 of the 9 participants who turned to this feature overcame the barrier and completed the second subtask (Table 6.6). Because of this success, we consider how the feature might have influenced different participants' behaviors.

Table 6.6 The nine participants who encountered the More-than-Once barrier. Dot notation is as in Table 6.5.

Participants who used the Generalize-with-repeat feature								
F1	F2	F3	M1	M2	M3	F5	M4	F6
●	●	●	○	●	○	●	●	●
Content Timing	Content Timing Self-Eff	Timing	Content	Content Timing Self-Eff	Self-Eff	Content Timing Self-Eff	Content Timing	Content Timing

Explanation content: Recall from Figure 6.4, that the explanation briefly gave a general idea and then presented an incomplete script snippet that participants could include in their scripts and then modify. Our intent was that users would include it, notice the missing piece, i.e., the code to pull drive times to the table, then fill it in. And for F5, M4, F6, the explanation led them to do exactly that. (F2 also noticed the missing piece but ran out of time to act upon it.) Figure 6.6 illustrates the way this played out for M4. The explanation helped F1 in an-other way: she remembered the repeat command but had forgotten how to do it. She then used the explanation to remember:

"I forgot how to repeat ... [Opens the suggestion] Ah, repeat... [Reads suggestion line by line and begins to edit a Repeat loop into her script]."

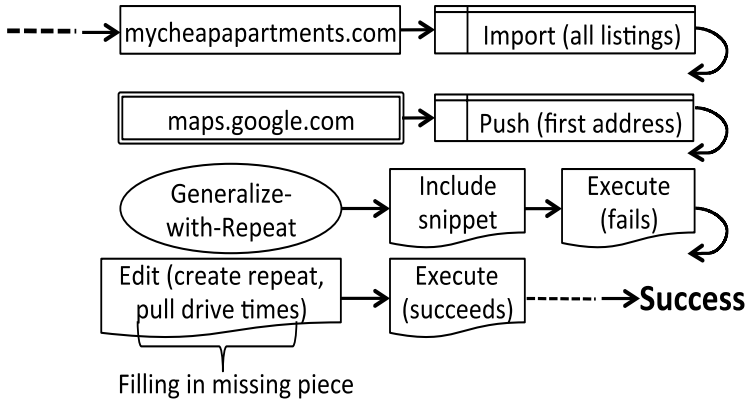


Figure 6.6 An episode from M4's session (Legend in Figure 6.5).

Two other cases of content were different, in that they were related to recognizing the relevance of repetition rather than how to implement repetition. In M2's case, he had not realized that his actions were generalizable at all. Instead, he processed the first three rows of his table one at a time. When he noticed the explanation, he used the script snippet it suggested, and changed to a loop rather than his previous one-at-a-time approach:

> *"[reads: Would you like to do the same thing to the rest of the cells?] Oh okay [reads the suggestion's script, then clicks "Include" to insert it into his script.]"*

Finally, M1's case showed a need for explanation content that was not present. There is a "role expressiveness" issue (Green & Petre, 1996) in the way a single entry of the `repeat` body appears in CoScripter scripts: the notation refers to the first row only of the table, e.g., `copy the cell in the "addresses" column of` **`row 1`** `of the scratchtable` (see the script in Figure 6.4's "After" part). This notation convinced M1 that repeat was not relevant, so he did not follow the suggestion. If the explanation content had clarified this issue, perhaps he would have.

Timing and context: A strength of the Generalize-with-repeat feature seems to be that it appears in the right context: when the user is actively working on a portion of the script that involves a row of the table, but is not using a repeat in that portion of the script. Further, the "Include" button automatically places the snippet into the user's script, reducing the cost of integrating the snippet, and encouraging users to start experimenting with it. This contrasts with a tutorial, which appears in a separate context and provides no direct encouragement or explicit support for integration with the task at hand. In fact, all seven participants who encountered the More-than-Once barrier and managed to accomplish the second subtask, did so by using the feature as a reference to implement their own repeat loop or by including the script snippet it provided.

The experience of M4 illustrates the value of integrating suggestions into the context of the task at hand. When he encountered the More-than-Once barrier, M4 referred to a tutorial example that illustrated the use of the repeat command. In response, however, he made no changes to his script. Instead, he subsequently tried copying an address from the table into Google Maps. This triggered the Generalize-with-repeat feature.

In contrast to his non-use of the tutorial, he incorporated the suggested snippet, then stepped through it in the context of his own program. After finding where further work was needed, he was able to create a new repeat block that effectively completed the second subtask without any further assistance.

This timing and context seemed very effective, and may be the primary reason so many of the participants acted upon the suggestion in a way that turned out well for their scripts.

Self-efficacy: Self-efficacy is a specific form of self-confidence: a person's prediction of how well he or she can perform a specific task (Bandura, 1977). Results from empirical studies across numerous populations (e.g., (Burnett, et al., 2010)) have shown that users with low self-efficacy tend to shy away from unfamiliar features designed to help them. Table 6.6 identifies the two males and two females whose self-efficacy scores from the background questionnaire were below the medians for their gender (marked as "Self-Eff"). The Idea Garden's suggestion appeared to interact with these low self-efficacy participants in both negative and positive ways. For example, M3, whose self-efficacy score was among the lowest in the male participants, when the feature came up, acknowledged the suggestion as "a direction to repeat" but did not follow through with it, perhaps due to his low self-efficacy: "This is hard. I don't know what I'm doing". On the other hand, the participant with the lowest self-efficacy score, F5, was able to make use of the suggestion's script snippet and eventually edit the snippet to succeed beyond the second subtask accomplishing the third subtask as well. Thus, in M3's case, the suggestion did not allay his concerns about his abilities, and may even have exacerbated them, whereas in F5's case, the suggestions may have reassured her that she was moving in the right direction.

6.6.3 RQ3: From Barriers to Learning

To assess possible learning, we considered evidence of learning to be the temporal sequence <barrier, Idea Garden use, correct answer>. That is, a participant first had to show a lack of knowledge by running up against a barrier, then had to turn to the Idea Garden for help, and finally had to answer post-test questions correctly. Our rationale was that those who ran into barriers were, by definition, demonstrating a knowledge gap in at least one of the concepts, patterns, and/or strategies.

Table 6.7 and Table 6.8 summarize the results for the knowledge items relevant to the Composition barrier and the More-than-Once barrier, respectively. The strategy questions included both broad, open-ended questions that tested whether the participant would think to apply the strategy in planning how to proceed with a new task, and more specific questions that tested whether the participant could apply the strategy either in the use of specific Idea Garden features or when facing a specific situation within the new task. The feature-specific questions for analogy were presented to only the participants who saw the corresponding feature during the task, and thus, those questions are treated separately in the tables.

Table 6.7 Interview results for questions related to the Composition barrier.

✔: answered a question correctly.
-: showed no evidence of the knowledge item.
✘: incorrect use of the knowledge item.
?: did not answer.
NA: participant was not asked that question.
Shaded: received help from researcher relating to the knowledge item.

Knowledge Item		Overcame barrier on their own			Did not overcome barrier on their own			Demonstrated Knowledge
		F1	F4	M3	F3	M1	M2	
Dataflow Concept (problem required it)		✔✔✔✔✔	✔✔✔✔✔	✔✔✔✔✔	✔✔✔✔✔	✔✔✔✔?	✔✔✔✔✔	6/6
Webpage-as-Component Pattern (problem required it)		✔	✔	✔	✔	✔	✔	6/6
Analogy Strategy	(context: broad, open-ended task planning)	-	✔	✔	-	-	-	2/6
	(context: using context-sensitive Second-webpage)	✔✔	-✘	NA	✔✘	NA	-✘	2/4
	(context: using context-free Second-webpage)	NA	NA	✔✔✘	✔✔?	✔✔✘	✔✔✔	4/4

Table 6.8 Interview results for questions related to the More-than-Once barrier (Symbols from Table 6.7).

Knowledge Item		Overcame barrier on their own							Did not overcome barrier on their own		Demonstrated Knowledge
		F1	F2	F3	M2	F5	M4	F6	M1	M3	
Iteration Concept (prediction/comprehension questions)		✔✔✔ ✔	✔✔✔ ✔	✘?✔ ✔	✔✔✔ ✔	✔✔✔ ✔	✔✔✘ ✔	✔✔✘ ✔	✔✔✔ ✔	✔✘✔ ✔	9/9
Repeat-Copy-Paste Pattern (problem required it)		✔	✔	✔[1]	✔	✔	✔	✔	✘	✘	7/9
Generalization Strategy	(context: broad, open-ended task planning)	✔-	- -	- ✔	- -	- -	- -	- -	- -	- -	2/9
	(context: specific situation within a task)	✔	✔	✔	✔	✔	✔	✔	✘	✘	7/9

[1] Researcher helped with first copy/paste, then participant invoked Idea Garden to overcome the rest.

Overall, every participant who encountered a barrier and then used the Idea Garden demonstrated understanding of all the relevant programming concepts (Dataflow and Iteration) and one of the patterns (Webpage-as-Component) during the interview; however, only those who subsequently overcame barriers on their own (i.e., without hints from the researcher) demonstrated understanding of the Repeat-Copy-Paste pattern, the ability to solve the more open-ended Analogy and Generalization problems, and the ability to apply the Generalization

6.7 Study 2 Discussion and Summary

Our results showed that 9 of the 10 participants who encountered barriers and used the Idea Garden, subsequently overcame at least one of their barriers on their own. Moreover, in the post-test, those 9 demonstrated understanding of all the programming concepts, patterns, and strategies relevant to the barriers they overcame on their own. Their difficulties with the barriers prior to using the Idea Garden, combined with their subsequent success after using the Idea Garden, and their demonstrated understanding during the post-test, triangulate to suggest they indeed learned from their experience.

Minimalist Learning Theory (Carroll, 1998), which inspired our design of the Idea Garden (5.4), provides a basis for understanding its features' effectiveness. This theory argues that

active users will be more likely to use and profit from learning-related resources that are situated within their task, rather than in some other program or resource, such as a tool or tutorial. Recall that few participants referred back to the tutorial examples; we hypothesize that this is because the examples were external to the task at hand.

Another way to understand these results is in terms of Attention Investment. Recall from 6.3.1 that Attention Investment is a model of cost, benefit, and risk which posits that users will be more inclined to learn new abstractions if they perceive that doing so requires low up-front costs and provides substantial benefits (Blackwell, 2002). CoScripter has numerous such abstractions, including tables and repetition that are needed for mashup tasks. In the case of the Idea Garden, the cost to venture forward starts with reading a tool tip—a cost that most users perceive to be low. The Idea Garden's suggestions seem to be concrete enough to keep users' further estimates of cost aligned with actual costs. The potential benefit also seems to be clearer to a participant: that overcoming the barrier will allow the user to move ahead with their task. In contrast, external tutorials or examples require switching to other documents and sifting through material to find the relevant parts, with a potential risk that relevance to the task at hand will still not be clear.

Recall also that, by design, the Idea Garden provides relevant but "flawed" or incomplete suggestions which led to Problem #1 and Problem #2 in our previous study (discussed in 6.3). Our hope was that this incompleteness would engage users intellectually and encourage them to apply strategies such as analogy that would stay with them and many of the improvements we described in 6.4 aimed at this goal. The current study's participants' success with overcoming barriers, combined with their correct answers to the analogy questions in our post-test, suggest that this approach showed merit by steering some participants away from Problem #1 and #2. Nonetheless, a few participants struggled with analogy, which prevented them from overcoming some barriers (Problem #1 and #2 still existed for some). This limitation suggests that analogies need to be more easily recognizable and applicable to the task at hand.

Although the Idea Garden's context-sensitive tooltips were designed to make suggestions more helpful to users, over-contextualizing assistance to the task at hand runs the risk of limiting transferability of learning to other situations. For many participants, it appears that this generally was not a problem with the Idea Garden, as these participants were able to

successfully answer post-test questions that asked them to describe how they would perform another programming task. It is an open question regarding the extent to which this apparent learning will lead to quantitative improvements in users' ability to correctly complete other programming tasks.

Overall, our results suggest that, by being available to support learning, the Idea Garden encouraged users actively trying to accomplish a programming task to learn along the way. While tools that automate away some barriers may help in the short term, this help may come at the expense of skills users will need to handle similar difficulties that may arise later. Our study's results suggest that the Idea Garden approach may help users to overcome their barriers now and in their future.

6.8 Improving the Idea Garden Features (Round 2)

Study 2's results showed that analogy remained hard for many participants. In this section, we describe how we attempted to address this issue (in *Choice of strategies* below) and improve various other aspects of the Idea Garden features.

A context-free version of the Getting-started feature and the Generalize-with-repeat feature: In the last iteration (presented in 6.4), we introduced a context-free version of the Second-webpage feature, i.e., the context-free Second-webpage feature. This feature demonstrated its benefits by giving a new idea to half of the participants who saw it. To allow the rest of the features to benefit from a context-free version, we added a context-free *Getting-started* feature (Figure 6.7) and a context-free *Generalize-with-repeat* feature (Figure 6.8). An additional benefit to the context-free version is that they can be always available to users to discover or revisit, not just at context-appropriate moments.

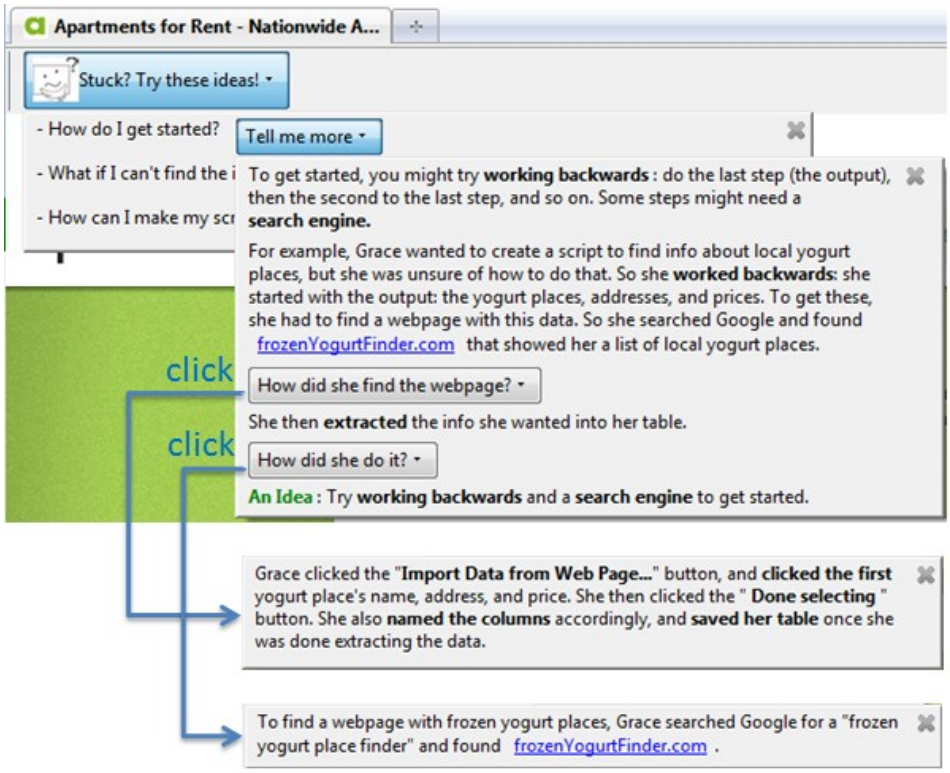


Figure 6.7 The context-free Getting-started feature. (Note: The context-sensitive Getting-started feature has the same content as the context-free Getting-started feature. The difference between a context-sensitive version and a context-free version of a feature is that the former is contextualized using users' code and data whereas the latter is not. Because the context-sensitive Getting-started feature becomes available at a time when there is no user data to contextualize the feature with, i.e., once CoScripter starts, it has the same content as the context-free version.)

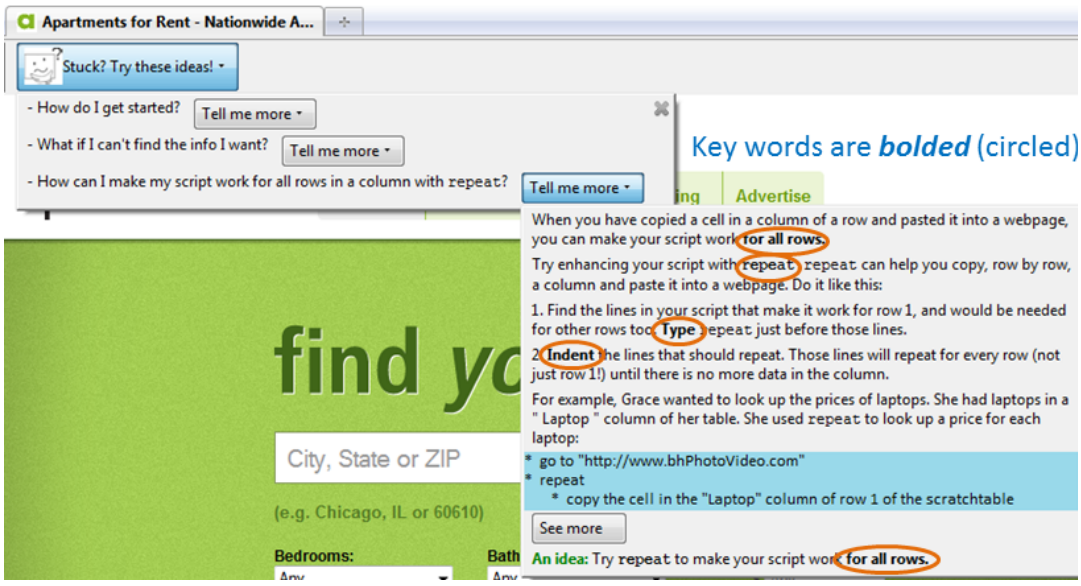


Figure 6.8 The context-free Generalize-with-repeat feature. Key words are bolded (We have also circled them for the purpose of this figure).

Bolded key words: To enhance the readability of the features' contents, we used bold text for key words. For example, Figure 6.8 shows a screenshot of the context-free Generalize-with-repeat feature in which some key words, e.g., `repeat`, were bolded.

Explicit strategy steps: We made the steps for carrying out the strategies explicit. For example, the generalization strategy used in the Generalize-with-repeat feature (both context-free and context-sensitive) now included the step of solving one instance of a problem and the step of generalizing the solution to that one instance to all instances in the problem. Figure 6.9 below shows how the improved wording of the Generalize-with-repeat feature explicitly explained these two steps. Strategy steps correspond to Mayer's problem-solving process model's procedural knowledge which is necessary for executing a solution to a problem (Mayer, 1990).

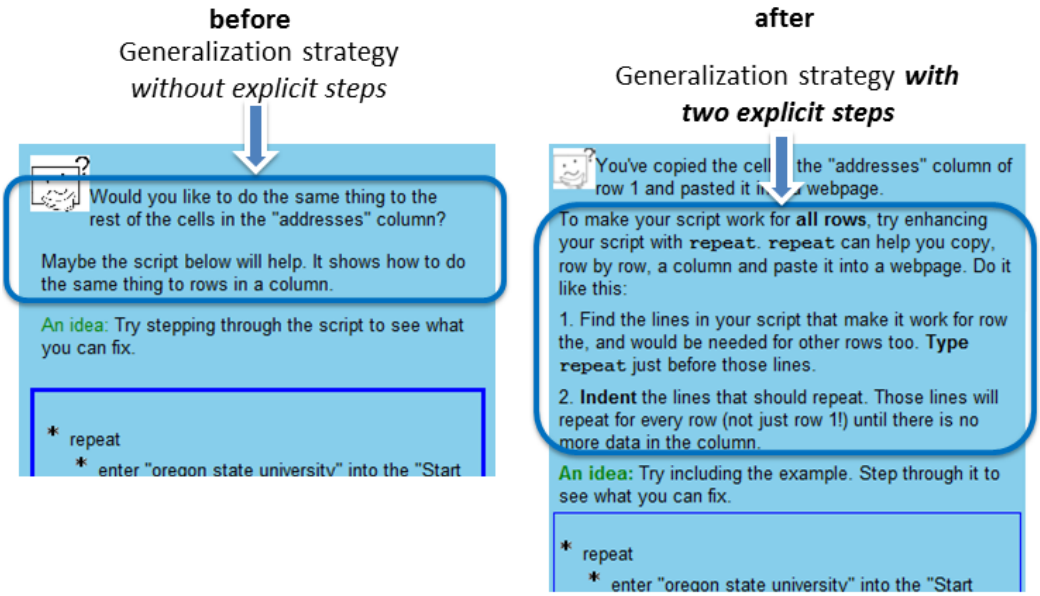


Figure 6.9 Generalize-with-repeat (context-sensitive) with steps of the generalization strategy explained explicitly.

Choice of strategies: The results from the two qualitative empirical studies showed that both the context-sensitive Second-webpage feature and the context-free Second-webpage feature had a success rate of about 50%, which was lower than the Generalize-with-repeat feature's success rate. Part of the reason seemed to be due to the difficulty participants had in applying the analogy strategy the features aimed to convey. To alleviate this problem, we introduced two additional strategies that we believed were easier to apply than the analogy strategy into the features: working backward for the context-free Second-webpage feature and divide-and-conquer for the context-sensitive Second-webpage feature in these two strategies. We emphasized the two new strategies in the features by explicitly explaining their steps. The analogy strategy's steps were not explained and thus de-emphasized. Our design rationale was that by including strategies we thought to be easier to apply, users would be able to make better use of the features. For example, Figure 6.10 shows how the context-free Second-webpage feature explains the working backwards strategy.

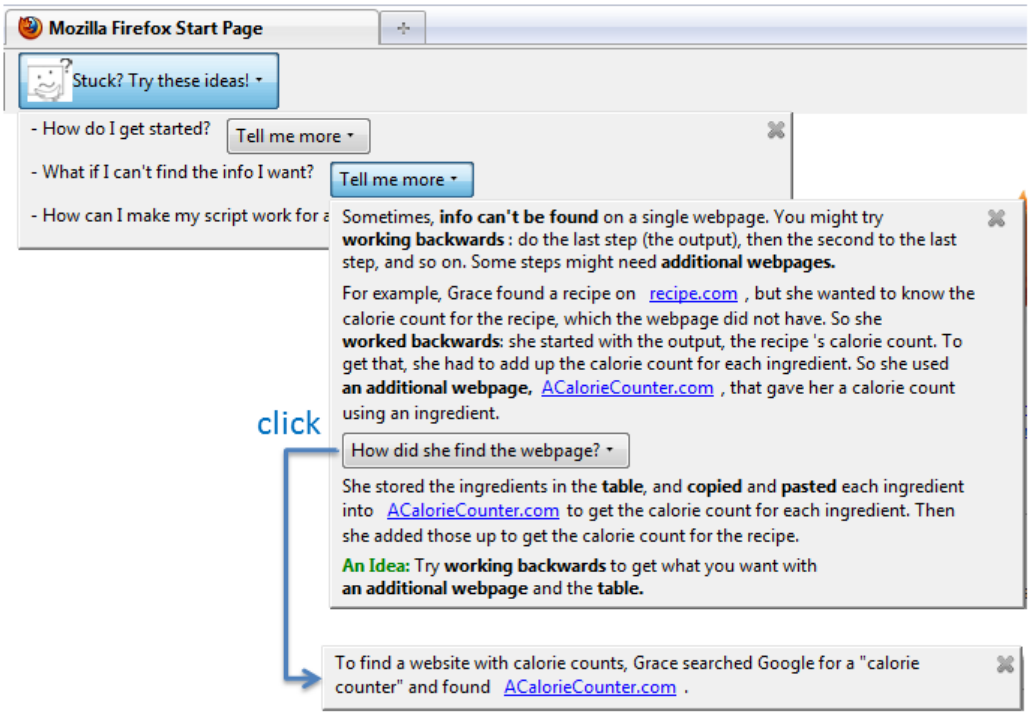


Figure 6.10 How the context-free Second-webpage feature explains the working backward strategy.

Mechanisms for implementing programming knowledge: To make the features' contents more actionable, we included specific actions for implementing the programming knowledge which we call *mechanisms*. For example, part of the mechanism to carry out the Webpage-as-Component pattern to use a second webpage to calculate driving time involves *putting "addresses" in the table* and *copying* an address from the table and *pasting* it into the second webpage address (underlined in Figure 6.11's "After" part). Mechanisms correspond to Mayer's problem-solving process model's procedural knowledge which is necessary for executing a solution to a problem (Mayer, 1990).

Figure 6.11 The context-sensitive Second Webpage feature with *mechanisms*, i.e., concrete actions to execute some programming knowledge. For example, the "copy" and the "paste" actions help execute the Webpage-as-Component pattern. (We have added the orange underlines for the purpose of the figure to point out the mechanisms used to apply the programming knowledge explained in the feature.)

One explained example in context-sensitive Second Webpage feature: Previously, a context-sensitive Second-webpage showed three intentionally flawed example websites that were not explained in detail. To add details of the relevant problem-solving strategy and programming knowledge, we focused on one example website explaining it using a concrete scenario showing how a user named Grace made use of this website as a second webpage (Figure 6.11's "After" part's second paragraph).We also reduced the number of example websites in

context-sensitive Second-webpage from three to one due to space constraints and the belief that one well-explained example would be more effective than multiple minimally explained examples.

Changes to features for the How-to-Start barrier: We replaced the *Start-with-a-column-name* (Figure 5.3) and *Finder-page* features (Figure 5.4) which had low success rates with participants in Study 1 (shown in Table 6.1), and they seemed to think that it was irrelevant to their current task. (Participants from Study 2 did not use these features.) As a result, we removed the Finder-page feature because it had failed all four participants who attempted to use it in Study 1, the lowest success rate among all features. The reason why it failed seemed to be unclear relevance: the feature showed an example script for looking for yogurt places which was not what the users were trying to achieve. As for the Start-with-a-column-name feature, since its purpose was to lead up to the Finder-page feature, we removed it as well.

To replace these two features, we created a Getting-started feature (context-sensitive, Figure 6.12). It focuses on explaining the essence of how to get started, i.e., by working backwards using a search engine to find what one wants as illustrated through a scenario with a user named Grace. The new feature's main goal was to solve the problem of perceived irrelevance.

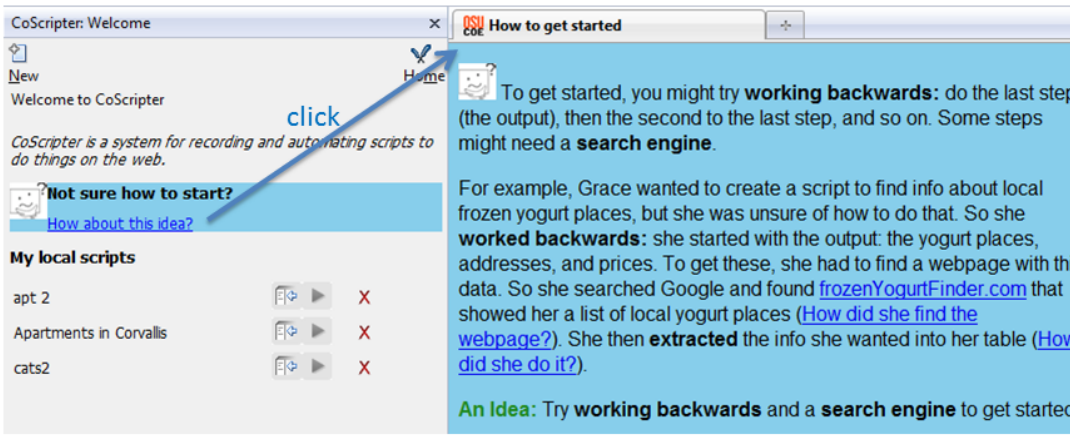


Figure 6.12 The Getting-started feature (context-sensitive) shows the user how to get started by working backwards (strategy) using a search engine (programming knowledge).

Table 6.9 summarizes changes made to the features in this round of re-design.

Table 6.9 The improved Idea Garden prototype for CoScripter -- Round 2. Changes over the previous iteration are shaded.

		Suggestions		
		Getting-started (context-sensitive: Figure 6.12, context-free: Figure 6.7)	*Second-webpage* (context-sensitive: Figure 6.11's after, context-free: Figure 6.10)	*Generalize-with-repeat* (context-sensitive: Figure 6.9, context-free: Figure 6.8's after)
Barriers targeted:	*How-to-Start*	X		
	Composition		X	
	More-than-Once			X
Programming concepts reflected in suggestions				
Data Extraction	Select a slice of structured data from a web page and putting it into the table	X		
Dataflow	Flow 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	Use a web page to find information (as opposed to computing information)	X		
Webpage-as-Component	Use a web page to compute information (as opposed to finding information)		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 the end goal, then figure out the last step to the goal, second to the last step, and so on until the givens are reached	X	X (context-free)	
Divide-and-Conquer	Divide up a problem into parts, solve individual parts, and then join the solutions to the individual parts		X (context-sensitive)	
Generalization	Solve one instance of a problem and generalize the solution to all instances in the problem			X

Chapter 7. Summative Evaluation: Can the Idea Garden Lead to Quantitative Improvements in End-User Programmers' Performance?

The results of the two qualitative studies presented in Chapter 6 showed that the Idea Garden is capable of helping end-user programmers to overcome programming barriers and learn the relevant problem-solving strategies and programming knowledge that the Idea Garden's features aimed to convey.

In this chapter, we present a quantitative summative study where we moved beyond learning to doing. We investigated whether end-user programmers who had used the Idea Garden could put their learning into practice in later programming tasks even if Idea Garden support was no longer available. Our research questions were:

RQ 1: Does the Idea Garden help end user programmers learn enough to do a programming task on their own without support?

RQ 2: Are there particular factors that help to determine end-user programmers' future success after using the Idea Garden?

7.1 The Experiment

7.1.1 Experiment Design

We conducted a between-subjects experiment with four treatments: one Control condition and three Idea Garden conditions - Strategy, Programming, and Combined. We asked our participants to first work on a programming task in CoScripter with the Idea Garden present, then asked them to perform a second programming task in CoScripter with the Idea Garden absent. Control participants did not have access to the Idea Garden during either of their tasks.

Each Idea Garden treatment contains features that address the same three barriers (i.e., How-to-Start, Composition, and More-than-Once), but each treatment's features differ in the way they address the barriers. The Strategy treatment provides suggestions to apply a problem-solving strategy; the Programming treatment provides programming knowledge. The Combined treatment contains both strategy and programming knowledge. For example, to address the Composition barrier, the context-sensitive Second-webpage feature from the

Strategy treatment contained the divide-and-conquer strategy (Figure 7.1) whereas the Programming treatment contained the Webpage-as-Component design pattern and the dataflow concept (Figure 7.2). The context-sensitive Second-webpage feature for the Combined treatment included both strategy information and programming knowledge (Figure 7.3).

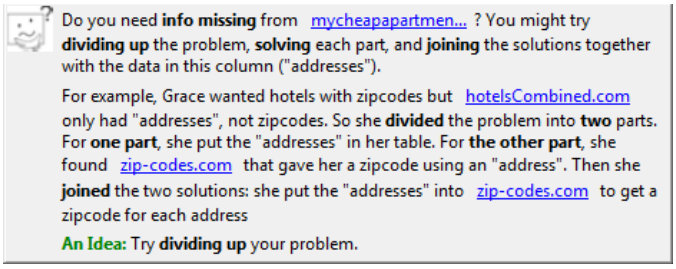


Figure 7.1 The Strategy treatment's context-sensitive Second-webpage feature. This feature describes the divide-and-conquer strategy.

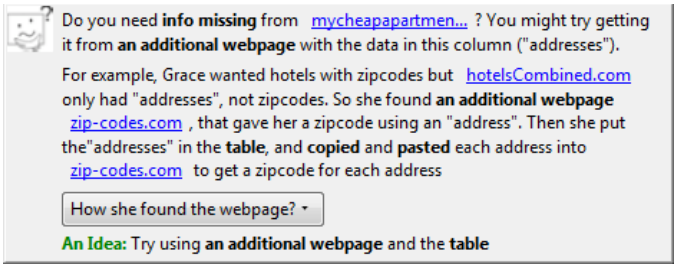


Figure 7.2 The Programming treatment's context-sensitive Second-webpage feature. This feature presents the dataflow concept and the Webpage-as-Component pattern.

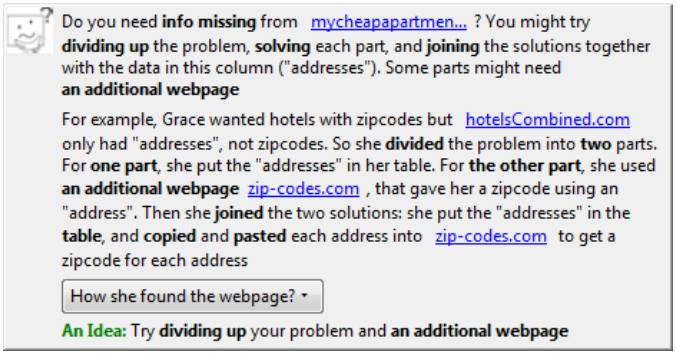


Figure 7.3 The Combined treatment's context-sensitive Second-webpage feature. This feature describes both the divide-and-conquer strategy and the dataflow concept and the Webpage-as-Component pattern.

7.1.2 Participants

We recruited undergraduate and graduate students at Oregon State University from 53 majors (e.g., English, biology, chemical engineering, human development and family studies) except computer science and electrical engineering. We disqualified any participants who had taken programming courses beyond an introductory level required for many majors' computer literacy requirements as well as anyone who had used two or more mainstream general programming languages (such as C/C++, Python, or PHP). We recruited 127 participants who met these criteria but due to data collection issues involving four participants, we were left with usable data for 123 participants.

7.1.3 Procedure

We assigned two CoScripter tasks to each participant. Idea Garden participants (those in Strategy, Programming, and Combined) had access to the Idea Garden during the first task whereas Control participants did not have access to the Idea Garden during the first task. In the second task, no participants had access to the Idea Garden. (Thus, for the Idea Garden participants, the first task is comparable to a "learning" task whereas the second task is comparable to a "learning transfer" task (Bransford, Brown, & Cocking, 2000). Idea Garden participants were not informed that the Idea Garden would be unavailable during the second task.

Participants filled out a background questionnaire and attended a 25-minute, hands-on tutorial about CoScripter functionality. The tutorial walked participants through how to create three scripts: one to look up information from a webpage, one to pull data from a webpage into the table, and one to push data from the table to a webpage. Following the tutorial, participants had 6 minutes to work on a practice task. We encouraged the participants to ask questions during this task. Participants then filled out a standard computer self-efficacy questionnaire (Compeau & Higgins, 1995) regarding CoScripter-related tasks.

Participants were given 25 minutes to work on the first task. Participants in the Idea Garden treatment had the Idea Garden enabled. To ensure that every participant was aware of the Idea Garden features, we interrupted the participants twelve minutes into the task to draw their

attention to the context-free features. Scripts and tables were saved every 30 seconds or whenever the user pressed the "save" button.

After the first task, Idea Garden participants filled out an opinion questionnaire regarding the context-sensitive and context-free versions of the three features. The questions displayed a picture of the feature and asked, "This feature helped me accomplish my task". A participant could respond using a five-point Likert scale or could indicate "Never saw" for the context-sensitive features. Participants could also leave comments about the features. To be consistent across all conditions, Control participants were asked to fill out a questionnaire containing questions that did not relate to any aspect of our study.

Participants were given 30 minutes to work on the second task, during which the Idea Garden was not available. After the second task, participants filled out a post-task self-efficacy questionnaire. Every participant was provided the opportunity to leave feedback about each task directly on the task sheet.

7.1.4 Tasks

Each participant worked on two tasks. The apartment task (Apt) asked a participant to create a script that searched for two bedroom apartments within ten minutes' driving time of the Ohio State University campus and were under $1,300. The Pet task asked a participant to create a script that searched for cats to adopt in the Corvallis area that were shorthair breed and from a reputable shelter. In the task descriptions, we listed the expected outputs of the scripts: a record of time from each apartment to campus in the table (for Apartment) or a record of the number of reviews for each shelter (for Pet) in the table.

The two tasks were intended to be equally difficult (although as we shall see, they were not). Each task consisted of three subtasks that required the same knowledge to accomplish: (1) using a second webpage to compute the missing information (e.g., using Yelp.com to find the number of reviews for a pet shelter listed on PetFinder.com), (2) using the `repeat` command to iterate over data (e.g., pet shelter names from PetFinder.com) in the table rows to compute the missing information, and (3) using the `copy` and the `paste` commands to pull the result of each computation (e.g., number of reviews for each shelter from Yelp.com) into the table.

7.2 Analysis Methodology

7.2.1 Task Performance

Since we were interested in whether participants could write higher-quality programs for a programming task only after interacting with the Idea Garden features, we focused on the second task's performance only. Thus, whenever we mention "task performance", we mean the second task where the Idea Garden was not available.

To evaluate the quality of each participant's performance in the second task, we graded the scripts and tables generated during the task. We graded three scripts: the largest auto-saved script, the most recent auto-saved script, and latest user-saved script, along with the accompanying tables. This resulted in three task scores per participant, from which we chose the highest score. We did this because we observed many users starting new scripts, making it difficult to verify which one was the user's "intended" solution.

We graded the scripts and tables against a rubric based on the three subtasks from Section 7.1.4. Each subtask was worth five points for a total of 15 possible points. Two researchers split up the scripts and the tables and graded them independently. Then, one researcher double-checked the grading. Since the rubrics were objective, we did not measure inter-rater agreement.

To compare Idea Garden participants' performance to that of control participants, we used Fisher's Exact test. We calculated a grand median score for all participants in the experiment and then assigned participants into the group of "equal to or above the grand median" or "below the grand median" and ran Fisher's Exact test on the counts in these groups. We did not use ANOVA because the scores did not fit a normal distribution (Kolomgorov-Smirnov $D=0.7361$, $p<2.2e-16$) nor did we use Kruskal-Wallis because of a large number of ties.

7.2.2 Ratings of Idea Garden's' Helpfulness

To assess participant's overall opinions of the features' helpfulness, we calculated each participant's average rating of the Idea Garden features the participant saw. Using a one-sample *t*-test, we compared the resulting average rating of the Idea Garden's helpfulness

against the expected mean of 3.0, which was a neutral rating. Two researchers coded whether participants reported difficulties about each task.

7.3 Results

7.3.1 RQ1: Does the Idea Garden help end users do programming tasks on their own?

In the aggregate, evidence that the Idea Garden helped was not strong. Idea Garden participants averaged higher scores than Control participants (Table 7.1), but the difference was not significant (at $p<.05$).

Table 7.1 Summary of participants' performance scores

Treatment	N	Mean	Median	StdDev
Control	28	5.3	3	5.1
Idea Garden (3 treatments)	95	5.9	4	4.8

However, analysis with the one-sample *t*-test showed that Idea Garden participants' reports of the Idea Garden's helpfulness from the post-session questionnaire were significantly higher than expected ($t=3.22$ $p=.00176$). The one-sample *t*-test compares the mean score of a sample (here, ratings of the Idea Garden by participants who experienced it) to an expected population mean (here, a rating of "neutral"). Table 7.2 summarizes.

Table 7.2 Average Participant Ratings of the helpfulness of the 6 Idea Garden Features. (On 94 instead of 95 participants because one Idea Garden participant did not rate any features.) In this paper, significant values are highlighted. ***: p<.001, **: p<.01, *: p<.05.

Average response to "This feature helped me accomplish my task" (5-point Likert)	Number of Participants
>3.0	57 (60.6%) ratings averaged agreement
=3.0	13 (13.8%) ratings averaged neutral
<3.0	24 (22.5%) ratings averaged disagreement
Sample mean = 3.22 One-sample t-statistic = 3.22, DF = 93, p-value =.00176***	

Given that so many Idea Garden participants found the Idea Garden features helpful, did aggregating the data mask effects on performance? This leads to our second research question, whether there were particular factors that contributed to the success of participants after using the Idea Garden. If the answer is no, a possible explanation of our results might be that, as in our previous study (Cao, et al., 2012), some participants who learned something from the Idea Garden were simply not able to apply their learning to overcoming barriers on their own.

However, another possibility is that the Idea Garden may have been helpful to only particular participants for only particular situations. We investigate this possibility next by considering the possible factors of *who* the Idea Garden may have helped and *when* it may have helped them.

7.3.2 RQ2: Factors affecting success with the Idea Garden: Who and When?

Regarding *who*, it is common for empirical studies of end-user programmers to include people with "little or no knowledge" of programming (e.g., (Dorn, 2011), (Gross, Yang, & Kelleher, 2011), (Kuttal, Sarma, & Rothermel, 2011), (Zang & Rosson, 2008)), but was there an important difference between the "little" vs. the "no" subpopulations?

To investigate, we separated these two subpopulations as follows. We counted anyone who said they had ever done any form of "programming" (even a course in high school, or having worked with HTML) as having little knowledge: 56 participants fell into this category. Otherwise we classified them as having no knowledge: 67 participants were in this category. We emphasize that "little" here indeed means very little: recall from Section 7.1.2, that *nobody* beyond a bare minimum of programming background was allowed to participate in the study.

Although we believed that users in the "no knowledge" category would do equally well as the "little knowledge" category because of our experiment's tutorial, those with little programming knowledge scored significantly higher than those with none at all (Fisher's test on task performance (Little knowledge: 35 participants scored at or above the grand median and 21 did not; No knowledge: 28 participants scored at or above the grand median and 39 did not) p=.0297). Thus, the "little knowledge" vs. "no knowledge" factor was implicated. In fact, this factor seems particularly important to Idea Garden evaluation because the Idea Garden

targets users most like the "little" subpopulation—i.e., users who can already do enough in the programming environment to get to a point where they encounter barriers and get stuck.

Regarding *when* (i.e., situation), a possibility that arose was a difference in difficulty between the Pet and the Apartment tasks. Participants seemed to have more trouble with the Pet task than with the Apartment task. For example, participants across *all* treatments scored an average of 2.1 points lower on Pet than on Apartment, and a significantly higher portion of participants commented on difficulties with the Pet task than did with the task (Fisher's test on comments regarding task difficulties (Pet: 25 participants described difficulties and 98 did not; Apartment: 7 described difficulties and 116 did not), p=.001). Thus, this factor was also implicated, and is also particularly important to this study because the Idea Garden is only relevant when a task is hard enough to cause users to run into difficulties.

Thus, taking the "who" and "when" factors into account, we used Fisher's exact test to compare the number of participants who scored above the grand median to those who scored below, separating by "little" vs. "no" subpopulation and separating the difficult (Pet) task from the easier (Apartment) task. For the targeted situation as per the discussion above—those with little knowledge of programming working in the fairly difficult Pet task—significantly more Idea Garden participants than Control participants scored above the grand median (Fisher's (1, 5; 15, 6), p=.0265), as illustrated by the inset in Table 7.3. Participant means in this category echo this summary, with Idea Garden participants averaging a score of 6.08 vs. the Control participants' mean of 3.51. This difference in performance was also confirmed by participants' ratings: as Table 7.4 shows, participants in the target situation rated the helpfulness of the Idea Garden features significantly higher than the expected population mean of 3.0 (one-sample t=2.46, df=20, p=.0231). Thus, we confirm that the Idea Garden helped participants with little knowledge learn enough to do a programming task on their own, without support, provided that the task was sufficiently difficult.

Table 7.3 Scores at or above (colored slices) or below (white slices) the grand median for idea Garden vs. Control, shown for all who/when combinations. (Idea Garden has more participants because it had three treatments.) Idea Garden participants scored significantly better than control participants in the Idea Garden target situation (thick border).

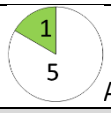
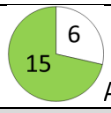
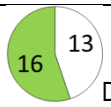
Task	Little knowledge		No knowledge	
	Control	Idea Garden	Control	Idea Garden
Pet				
	Fisher's exact test p = .0265*		Not significant	
Apt				
	Not significant		Not significant	

Table 7.4 Participant ratings of Idea Garden feature helpfulness for participants with little knowledge, in the Pet task.

Response to "This feature helped me accomplish my task"	Number of Participants
>3.0	16
= 3.0	0
<3.0	5
mean = 3.29 significantly different from expected mean of 3.0. One-sample t-statistic = 2.46, DF = 20, p =.0231*	

The remaining situations, shown also in Table 7.3, all had slightly higher scores in the Idea Garden than the Control group, but none of these reached significance. Thus, the "who" and the "when" together mattered in the Idea Garden's helpfulness to participants' ability to overcome the problems they encountered.

7.3.3 RQ2: Who alone? When alone?

Finally, we consider whether combining "who" and "when" as above obscures one of the "who" or "when" factors *alone* being responsible for the significant difference in performance in Idea Garden Participants versus Control Participants.

The answer is that neither factor alone explained the results. Table 7.5 shows suggestive differences based on subpopulation alone, and Table 7.6 shows suggestive differences based on task difficulty alone, but these differences did not rise to significance.

Table 7.5 Participants who scored at or above (colored slices) the grand median separated by little or no knowledge. In both subpopulations, Idea Garden participants scored somewhat higher than control participants, but when task was not taken into account, the differences did not rise to significance.

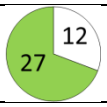
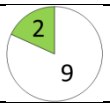
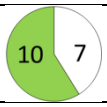
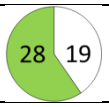
Little Knowledge		No Knowledge	
Control	Idea Garden	Control	Idea Garden
			
Not significant		Not significant	

Table 7.6 Participants who scored at or above (colored slices) the grand median separated by task. In the more difficult Pet task, Idea Garden participants scored somewhat higher than control participants, and in the easier Apt, they scored almost identically. When subpopulation was not taken into account, the differences did not rise to significance.

Pet		Apt	
Control	Idea Garden	Control	Idea Garden
			
Not significant		Not significant	

7.4 Discussion and Open Questions

7.4.1 Which Treatment Might be the Best?

Our evidence suggests that the Programming treatment worked best for participants with little knowledge doing the difficult Pet task. Programming participants scored on average 4.56 points higher than Control participants (Table 7.7). Programming had the largest percentage of participants who scored at or above the median (83.3%) (Table 7.8). Programming participants

also rated the Programming treatment's helpfulness favorably, with 100% of its participants finding the Idea Garden helpful Table 7.9. This initial evidence thus leads us toward our first open question:

> *Open Question 1: What characteristics and information associated with the Idea Garden's "Programming" treatment lead to good task performance and good subjective ratings from participants?*

Table 7.7 Summary statistic for Idea Garden participants with little knowledge who did the Pet task.

Condition	N	Mean	Median	StdDev
Control	6	3.51	2.25	4.74
Strategy	8	5.23	5.125	4.58
Programming	*6*	*8.06*	*7.9*	*4.17*
Combined	7	5.34	4	4.04

Table 7.8 Task performance of participants with little knowledge who did the pet task. Programming treatment had the highest percentage of participants scoring at or above the median.

Treatment	< grand median	>= grand median
Control	5	1 (16.7%)
Strategy	3	5 (62.5%)
Programming	*1*	*5 (83.3%)*
Combined	2	5 (71.4%)

Table 7.9 Subjective ratings of participants with little knowledge who did the pet task. The Programming treatment had 100% of its participants finding the features helpful.

Treatment	Not Helpful	Neutral	Helpful
Strategy	2	0	6 (75%)
Programming	*0*	*0*	*6 (100%)*
Combined	3	0	4 (57%)

The first reason for Programming's success may be that Programming provided knowledge that was concrete enough for participants to act upon. The Programming features aimed to convey broad programming concepts and mini design patterns, both of which are concrete and actionable. For example, Programming's context-sensitive Second-webpage feature (Figure

7.2) specifically recommends participants to use a second webpage whereas Programming's Generalize-with-repeat feature introduced the iteration concept and the related programming construct `repeat`. Such knowledge helped participants as suggested by the quotes below:

> *Participant 13411: "It was nice that [the Idea Garden] recognized when I would want to use the* `repeat` *command".*
>
> *Participant 23344: "This was helpful because getting the script to work for all rows and columns was tricky for me at first".*

In comparison, the Strategy features provided general problem solving guidance that helped some participants but confused others. Some participants could not figure out how to execute the working backward strategy or the divide-and-conquer strategy:

> *Participant 21055: "[It was] not clear enough on how to work backwards".*
>
> *Participant 23255: "It[']s an Ok suggestion, but it doesn't say how to 'join the solutions together'".*

The second possible reason why the Programming treatment did well may be related to the lengths of the content. The contents of the Programming (and Strategy) features were shorter than the Combined feature's contents, and as per the Attention Investment model (Blackwell, 2002), likely required less effort to process. Although the Combined treatment contained programming knowledge, its length was the longest amongst all treatments. The added cognitive cost involved in processing this information might have rendered the Combined treatment less effective than the Programming treatment.

7.4.2 Differences in Information Processing by Gender

When we considered all participants in our study, our results suggest that gender and treatment in combination may have had an influence on task performance and questionnaire responses. The evidence suggests that females performed better with the Combined treatment than other treatments whereas males performed better with the programming treatment. This leads us to the following open question:

Open Question 2: How can we design Idea Garden features to support both the comprehensive information processing approach taken by females and the selective approach taken by males?

The Selectivity Hypothesis posits that males and females differ in the approaches they use to process information (Myers-Levy, 1989). Males prefer to take a selective approach to information processing that relies on cues that are salient and relevant, whereas females take a comprehensive approach, processing information from a variety of available cues. In the end-user programming domain, these trends have been observed in participants debugging shell scripts (Grigoreanu, et al., 2009) as well as debugging spreadsheets (Grigoreanu, et al., 2008).

Applying this theory to our case predicts that the Combined treatment works best for females because of its completeness whereas the Programming treatment works best for males due to its brevity and its relevance to the current task. Table 7.10 shows task performance broken down by gender and treatment. For females, Combined treatment had the highest percentage of participants scoring at or above the grand median at 62% (13 out of 21). For males, the Programming group had the highest percentage of participants scoring at or above the grand median at 63% (11 out of 17), followed closely by the Strategy treatment at 63% (5 out of 8). These preliminary results suggest that female and male end user programmers do follow the Selectivity Hypothesis when examining the Idea Garden approach, and may help guide investigations into how to effectively support both female and male end user programmers in the design of Idea Garden features.

Table 7.10 Task performance of all males and females broken down by treatment. Females performed best with combined whereas males performed best with programming.

	Females		Males	
	< grand median	>= grand median	< grand median	>= grand median
Strategy	13	8 (38%)	3	5 (63%)
Programming	9	8 (47%)	6	11 (65%)
Combined	8	13 (62%)	5	6 (55%)

7.5 Summary

In this chapter, we have presented a summative evaluation of the Idea Garden prototype for CoScripter. We found evidence revealing that participants in the target population of the Idea Garden, namely end-user programmers with little programming knowledge, were able to learn from the Idea Garden, retain that knowledge, and use it toward a challenging programming task without the Idea Garden around to assist them.

Two trends emerged from this study need further investigation. One trend was that the Programming treatment seemed to have resulted in best performance and most favorable participant responses. Another trend was that males seemed to have performed the best with Programming whereas females seemed to have benefited most from Combined, consistent with the prediction of the Selectivity Hypothesis. Since both of these trends are based on counts or percentages, further investigation is needed to confirm these trends. Also, our sample size was fairly small, and we investigated only one platform, so follow-up empirical work is warranted to investigate the generality of our conclusions.

This study, together with the two empirical studies presented in Chapter 6, provide evidence that the Idea Garden is a promising approach for providing problem-solving support in end-user programming environments.

Chapter 8. Recommendations for Creations of Idea Gardens in Other Programming Environments

Through two qualitative empirical studies (Chapter 6) and a quantitative lab experiment (Chapter 7), we have shown that an Idea Garden can help end-user programmers overcome programming barriers, learn problem-solving strategies and programming knowledge, and perform significantly better than end users who had no access to the Idea Garden in a challenging programming task.

In this section we provide recommendations to the designers who are interested in creating an Idea Garden for their end-user programming environment. Toward this goal, we describe the following facets of creating an Idea Garden: understanding barriers in a host environment (Section 8.1), designing Idea Garden features (Section 8.2), and a software architecture for an Idea Garden (Section 8.3).

8.1 Understand Barriers in a Host Environment

In order to devise Idea Garden features aimed at helping end-user programmers to overcome programming barriers, we need to understand the barriers first. Two broad approaches to study barriers are empirical and literature-based.

8.1.1 Empirical Approaches to Studying Barriers

In the empirical approach, researchers directly engage in empirical studies to investigate their users' barriers. As an example, our own work included this approach. We conducted two empirical studies to gather data on barriers (Chapter 3 and Chapter 4).

Such studies may be conducted in several different ways, by focusing on programming activities, end-user programmers' personal accounts, or artifacts. One possibility is lab usability studies as taken by this thesis, using the think-aloud to attempt to tap into end-user programmers' thought processes as they worked on programming tasks. Another possibility to study activities directly is contextual inquiry or observation. For example, Subrahmaniyan observed an end-user programmer at her office during a trial period of a software visualization tool to identify barriers occurred in this trial period (Subrahmaniyan, Burnett, & Bogart, 2008). In addition to studying end-user programmers' programming activities directly, other studies

gathered data through end-user programmers' accounts of their experiences, e.g., by conducting interviews as in (Dorn & Guzdial, 2010), and/or by examining artifacts related to programming barriers such as questions posted at online discussion boards such as in (Chambers & Scaffidi, 2010) or code repositories such as in (Bogart, Burnett, Cypher, & Scaffidi, 2008).

As for data analysis, researchers might consider "borrowing" code sets from our studies, e.g., the reflection-in-action code set (Section 3.2), the ideation code set (Section 3.2), and the method of matching patterns in data with theory such as Simon's problem-solving theory (Section 4.2). Note that our codes were especially developed for the Popfly and CoScripter environments. Thus, when "borrowing", researchers will inevitably need to operationalize these codes in the context of their programming environments. For example, our ideation codes in Table 3.1 are based on blocks, the basic constructs for building programs in Popfly. Researchers will have to consider what counts as an idea in their programming environments.

8.1.2 Literature-Based Approaches to Studying Barriers

It is also possible to learn about barriers by consulting existing literature such as Ko et al.'s work on learning barriers of Visual Basic learners (Ko, Myers, & Aung, 2004) and Dorn and Guzdial's work on barriers of web developers (Dorn & Guzdial, 2010). This approach seems reasonable for formative purposes because some barriers appear to generalize across different programming environments. For example, Ko et al.'s "design" barriers were similar to Chambers and Scaffidi's "problem setting" barriers. Another example is the similarity in the debugging feature usage barriers faced by end-user programmers of Forms/3 (Beckwith, et al., 2006) and in those faced by users of Excel (Beckwith, Innman, Rector, & Burnett, 2007).

In order to choose literature closest to one's work, a researcher might use Table 2.1 as reference. This table summarizes some work on barriers in end-user programming along different dimensions. Work that aligns with the researcher's agenda in the most number of dimensions is arguably the closest to what the researcher needs. For example, for a researcher who is interested in creating an Idea Garden for a mashup environment to support idea generation, the barriers work from this thesis is probably more useful than work from, say, the domain of animation.

Literature-based approaches have pros and cons. On the upside, one might argue that these approaches can save researchers from having to conduct empirical studies which may take considerable time. In addition, building on well-researched results is one of the basic principles of scientific research. On the down side, as with the application of results from any scientific research, generalizability might be an issue. The extent to which the findings from existing literature might generalize to the researcher's situation may vary. For example, our work was partially inspired by Ko et al.'s work but we did not take Ko's barriers literally. An example is our own variant of Ko et al.'s coordination barrier which we have referred to as the Composition barrier. Our Composition barrier expanded Ko et al.'s coordination barrier to include the barrier of not knowing a goal can be achieved by combining functionality from multiple APIs whereas Ko et al.'s original description of the barrier only included difficulties in not knowing how to make multiple modules work together.

Based on this discussion, therefore, we would not recommend that researchers rely solely on literature-based approaches. Rather, we suggest using an empirical approach to supplement a literature-based approach as a means to triangulate.

8.2 Designing Idea Garden Features

Having identified a list of barriers in a target end-user programming environment, how might we go about designing Idea Garden features to help users overcome these barriers? It is important to note that such features aim to address ill-defined problems and designing them is itself an ill-defined problem. Therefore, the design process of an Idea Garden is likely to involve multiple iterations of framing, action, and reflecting, just as described in Schön's reflection-in-action theory (Schön, 1983).

8.2.1 Definition of an Idea Garden, Revisited

Recall from Section 5.5, the Idea Garden is an extension to an existing end-user programming environment that follows the principles of Minimalist Learning Theory, adopts the negotiated interruption style, and includes the elements and properties presented in Table 5.2 from Chapter 5. Given the empirical results of Chapter 6 and Chapter 7, we revise the set of elements and properties that were presented in Table 5.2, expanding the set of elements and properties.

The additions are mechanisms for carrying out programming knowledge (Section 6.8), explicit steps for carrying out strategies (Section 6.8), and having both context-sensitive and context-free versions of a feature (Section 6.4 and 6.8). We indicate the additions with highlights in Table 8.1.

Table 8.1 Elements and properties of an Idea Garden (Updated). New additions are highlighted.

Type of element/property	The elements and properties	Motivation
Feature Content	Problem-solving strategies (see Table 5.1 for example strategies.)	Suggested by our empirical results (Chapter 4, Chapter 6, Chapter 7), Simon's problem-solving theory, and Mayer's problem-solving process model.
	Programming knowledge (See Table 5.1 for example programming knowledge.) This knowledge comes in the form of concepts, design patterns, and mechanisms (Section 6.8).	
	Intentionally flawed examples (e.g., Part 4 in Figure 5.7)	In order to avoid solving users' problems for them, an Idea Garden's examples cannot be correct.
	Explicit steps for carrying out strategies (Section 6.8)	Suggested by our empirical results (Chapter 6, Chapter 7) and Mayer's problem-solving process model.
	Mechanisms for carrying out programming concepts and design patterns (Section 6.8)	Suggested by our empirical results (Chapter 6, Chapter 7) and Mayer's problem-solving process model.
Feature Form	Suggestion template (Figure 5.7)	The suggestion template prescribes the content of the features in a host-independent way. (An Idea Garden implementor then fills in the template with host-dependent details.).
	Context-sensitive (all features in Chapter 5) and context-free features (Section 6.4, 6.8)	Having both context-sensitive and context-free features allows the user to always have access to features.
Interaction Style	Interruption style: negotiated	The negotiated interruption style works better than the immediate style for users working with programming problems (Robertson, et al., 2004).
	Personality: non-authoritative	The Idea Garden should not appear to be an automatic problem-solver or an authoritative figure that can dictate what the user should do. Users are in charge of their own task and interaction with the Idea Garden.

8.2.2 Features' Content

As mentioned in Table 8.1, Idea Garden features must scaffold two sets of knowledge: strategy knowledge and programming knowledge.

8.2.2.1 Programming Knowledge

As listed in Table 8.1, an Idea Garden supports three types of programming knowledge to help users overcome a barrier: programming concepts, design patterns, and mechanisms for carrying the concepts and patterns.

We recommend two ways to find out what programming concepts their users are lacking: empirical and literature-based. An example of the empirical way is our formative work (Chapter 4). In this work, we found some users repeating copy-paste commands on each table cell without using a loop construct, a clear indication of a lack of the iteration concept. Another way is to consult the literature. As a reference, Dorn and Guzdial's paper (Dorn & Guzdial, 2010) includes a list of core programming concepts that generalizable across different programming languages and environments. Researchers can analyze their programming languages light of these programming concepts.

Likewise, empirical and literature-based approaches also apply to finding what design patterns users are lacking. As an example of the empirical way, our work has begun to explore mini design patterns in the context of CoScripter, for example, the Webpage-as-Component pattern and Repeat-Copy-Paste patterns (as explained in Table 6.9). We identified these patterns through our experience with CoScripter. Regarding the literature–based approach, some researchers have begun developing a corpus of patterns commonly found in their users' code. For example, Basawapatna et al. have developed a set of "Computational Thinking Patterns" commonly found in animations coded created by children using AgentSheets (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011). For example, collision is a pattern where two agents physically collide. Their patterns are generalizable across different animation programs created in AgentSheets (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011). These patterns appear to embody core ideas in animation programming and therefore could be useful to other animation languages.

In addition to supporting programming concepts and design patterns, an Idea Garden also supports mechanisms, information about *how* to apply the concepts and patterns. Mechanisms fall under procedural knowledge (Mayer, 1990). Therefore, part of the necessary groundwork for a new Idea Garden instance is to ascertain what mechanisms are needed for a user to carry out the programming concepts and design patterns supported by the Idea Garden instance.

8.2.2.2 Problem-Solving Strategies

As for strategies, we believe that there are at least two ways that might help researchers decide what strategies to include in their features. One way is to explore natural mappings between the programming knowledge a feature aims to convey and existing problem-solving strategies. An example is our choice of the generalization strategy for the Generalize-with-repeat feature. The iteration concept works naturally with the generalization strategy because the code for iteration embodies the "general" form that works for all individual elements. To achieve the general form, the problem-solver must generalize the solution to a single element in a set to all elements in that set, an application of the generalization strategy. Our work has begun to gather a list of common problem-solving strategies as listed in Section 4.2.1.

8.2.2.3 Intentionally flawed examples

We defer the discussion of design challenges associated with having intentionally flawed examples in Section 8.2.5 and 8.2.6.

8.2.3 Features' Form

A feature's form is used to help deliver the feature's content effectively. Table 8.1 lists two form-related elements necessary to an Idea Garden: a suggestion template and context-sensitive/-free features.

8.2.3.1 A Suggestion Template

Through our design exploration, we have been using a host-independent suggestion template to organize the content of our features. The suggestion template had a lead-in, a gist, an action item, and one or more examples (Figure 5.7 shows the initial template. The latest template is shown in Figure 8.1.). Researchers can concretize this template with information specific to their environment. Using our Second-webpage feature as an example (shown in Figure 6.11's

"After" part), its lead-in sentence infers the user's goal, e.g., "Do you need information missing from…" Then the gist explains the problem-solving strategy and programming knowledge (in the example, the divide-and-conquer strategy, the dataflow concept, and the Webpage-as-Component pattern) on a high level followed by a concrete example that explains the same strategy and programming knowledge on a detailed level. At last, an action item encourages the user to try what the feature suggests.

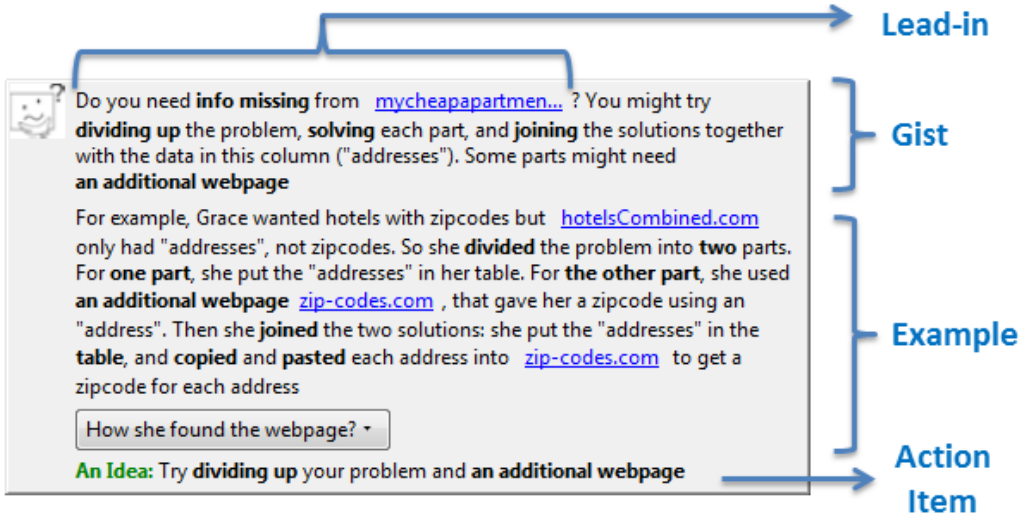


Figure 8.1 The suggestion template.

8.2.3.2 Context-Sensitive and Context-Free Versions of a Feature

As shown in Table 6.9, the context-sensitive and context-free versions of the same feature always contain the same programming knowledge because they target the same programming barrier. However, they may have different programming strategies due to the presence or absence of user's context. For example, the context-sensitive Second-webpage feature (Figure 6.11's "After" part) contains the divide-and-conquer strategy because it becomes available when a user has finished part of a task (having imported some data into the table), a context in which the task can be seen as having at least two parts, the part that has been accomplished and the part that has not. In this context, the divide-and-conquer strategy is relevant. Lacking this context, the context-free version of the Second-webpage feature (Figure 6.10) conveys the working backward strategy which is applicable regardless of how far a user is into a task.

Instead of using different strategies, the two versions can use the same strategy. An example is the Generalize-with-repeat feature where both versions use the generalization strategy.

The two versions may also use different examples due to context. For example, the context-sensitive Generalize-with-repeat feature uses an example based on user's code whereas the context-free version uses a static example.

As a general recommendation, researcher should include in the context-sensitive version of a feature strategies and examples appropriate for users' context.

8.2.4 Features' Interaction Style

We discuss two properties of the Idea Garden features' interaction style: interruption style and personality.

8.2.4.1 Interruption Style

Idea Garden uses the negotiated interruption style to notify users of it features. Using the negotiated style, the computer first needs to announce that it needs to interrupt and to have a way of allowing the user to negotiate with it. For example, when the context-sensitive Generalize-with-repeat feature is ready, the Idea Garden does not present it to its user immediately. Instead, it waits for an opportunity to notify the user that it has something to show them, e.g., when CoScripter displays a command in user's script and refreshes the screen. This is when the Idea Garden displays a button for the feature as a notification to the user (Figure 8.2). The full content of the feature is hidden from the user in the button. Note that the user is neither required to acknowledge the existence of the button nor forced to interact with it. This gives the user the control over whether to view the feature or not, and if so, when. As we have already mentioned in Section 5.3.1, the negotiated style has been found to be associated with positive debugging performance in end-user programmers (Robertson, et al., 2004).

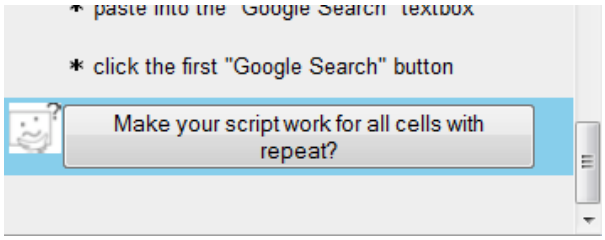


Figure 8.2 The context-sensitive Generalize-with-repeat feature's button. Idea Garden hides the content of the feature in a button.

8.2.4.2 Personality & User Expectations

The Idea Garden is neither an automatic problem-solver nor an authority. Rather, it is more like a helpful friend or a partner to a user who is perhaps not that smart. To help convey this, we have used a quizzical face to represent the Idea Garden (shown in Figure 5.2). The quizzical face aims to show the user that the Idea Garden is not completely sure of its suggestions and might make mistakes in trying to help the user.

In fact, none of the Idea Garden features' suggestions work in a "plug-in and read-to-use" fashion, an important point to convey to users. The examples cannot be used directly, e.g., context-sensitive Second-webpage's example about finding zip codes for hotels based on hotels' addresses (shown in Figure 6.11's "After" part) did not help with users' task which was about finding apartments directly. However, users' expectations based on their experience with computer generated suggestions, e.g., music recommendations, were largely leaning toward "it'd better be what I want". This has been a particular challenge to designing Idea Garden features. As the following quote from a participant in the summative study (Chapter 7) indicates, intentionally flawed examples can confuse users.

Thus, researchers need to carefully consider how to set the expectations of their Idea Garden's users.

Participant 22214: "It automated the wrong command lines for my and got me confused. I had to start the assignment all over again."

8.2.5 Coping with Design Challenges: Making Features Relevant

As mentioned in Section 8.2.1, one essential ingredient of an Idea Garden is intentionally flawed examples. This is to avoid giving away examples that users can directly plug into their program without having to learn. This design decision has in fact lead to a challenge: how to make features that provided no direct solutions to users' problem but were, at the same time, perceived by users as relevant to their tasks.

We found two tactics to be useful to increase the apparent relevance to users' tasks: relating a feature's purpose to a user's goal and contextualizing a feature's content with a user's information. While these two tactics do not attempt to address the examples directly, they helped make the examples in the features more relevant to users. To relate a feature to a user's goal, we have made the first sentence of our features to hint at our users' goal. For example, the context-sensitive Second-webpage feature (Figure 6.11's "After" part) infers that the user's goal is to find information missing from the website where the data in his/her table came from.

One might ask how one might know what goals a user might have during the course of a programming task. One way to figure this out is to perform a task analysis of the kinds of programming tasks the target end-user programming environment supports. Results from the task analysis will help inform what the "ideal" goals are. For example, the basic process of creating a mashup script in CoScripter involves the steps of gathering data from a source website (e.g., "addresses" from apartments.com), using that as input for a calculator page (e.g., input "addresses" into Google Maps) to compute information, and entering the results into a table. Each step can be seen as a goal. The second step indicates the goal of finding "info missing from [a website]" as is hinted in the lead-in sentence of the context-sensitive Second-webpage feature shown in Figure 6.11's "After" part.

The second tactic we have used to increase relevance was to contextualize features with users' information. (Note that this tactic only applies to context-sensitive.) For example, the context-sensitive Second-webpage feature uses the user's data to contextualize its content. As is shown in Figure 6.11's "After" part shows, the feature is contextualized with "mycheapapartmen…[mycheapapartments.com]" which is the website from which the user has imported data and "addresses" which is the type of data that the Idea Garden has guessed

based on the data user has imported from "mycheapartments.com". Another example is the context-sensitive Generalize-with-repeat feature which uses user's code to contextualize its content (Figure 6.4's "After" part). Including contextual information such as user's data (e.g., "addresses") and code helps to relate the feature's content to the user's current task.

8.2.6 Coping with Design Challenges: Balancing Good and Intentionally Flawed Parts of a Feature

The main goal of an Idea Garden is to help users learn to help themselves. In order for users to benefit from the Idea Garden features, the features must convey relevant knowledge. For example, the Second-webpage feature (context-sensitive, Figure 6.11's "After" part) explains the divide-and-conquer strategy, the dataflow concept, and the Webpage-as-Component pattern. But for users to learn to help themselves, the features cannot provide direct solutions to users' problems. For example, the context-free Second-webpage feature's example is about how to look up calories of a recipe (Figure 6.10), is unlikely to be what the user needs.

To balance the correct and intentionally flawed parts of a feature, we list the requirements in Table 8.2. The requirements are: (1) always provide correct strategy and programming knowledge in the gist part of a feature, (2) use flawed/incomplete examples which convey the correct strategy and programming knowledge, (3) allow computed information (used to contextualize features) to be incorrect. We always make sure that the gist part of a feature contains the useful strategy and programming knowledge because those are the essential contents of a feature. We explain the strategy and programming knowledge using a well-explained example which is always either flawed with the wrong topic or incomplete. This is to make sure that users need to exercise their brains in order to make use of the feature. In our latest design iteration (Section 6.8), we limited the number of examples per feature to just one because we believed that one well-explained examples would work better than multiple minimally explained examples. Computed information can be correct or incorrect. It does not contain strategy or programming knowledge.

Table 8.2 Requirements for balancing correct and intentionally flawed parts of an Idea Garden feature.

Part of Feature	Correct or Intentionally Flawed	Example in an Idea Garden Feature for CoScripter	What User Needs to Do to Make Use of the Part?
Gist	Always correct: strategy and programming knowledge	Second-webpage (context-sensitive): Figure 6.11's "After" part's first paragraph. The gist conveys the divide-and-conquer strategy, the dataflow concept, and the Webpage-as-Component pattern.	Understand the strategy and programming knowledge maybe with the help from the Examples (see next row).
Examples	Always flawed: examples with wrong themes but aims to convey correct strategies and programming knowledge	Second-webpage (context-sensitive): Figure 6.11's "After" part's second paragraph. The example is looking up a zip code for an address which is unlikely to be what the user needs.	- Recognize that the example is not meant to be used as-is. - Understand the strategy and programming knowledge underlying the example. - Apply the strategy and programming knowledge to one's task.
	Always flawed: incomplete example code but aims to convey correct strategies and programming knowledge	Generalize-with-repeat feature (context-free)'s script example: Figure 6.8's script part. The repeat script example is incomplete because it only shows the steps for looking up a price for each laptop but does not show how to put the prices back to the table.	- Understand the strategy and programming knowledge underlying the example. - Complete the missing part of the example code for one's task.
Computed info	Can be correct or incorrect	Second-webpage (context-sensitive): Figure 6.11's "After" part. "addresses" in the first paragraph is computed information; it could be incorrect. For example, Idea Garden might think that the info is "address" whereas in reality, it is a company's "name".	N/A (Computed info helps to contextualize features. It does not carry knowledge that users need to understand)

8.2.7 Coping with Design Challenges: Managing Costs and Benefits

Throughout the course of our design, we have been continuously aiming for feature designs that conveyed low perceived costs and high perceived benefits as per Attention Investment Model's recommendation (Blackwell, 2002). To raise perceived benefits, we have mainly focused on increasing the relevance of features. As we have discussed in Section 8.2.5, we have found two tactics for increasing relevance to be helpful: relating features to users' goals and contextualizing features' with users' information such a user's data or code.

As for lowering users' perceived costs for processing the features, we have been mainly concentrating on limiting the lengths of features' contents and enhancing the readability of the contents. To keep the length short, we iterated over the wordings of the features, removing unnecessary words and tightening things up. We have also reduced the number of examples used in some features, e.g., the context-free Second-webpage features (Section 6.8), from three to one out of consideration of length. To enhance the readability of features, we have been using a template to organize the contents of our features (Figure 5.7). We have introduced bolding of keywords into the features, e.g., we bolded the strategy steps and words that indicate actions for carrying out programming knowledge (Section 6.8).

8.3 Software Architecture

We used the software architecture described below to implement both the Idea Garden for CoScripter, and also to implement the Idea Garden for Gidget.

Recall from Section 5.2, that the Idea Garden is an extension of existing end-user programming environments, the host environments. The Idea Garden dynamically makes its features available to users of end-user programming environments using a stream of information received from its host. This information includes the user's code, data, and recent activities. Figure 5.1 depicts this conceptual architecture.

Under the hood, as depicted in Figure 8.3, an Idea Garden for a host environment is a *listener* of the host. It follows the listener pattern, or as in (Gamma, Helm, Johnson, & Vlissides, 1995), the Observer Pattern, and listens for relevant user events, code, and data from its host.

Idea Garden has *host-independent* components and *host-dependent* components, and an implementor of a new Idea Garden needs to customize only the host-dependent components.

The host-independent components that do not require customization for the host are indicated by a white box in the Idea Garden area of Figure 8.3: they are the Controller, Information Processor, Abstract Suggestion, Abstract Listener, and Abstract Actioner. The host-dependent components that need to be customized according to the host are indicated by a pink box in the Idea Garden area of Figure 8.3: they are the Host-Specific Listener, Host-Specific Actioner, and Host-Specific Suggestions. In the rest of the section, we will explain what each component is responsible for and how to customize the host-specific components

8.3.1 Implementing the Idea Garden as a Whole

We implemented the Idea Garden as a JavaScript class that held a reference to each of the components shown in Figure 8.3. We implemented each component as a class as well.

Given such an implementation, to create and initialize an Idea Garden, an implementor needs to insert a call from the host to the constructor of the Idea Garden class. For example, we inserted a call to Idea Garden class's constructor into CoScripter's `onLoad()` method where CoScripter's UI is initialized.

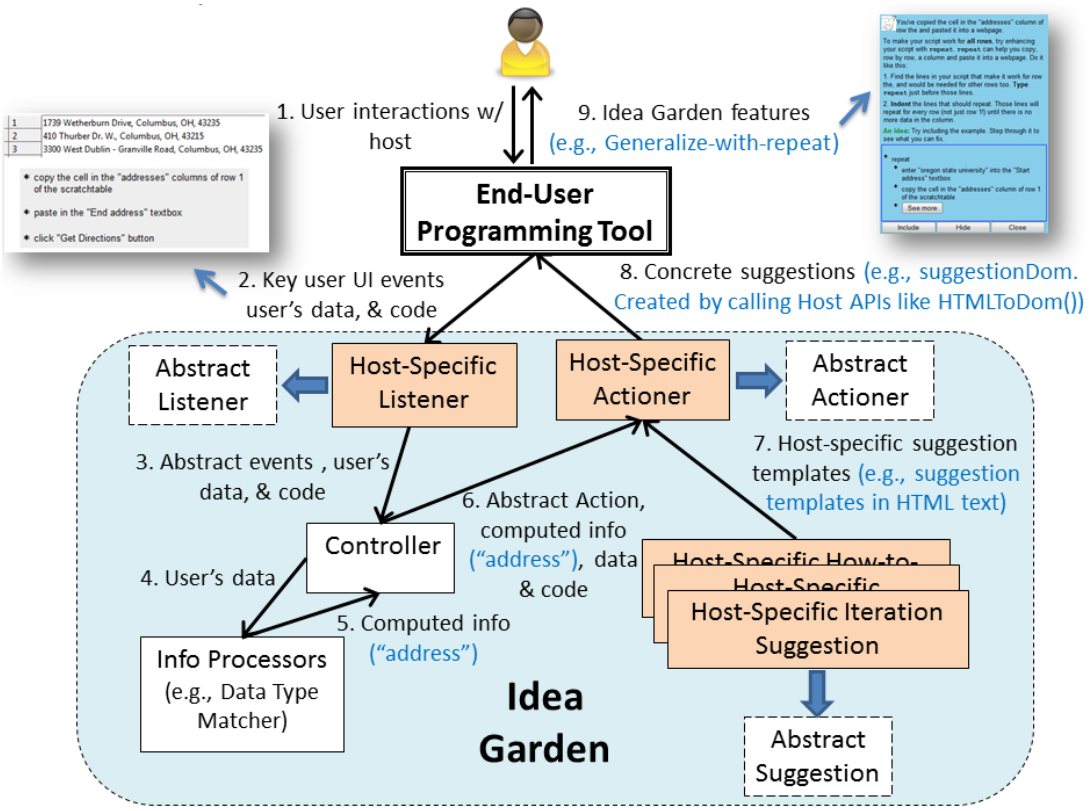


Figure 8.3 The Idea Garden software architecture. In the Idea Garden area, the pink (shaded) boxes inside the roundtangle represent host-specific components, and the white boxes represent the host-independent components. The solid-border boxes indicate concrete components. The dashed-border boxes are abstract components that some concrete components inherit from (indicated by a fat blue arrow). The thin-black arrows in the figure indicate the flow of information. Information is labeled and numbered. The blue text next to the information indicate an example of the information. For example, "address" is an example of 5. Compute info.

8.3.2 How Host Information Flows into the Idea Garden

Once an Idea Garden instance is created and initialized, in order for it to receive the relevant user events, code and data from the host, the implementor needs to insert a call to a special method of the instance, `IdeaGarden.HostInfoToIdeaGarden(events, code, data)`. The implementor places this call in the appropriate place in the host's code where the relevant information is available. For example, in CoScripter, we placed the call to `IdeaGarden.HostInfoToIdeaGarden(events, code, data)` in CoScripter's

event/code pipeline. As a detailed example, when a user types "Google Maps" into the browser's address bar, the `"go to"` event and `"go to 'Google Maps' "` command, i.e., code, become available in CoScripter's event/code pipeline (data is `null` in this example) and then flows into the Idea Garden via `IdeaGarden.HostInfoToIdeaGarden(events = "go to", code = "go to 'Google Maps'", data = null)`.

The stream of information keeps flowing into the Idea Garden instance until the host environment terminates.

8.3.3 Host-Specific Listener

When users' events, code, and data flows into the Idea Garden via `IdeaGarden.hostInfoToIdeaGarden()` method, the Host-Specific Listener abstracts them to a form suitable for sending along to the host-independent Controller. Thus, the Host-Specific Listener's two specific responsibilities are:

(1) It translates the concrete user events (and sometimes the data and the code too) into abstract events (in `listener.processHostInfo()`). For example, when a user copies a cell's content (e.g., an address) from a cell in the CoScripter table, pastes it into a webpage (e.g., Google Maps), and clicks a button to calculate (e.g., the "Get Directions" button), a sequence of "copy from table", "paste into a webpage", and "calculate" events are fired off. The Host-Specific Listener (the one for CoScripter) observes these events and translates them to an Idea Garden abstract event called "user_needs_help_with_iteration".

How to customize: To customize this functionality of the Host-Specific Listener, an implementor simply maps "interesting" concrete event sequences (interesting because empirical observations showed that they were sometimes tied to barriers) to appropriate abstract events. For CoScripter, our observations showed that users executing this event sequence sometimes needed help with iteration to make his/her script work for all cells in a column instead of just one. That is why we mapped that sequence to the corresponding abstract event of "user_needs_help_with_iteration". Table 8.3 below tabulates the CoScripter event sequences and their corresponding abstract events in the Idea Garden. (Recall that the Idea Garden does not need to be

certain that a user needs help with iteration; it is simply deciding that he/she might, so that it can have a context-sensitive suggestion about it ready if they ask for it.)

Table 8.3 Mappings between CoScripter event sequences and Idea Garden abstract events.

CoScripter Event Sequences	Idea Garden Abstract Events	Descriptions
CoScripter_starts	user_needs_help_getting_started	The "start" event indicates the start of the CoScripter application. This is when a user might be prone to the *How-to-Start* barrier, i.e., user_needs_help_getting_started.
copy_from_table, paste_to_webpage, click	user_needs_help_with_iteration	When a user has copied a row of a column and pasted it into a webpage, they might be prone to the *More-than-Once* barrier, not knowing how to generalize their script for all rows, Thus, the corresponding abstract event is user_needs_help_with_iteration.
user_imported_data_to_table	user_needs_help_with composition	When a user has imported data from a webpage, they could use the data as input to a second webpage to get additional information. The corresponding abstract event is user_needs_help_with composition.
CoScripter_starts	user_needs_context-free_suggestions	When CoScripter starts, this is the time to make the context-free features available. Therefore, the abstract event is user_needs_context-free_suggestions.

(2) The Host-Specific Listener then passes the abstract event along with users' code and data to the Controller (by calling `controller.onReceiveAbstractEvent()`). Continuing the example, the code that the Host-Specific Listener passes into the Controller includes the following CoScripter commands: `copy the cell in the "addresses" column of row 1 of the scratchtable`, `paste into the "Start address" textbox`, and `click the "Get Directions" button`. (User's data is not used in this example.) The only reason the Host-Specific Listener passes (host-specific) code/data to the host-independent Controller is so that the

Controller can in turn pass this information along. The Controller does not itself actually use host-specific information.

The Host-specific Listener inherits from an Abstract Listener which specifies the interface for the `processHostInfo()` method.

8.3.4 Controller

The Controller is independent of the host environment, and does not need to be customized. It has the following responsibilities:

(1) Upon receiving an abstract event from the Host-Specific Listener, the Controller pairs the abstract event with an abstract action. This is done via a look-up table that stores all the pairings of an abstract event to an abstract action. Continuing the example from 8.3.2, the Controller maps the "user_needs_help_with_iteration" abstract event with the "show_iteration_suggestion" abstract action.
This does not require customization because an abstract event always pairs with the same abstract action regardless of the host environment. For example, the abstract event, "user_needs_help_with_iteration" always corresponds to the abstract action, "show_iteration_suggestion" no matter if the target environment is CoScripter or Gidget or some other end-user programming environment. Table 8.4 lists the pairs of abstract events and abstract actions.

Table 8.4 Pairs of abstract events and abstract actions.

Abstract Events	Abstract Actions
user_needs_help_getting_started	show_getting_started_suggestion
user_needs_help_with_iteration	show_iteration_suggestion
user_needs_help_with composition	show_second-webpage_suggestion
user_needs_context-free_suggestions	show_context-free_suggestions

(2) The Controller then sends the matched abstract action to the Host-Specific Actioner, passing along the code that it received from the Host-Specific Listener by calling `actioner.onReceiveAbstractAction()`. In the example, the Controller sends "show_iteration_suggestion" to the Host-Specific Actioner.

(3) If the Host-Specific Listener also sent additional data to the Controller, the Controller first sends the data to the Information Processor, which calculates additional

information based on the data and sends the calculated information back to the Controller. The Controller then passes the calculated information on to the Host-Specific Actioner. (Section 8.3.5 shows an example of an Information Processor in CoScripter and what computed information it is able to produce).

8.3.5 Information Processor

The Information Processor computes additional information based on the input data. The Information Processor used in CoScripter's Idea Garden is the *Data Type Matcher*[1] which determines a *data type* based on input data. For example, for the input data "1900 SW Jefferson St.", the data type matcher would return the "address" data type. In addition to "address", the Data Type Matcher is capable of recognizing data of several other common data types such as "name", "currency", "time", and "distance".

Information Processor is host-independent and therefore does not need customization. For example, we can reuse the same Data Type Matcher for Gidget or some other end-user programming environment.

8.3.6 Host-Specific Actioner

Given an abstract event that requires an action, the Host-Specific Actioner produces an appropriate action. Thus, upon receiving an abstract action, user's code, and data (including computed data, if applicable), the Host-Specific Actioner does four things:

(1) It calls the appropriate host-specific suggestion module to get a host-specific suggestion template. For example when receiving the "show_iteration_suggestion" abstract event, the Host-Specific Actioner for CoScripter calls the CoScripter-Specific Iteration Suggestion to retrieve a suggestion template (by calling `CoScripter-SpecificIterationSuggestion.produceHTML()`).
How to customize: To customize this part, the implementor inserts a call to the appropriate Host-Specific Suggestion from the Host-Specific Actioner. We will explain how to customize a Host-Specific Suggestion in Section 8.3.7.

[1] This part of the prototype still needs to be refactored.

(2) Second, the Host-Specific Actioner needs to fill in the suggestion template so that it makes use of the user's context. For example, it fills in the iteration suggestion template, drawing appropriately from user's code (and, the computed information, if present) that it received from the Controller. For example, in the context of CoScripter which is a browser plug-in, the iteration suggestion's content, which is written in HTML, looks like the following paragraph. The underlined portions of the suggestion have been contextualized with user's data. The greyed parts have been contextualized with user's code. The output from this step is a suggestion, contextualized with users' information, in the form of a string. In the case of CoScripter, the string includes HTML tags as in the paragraph below. We will refer to it as `suggestionDivString`.

How to customize: To customize this part, the implementor inserts code that fills in the holes in the suggestion template with user's code and data from the Host-Specific Actioner, producing results like this `suggestionDivString`:

```
<div class="suggestion">
      <div> You've copied a cell from row 1 of the "addresses" column
      and pasted it into a webpage.</div>
      <div> To make your script work for all rows, try enhancing your
      script with <bold> repeat </bold>… Do it like this…:</div>
      <ul>
            <li>repeat
                  <ul>
                        <li> copy the cell in the "addresses" column
                        of row 1 of the scratchtable</li>
                        <li> paste into the "Start address"
                  textbox</li>
                        <li> click the "Get Directions" button</li>
                  </ul>
            </li>
      </ul>
      <button>Include</button>
      <button>Hide</button>
      <button>Close</button>
</div>
```

(3) Third, the Host-Specific Actioner turns the suggestion string (e.g., the `suggestionDivString` above) into UI widgets suitable for the host by calling host APIs. For example, the Host-Specific Actioner for CoScripter passes the `suggestionDivString` above into CoScripter's `HTMLToDom()` method to create a `suggestionDom`, which is an HTML DOM element that contains the suggestion. The output from this step is an UI widget, e.g., `suggestionDom`, ready to be placed into the host environment's UI such that the user will be able to view it on demand.
How to customize: The implementor inserts calls to host APIs to turn a contextualized suggestion in the form of a string into UI widgets that can be displayed in the host.

(4) Finally, the Host-Specific Actioner decorates the host environment's UI to make the suggestion available to the user. For example, the Host-Specific Actioner for CoScripter calls CoScripter's `insertButton()`[2] method to place `suggestionDom` in the CoScripter's script panel. As a result, an indicator of the CoScripter-specific iteration suggestion has been created and placed into the CoScripter UI. Notice that it is shown as a button (Figure 8.4 left). When the user clicks the button, the full content of the suggestion will be shown (Figure 8.4 right).

[2] This method is called `insertSuggestion()` in our prototype but it is not specific to the Idea Garden. We added this method to add UI functionality that CoScripter did not have. It is possible that the implementor will need to similarly add missing UI capabilities for other environments.

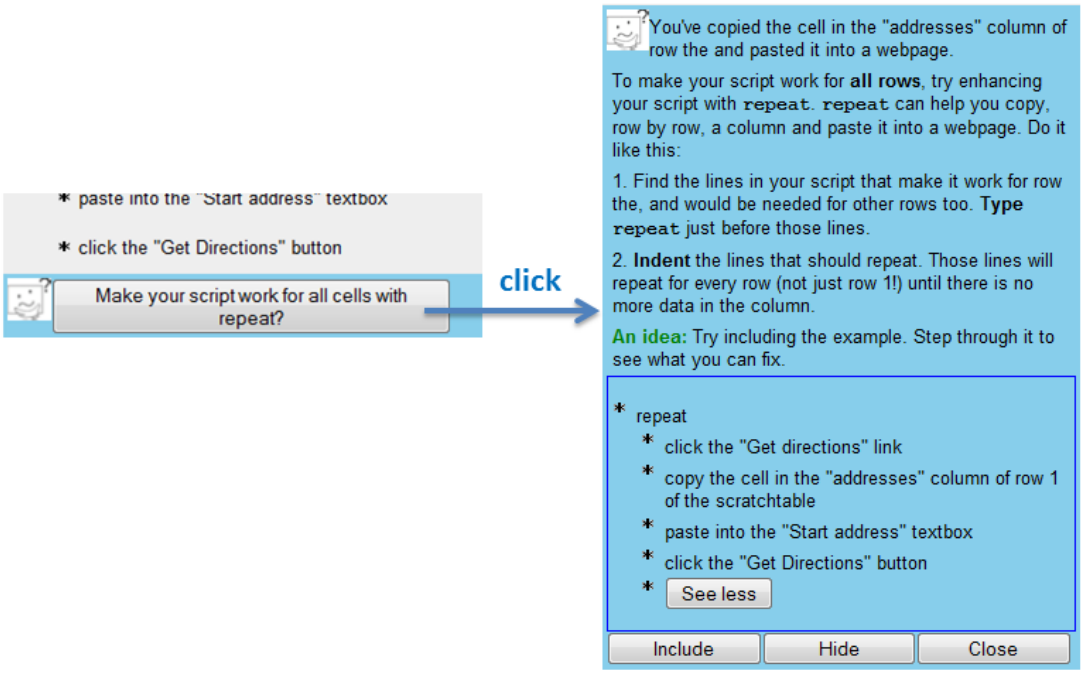


Figure 8.4 Host-Specific Actioner calls host APIs to decorate the host environment with UI widget. For example, the CoScripter iteration suggestion is shown as a button (left). If a user clicks it, it displays the full suggestion (right).

How to customize: To customize this part, an implementor needs to insert calls to the appropriate host APIs from the Host-Specific Actioner to place the widget in the host's UI.

8.3.7 Host-Specific Suggestions

A suggestion addresses a barrier. A Host-Specific Suggestion addresses a barrier in a way that is specific to a host. For example, the following table shows the difference between the part of the suggestion content of a *host-independent* iteration suggestion and a CoScripter Host-Specific Iteration Suggestion.

To customize a Host-Specific Suggestion, an implementor needs to replace the wordings in "[ ]" in the host-independent suggestion with wordings specific to the host environment. For example, in Row 1 of Table 8.5, we replaced "[OPERATED ON ELEMENT X OF SET Y]" with the host-specific phrase "copied a cell from row 1 of the 'address' column and pasted it

into a webpage". As another example, in Row 1 of Table 8.5, we replaced "[PROGRAM]" with "script" because a script is a program in the context of CoScripter.

A Host-Specific Suggestion inherits from Suggestion which defines interfaces such as `produceHTML()` for getting the content of a suggestion

Table 8.5 An example of a host-independent suggestion and a host-specific suggestion. The greyed parts of the host-independent suggestion are specialized in the grey parts of the host-specific suggestion.

Line #	Host-Independent Iteration Suggestion	(Host-Specific) CoScripter-Specific Iteration Suggestion
1	You've [OPERATED ON ELEMENT X OF SET Y].	You've copied a cell from row 1 of the "address" column and pasted it into a webpage.
2	To make your [PROGRAM] work for all [ELEMENTS], try enhancing your [PROGRAM] with <bold> [LOOP CONSTRUCT] </bold>… Do it like this…	To make your script work for all rows, try enhancing your script with <bold> repeat </bold>… Do it like this…
3	<!-- PLACE HOLDER FOR EXAMPLE PROGRAM -->	 - repeat - copy the cell in the "addresses" column of row 2 of the scratchtable …

Chapter 9. Conclusions and Future Work

9.1 Summary and Contributions

In this thesis, we have presented the Idea Garden approach for devising problem-solving support for end-user programmers. The Idea Garden was inspired by theories from design, creativity, and problem-solving (Chapter 2), and was further informed by our formative empirical work investigating barriers end-user programmers faced (Chapter 3, Chapter 4). The Idea Garden focuses on bringing problem-solving strategies and programming knowledge to end-user programmers in the context of their self-directed programming task. It is in the form of features extending an existing end-user programming environment.

To empirically evaluate the Idea Garden approach, we prototyped a proof-of-concept Idea Garden extension for the CoScripter environment with features targeting three common barriers in end-user programming literature (Chapter 5). We found, through an iterative design and evaluation process, that with the help of the Idea Garden features, users were not only able to overcome the barriers but also learn the relevant problem-solving strategies and programming knowledge the features aimed to convey (Chapter 6). Furthermore, our quantitative summative evaluation yielded evidence that, by being exposed to the Idea Garden during one programming task, participants from our target population were able to learn enough from the Idea Garden and create more correct code in a challenging subsequent task without further help from the Idea Garden (Chapter 7).

The Idea Garden is a general concept, not specific to CoScripter. The extent of its generality is seen in the parts of the Idea Garden architecture that are host-independent, and the fact that adding an Idea Garden to some other end-user programming environment can be implemented by customizing five classes (Chapter 8). We also summarized some design recommendations and offered lessons we have learned about particular design challenges that can arise, for researchers who are interested in creating an Idea Garden for their own end-user programming environments (Chapter 8).

Our work's primary contribution is the Idea Garden approach for devising problem-solving support in existing end-user programming environments. We have demonstrated through a proof-of-concept of an Idea Garden and empirical studies of the prototype that our approach is

not only feasible but also beneficial to end-user programmers in helping them *learn to problem solve on their own*.

9.2 Future Work

The ultimate goal of an Idea Garden is to help end-user programmers become better problem solvers—even if they do not start out with a goal of learning better problem-solving, but simply want to get their tasks done Work in this thesis has just begun to explore one opportunity toward accomplishing this goal, i.e., by giving end-user programmers some starter ideas to work with when they are unable to overcome a barrier on their own. Future work might explore additional ways in which an Idea Garden might promote problem-solving skills in end-user programmers. We list a few possibilities below.

Encouraging Flexibility: Creativity literature suggests the more flexible a person is, the more varieties of ideas he/she is likely to produce. Flexibility is also linked to the ability to reframe: very different ideas might suggest a change in a user's frame. Our work has begun to address this question by giving users starter ideas. However, it largely remains an open question as to how an Idea Garden might encourage flexibility. Perhaps, one way might be to facilitate the mixing of existing ideas a user already has as creativity literature suggests that production of ideas depends upon the contents of a problem solver's mind and how he/she "mix" these ingredients (Osborn, 1953). This would require a programming environment to allow for exploration of multiple ideas simultaneously. Programs such as the Parallel Pies (Terry & Mynatt, 2004) have begun providing such capabilities. Another way might be to promote flexibility through contrasting ideas as mentioned in Section 2.2.2. This suggests support for measuring distance between ideas. An example approach is the one taken by Koh et al. in their work on measuring divergence between students' code and tutorial programs (Koh, Bennett, & Repenning, 2011).

Support for Adequate Elaboration: Adding details to an idea could make an idea better. In the Popfly study (Chapter 3), we observed participants avoiding elaboration and abandoning potentially good ideas early on, a phenomenon we termed *under-elaboration*. We also saw participants elaborating excessively and missed the opportunity to consider other potentially useful ideas, a phenomenon we termed over-elaboration. As we suggested in Section 3.5.2, an Idea Garden might explore ways to help its users detect over- and under-elaboration and

maybe even help channel their efforts, e.g., by balancing the cost of elaboration to prevent under- and/or over-elaboration.

Allowing Different Idea Representations: Designers represent ideas using various representations such as drawings, objects, and written words that vary in the level and amount of details they capture (Visser, 2006). Designers have been found to use imprecise representations, e.g., rough sketches, in early design to express the provisional characters of their ideas (Goel, 1995). We have argued that end-user programmers are designers as well (Chapter 2), and therefore, we believe that they, too, could benefit from using different design representations. However, it seems that most existing end-user programming environments allows for the code itself as the sole representation of ideas (and comments, usually interleaved with the code, as secondary notation). One disadvantage of this is that code requires precision. Tending to coding details early on may lead to premature commitments to ideas that would not work out in the end such as in the case of F4 from the Popfly study (Chapter 3) who prematurely fixated on using VirtualEarth. Another disadvantage of code being the only representation of ideas is that it does not allow for different abstractions of ideas. Future work might explore ways in which idea representations for different levels of abstraction might be supported.

Helping Idea Evaluation: Besides helping with idea generation, we imagine an Idea Garden assisting users in *idea evaluation*. We think that one important skill in idea evaluation is to be able to form and test hypotheses. Hypothesis testing is an integral part of the scientific method and the foundation of modern science. In our studies, we have observed signs of hypothesis testing in some participants, e.g., M2 from Section 3.5.1 tested the hypothesis that connecting blocks in series might work but it did not work so he rejected that hypothesis. Such observations show that hypothesis testing is viable with end-user programmers. An Idea Garden could provide support for forming hypotheses, organizing evidence for hypotheses, and tracking which hypotheses worked or not.

Another important skill useful for evaluating an idea is variable isolation, the ability to separate variables and their results. Whereas these skills are picked up by computer science students by taking programming courses, they seem to have remained "in the dark" for many end-user programmers. For example, F4 as mentioned in Section 3.6 accumulated many ideas before evaluating them, making it impossible to isolate the effects of individual ideas. An Idea

Garden might help users like F4 to develop the ability to isolate variables by linking the effects of an idea to the code.

Foster Traits of a Good Designer, a Creative Thinker, and Great Problem-Solver: We believe that it takes more than general problem-solving strategies and programming domain knowledge to become a good designer, creative thinker, and a great problem solver. As already mentioned in Section 2.2, literature has identified personal traits associated with successful designers and highly creative problem solvers. Such traits include being comfortable with ambiguity, willingness to take risks, perseverance, and self-confidence (Amabile, 1996; Cross, 2006). It remains an open question as to how an Idea Garden might foster such traits. Work in end-user programming, e.g., (Grigoreanu, et al., 2008), has shown that it is possible for feature design to influence personal traits like self-efficacy, a form of self-confidence. We believe that fostering traits like these may help end-user programmers move one step closer toward becoming good problem solvers.

Bibliography

Alexander, A. (1979). *The timeless way of building.* Oxford University Press.

Amabile, T. (1996). *Creativity in context: Update to the social psychology of creativity.* Westview Press.

Andersen, R., & Mørch, A. (2009). Mutual development: A case study in customer-initiated software product development. In V. Pipek, M. B. Rosson, B. de Ruyter, & V. Wulf, *End-user development* (pp. 31-49). Berlin/Heidelberg: Springer.

Anderson, L., Krathwohl, D., Airasian, P., Cruikshank, K., Mayer, R., Pintrich, P., . . . Wittrock, M. (2000). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Allyn & Bacon.

Bandura, A. (1977). Self-efficacy: Toward a unifying theory behavioral change. *Psychological Review, 8*(2), 191-215.

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. C., & Marshall, K. S. (2011). Recognizing computational thinking patterns . *ACM Technical Symposium on Computer Science Education* (pp. 245-250). Dallas, Texas: ACM Press.

Bayazit, N. (2004). Investigating design: A review of forty years of design research. *Design Issues, 20*(1), 16-29.

Bazjanac, V. (1974). Architectural design theory: Models of the design process. In W. Spillers, *Basic Questions of Design Theory* (Vol. 3). Amsterdam: North-Holland Publishing Co.

Beckwith, L., Burnett, M., Grigoreanu, V., & Wiedenbeck, S. (2006). Gender HCI: What about the software? *Computer, 39*(11), 83-87.

Beckwith, L., Burnett, M., Wiedenbeck, S., Cook, C., Sorte, S., & Hastings, M. (2005). Effectiveness of end-user debugging software features: Are there gender issues? *ACM Conference on Human Factors in Computing Systems* (pp. 869-878). Portland, OR, USA: ACM.

Beckwith, L., Innman, D., Rector, K., & Burnett, M. M. (2007). On to the real world: Gender and self-efficacy in Excel. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 119-126). IEEE Press.

Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., & Cook, C. (2006). Tinkering and Gender in End-User Programmers' Debugging. *ACM Conference on Human Factors in Computing Systems* (pp. 231-240). Montreal, Canada: ACM.

Blackwell, A. (2002). First steps in programming: A rationale for attention investment models. *Symposium on Human-Centric Computing Languages and Environment* (pp. 2-10). IEEE.

Blackwell, A., & Hague, R. (2001). AutoHAN: An architecture for programming the home. *Human-Centric Computing Languages and Environments* (pp. 150-157). Stresa: IEEE.

Bloom, B., & Broder, L. (1950). *Problem solving processes of college students: a Supplementary educational monograph.* Chicago: University of Chicago Press.

Boden, M. (1994). *The dimensions of creativity.* Cambridge, London, England: MIT Press.

Bogart, C., Burnett, M. M., Cypher, A., & Scaffidi, C. (2008). End-user programming in the wild: A field study of CoScripter scripts. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 39-46). Herrsching am Ammersee, Germany: IEEE Press.

Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S. (2010). Example-centric programming: Integrating web search into the programming environment. *ACM Conference on Human Factors in Computing Systems* (pp. 513-522). Atlanta: ACM.

Brandt, J., Guo, P. J., Lewenstein, J., & Klemmer, S. R. (2008). Opportunistic programming: How rapid ideation and prototyping occur in practice. *WEUSE IV: International Workshop on End-User Software Engineering* (pp. 1-5). Leipzig, Germany: ACM Press.

Bransford, J., Brown, A., & Cocking, R. (2000). *How People Learn: Brain, Mind, Experience, and School.* National Academy Press.

Burnett, M., Fleming, S., Iqbal, S., Venolia, G., Rajaram, V., Farooq, U., . . . Czerwinski, M. (2010). Gender differences and programming environments: Across programming populations. *International Symposium on Empirical Software Engineering and Measurement.* Bolzano-Bozen, Italy.

Cao, J., Fleming, S., & Burnett, M. (2011). An exploration of design opportunities for “gardening” end-user programmers' ideas. *International Symposium on Visual Languages and Human-Centric Computing* (pp. 35-42). Pittsburg: IEEE Press.

Cao, J., Kwan, I., White, R., Fleming, S., Burnett, M., & Scaffidi, C. (2012). From barriers to learning in the idea garden: An empirical study. *International Symposium on Visual Languages and Human-Centric Computing* (pp. 59-66). Innsbruk, Austria: IEEE Press.

Cao, J., Rector, K., Park, T., Fleming, S., Burnett, M., & Wiedenbeck, S. (2010). A debugging perspective on end-user mashup programming. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 149–156). Marid, Spain: IEEE.

Cao, J., Riche, Y., Wiedenbeck, S., Burnett, M., & Grigoreanu, V. (2010). End-user mashup programming: Through the design lens. *ACM Conference on Human Factors in Computing Systems* (pp. 1009-1018). Atlanta, USA: ACM.

Carroll, J. (1998). *Minimalism Beyond the Nurnberg Funnel.* MIT Press.

Carroll, J., & Rosson, M. B. (1987). Paradox of the active user. In J. Carroll, *Interfacing thought: Cognitive aspects of human-computer interaction,* (pp. 80-111). MIT Press.

Chambers, C., & Scaffidi, C. (2010). Struggling to excel: A field study of challenges faced by spreadsheet users. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 187-194). Pittsburg, USA: IEEE.

Compeau, D., & Higgins, C. (1995). Computer self-efficacy: Development of a measure and initial test. *MIS Quarterly, 19*(2), 189-211.

Costabile, M., Mussio, P., Parasiliti Provenza, L., & Piccinno, A. (2009). Supporting end users to be co-designers of their tools. In V. Pipek, M. B. Rosson, B. de Ruyter, & V. Wulf, *End-user development.* Berlin/Heidelberg: Springer.

Cross, N. (2006). *Designerly Ways of Knowing.* London: Springer.

Cross, N. (2007). From a Design Science to a Design Discipline Understanding Designerly Ways of Knowing and Thinking. In R. Michel, *Design Research Now: Essays and Selected Projects* (pp. 41-54). Basel: Birkhäuser.

Cypher, A., Dontcheva, M., Lau, T., & J. Nichols, J. (2010). *No code required: Giving Uusers tools to transform the web.* Elsevier .

Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly, 13*(3), 319-340.

Díaz, P., Aedo, I., Rosson, M., & Carroll, J. (2010). A visual tool for using design patterns as pattern languages. *Advanced Visual Interfaces* (pp. 67-74). Capri Island: ACM.

Do, E. Y.-L., & Gross, M. D. (2007). Environments for creativity – A lab for making things. *ACM Conference on Cognition and Creativity* (pp. 27-36). Washington DC, USA: ACM.

Dorn, B. (2011). ScriptABLE: Supporting informal learning with cases. *International Computing Education Research Conference* (pp. 69-76). Providence, Rhode Island, USA: ACM.

Dorn, B., & Guzdial, M. (2010). Learning on the job: Characterizing the programming knowledge and learning strategies of web designers. *ACM Symposium on Human Factors in Computing System* (pp. 703-712). Atlanta, GA: ACM Press.

Dorst, K., & Cross, N. (2001). Creativity in the design process: co-evolution of problem–solution. *Design Studies* (pp. 425-437). Elsevier.

Ennals, R., Brewer, E., Garofalakis, M., Shadle, M., & Gandhi, P. (2007). Intel Mash Maker: Join the web. *SIGMOD Record, 36*(4), 27–33.

Fischer, G. (2009). End-user development and meta-design: foundations for cultures of participation. *International Symposium on End-User Development* (pp. 3-14). Siegen, Germany: Springer-Verlag.

Flavell, J. (1979). Metacognition and cognitive monitoring: A new area of cognitive-developmental inquiry. *American Psychologist, 34*, 906-911.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software.* Addison.

Gantt, M., & Nardi, B. (1992). Gardeners and gurus: Patterns of cooperation among CAD users. *ACM Conference on Human Factors in Computing Systems* (pp. 107–118). Monterey, California: ACM.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science, 7*, 155-170.

Gero, J. S., & Kannengiesser, U. (2004). The situated function–behaviour–structure framework. *Design Studies, 25*(4), 373-391.

Goel, V. (1995). *Sketches of thought.* Cambridge, MA: MIT Press.

Green, T., & Petre, M. (1996). Usability analysis of visual programming environments: A "cognitive dimensions" framework. *Journal of Visual Languages and Computing, 7*(2), 131-174.

Grigoreanu, V., Brundage, J., Bahna, E., Burnett, M., ElRif, P., & Snover, J. (2009). Males' and females' script debugging strategies. *International Symposium on End-User Development.* Siegen, Germany.

Grigoreanu, V., Cao, J., Kulesza, T., Bogart, C., Rector, K., Burnett, M., & Wiedenbeck, S. (2008). Can feature design reduce the gender gap in end-user software development environments? *IEEE Symposium on Visual Languaes and Human-Centric Computing* (pp. 149-156). Herrsching am Ammerssee, Germany: IEEE Press.

Gross, P., & Kelleher, C. (2009). Non-programmers identifying functionality in unfamiliar code: strategies. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 75-82). Corvallis, OR, USA: IEEE.

Gross, P., Herstand, M., Hodges, J., & Kelleher, C. (2010). A code reuse interface for non-programmer middle school students. *ACM Symposium on User Interface Software and Technology* (pp. 219–228). New York: ACM.

Gross, P., Yang, J., & Kelleher, C. (2011). Dinah: An interface to assist non-programmers with selecting program code causing graphical output. *ACM Conference on Human Factors in Computing Systems* (pp. 3397-3400). Vancouver BC, Canada: ACM.

Guilford, J. P. (1968). *Intelligence, Creativity and Their Educational Implications.* San Diego, CA, USA: Robert R. Knapp.

Hartmann, B., Doorley, S., & Klemmer, S. R. (2008, July-September). Hacking, mashing, gluing: Understanding opportunistic design. *IEEE Pervasive Computing, 7*(3), pp. 46-54.

Hartmann, B., MacDougall, D., Brandt, J., & S. Klemmer, S. (2010). What would other programmers do: Suggesting solutions to error messages. *ACM Conference on Human Factors in Computing System* (pp. 1019–1028). Atlanta, U.S.A.: ACM Press.

Hartmann, B., Yu, L., Allison, A., Yang, Y., & and Klemmer, S. (2008). Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning. *ACM Symposium on User Interface Software and Technology* (pp. 91-100). Monterey, CA, USA: ACM.

Hayes, J. R. (1978). *Cognitive psychology: Thinking and creating.* Homewood, IL: Dorsey Press.

Hundhausen, C. D., Farley, S., & Brown, J. L. (2009). Can direct manipulation lower the barriers to computer programming and promote transfer of training? An experimental study. *ACM Transactions on Computer-Human Interaction, 16*(3), 13:1-13:40.

Jansson, D., & Smith, S. (1991). Design fixation. *Design Studies, 12*(1), 3-11.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson, *In Cognitive Skills and their Acquisition* (pp. 255-283). Hillsdale, NJ, USA: Lawrence Erlbaum Associates.

Jonassen, D. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development, 48*(4), 63-85.

Kannengiesser, U., & Gero, J. S. (2007). A function–behavior–structure ontology of processes. *Artificial Intelligence for Engineering, Design, and Manufacturing, 21*, 379-391.

Kelleher, C. (2009). Barriers to Programming Engagement. *Advances in Gender and Education, 1*, 5-10.

Kelleher, C., & Pausch, R. (2006). Lessons learned from designing a programming system to support middle school girls creating animated stories. *Symposium on Visual Languaes and Human-Centric Computing* (pp. 165-172). Brighton: IEEE.

Kissinger, C., Burnett, M., Stumpf, S., Subrahmaniyan, N., Beckwith, L. Y., & Rosson, M. B. (2006). Supporting end-user debugging: What do users want to know? *The International Conference on Advanced Visual Interfaces* (pp. 135-142). Venice, Italy: ACM.

Ko, A., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., . . . Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM Computing Survey, 43*(3), 21:1-21:44.

Ko, A., Myers, B., & Aung, H. (2004). Six learning barriers in end-user programming systems. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 199-206). Rome, Italy: IEEE.

Koh, K. H., Bennett, V., & Repenning, A. (2011). Computing indicators of creativity. *ACM Creativity & Cognition.* Atlanta, GA: ACM Press.

Kolodner, J. (1993). *Case-Based Reasoning.* San Mateo, CA, USA: Morgan Kaufmann Publishers, Inc.

Kolodner, J. L., Owensby, J. N., & Guzdial, M. (2004). Theory and practice of case-based learning aids. In D. H. Jonassen, *Theoretical Foundations of Learning Environments* (pp. 215-242). Mahwah, NJ, USA, N.J., U.S.A: Lawrence Erlbaum Associates.

Krutchen, P. (2005). Casting Software Design in the Function-Structure-Behavior Framework. *IEEE Software, 22*(2), 52-58.

Kulesza, T., Stumpf, S., Burnett, M., & Kwan, I. (2012). Tell me more? The effects of mental model soundness personalizing an intelligent agent. *ACM Symposium on Human Factors in Computing Systems* (pp. 1-10). Austin, TX: ACM Press.

Kuttal, S. K., Sarma, A., & Rothermel, G. (2011). History repeats itself more easily when you log it: Versioning for mashups. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 69-72). Pittsburg, PA, USA: IEEE.

Landay, J., & Myers, B. (2001). Sketching interfaces: Toward more human interface design. *IEEE Computer, 34*(3), 56-64.

Levine, M. (1994). *Effective Problem Solving.* Prentice Hall.

Lieberman, H. (2001). *Your wish is my command: Programming by example.* Morgan Kaufmann.

Lin, J., Wong, J., Nichols, J., Cypher, A., & Lau, T. (2009). End-user programming of mashups with Vegemite. *International Conference on Intelligent User Interface* (pp. 97–106). Island of Madeira: ACM.

Little, G., Lau, T., Cypher, A., Lin, J., Haber, D., & Kandogan, E. (2007). Koala: Capture, share, automate, personalize business processes on the web. *ACM Conference on Human Factors in Computing Systems* (pp. 943-946). San Jose: ACM.

Lowenstein, G. (1994). The psychology of curiosity. *Psychological Bulletin, 116*(1), 75-98.

Lubart, T. I. (2005). How can computers be partners in the creative process: classification and commentary on the special issue. *International Journal of Human-Computer Studies, 63*(4-5), 365-369.

Mayer, R. E. (1990). Problem solving. In M. W. Eysenck, *The Blackwell dictionary of cognitive.*

McFarlane, D. (2002). Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Human-Computer Interaction, 17*(1), 63-139.

Milgram, R. M. (1990). Creativity: An idea whose time has come and gone. In R. Mark, & A. Robert, *Theories of Creativity* (pp. 215–233). London, UK: Sage Publications.

Miller, R., Bolin, M., Chilton, L., Little, G., Webber, M., & Yu, C.-H. (2010). Rewriting the web with chickenfoot. In A. Cypher, M. Dontcheva, T. Lau, & J. Nichols, *In No Code Required: Giving Users Tools to Transform the Web* (pp. 39-63). Morgan Kaufmann.

Myers, B. A., Pane, J. F., & Ko, A. (2004). Natural programming languages and environments. *Communications of the ACM, 47*(9), 47-52.

Myers, B., Park, S., Nakano, Y., Mueller, G., & Ko, A. (2008). How designers design and program interactive behaviors. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 177-184.). Herrsching am Ammerssee, Germany: IEEE.

Myers-Levy, J. (1989). Gender differences in information processing: A selectivity Interpretation. In P. Cafferata, & A. Tybout, *Cognitive and Affective Responses to Advertising.* Lexington Books.

Nardi, B. (1993). *Small Matter of Programming: Perspectives on End-User Computing.* Cambridge, MA, USA: MIT Press.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

Newman, M., Lin, J., Hong, J., & Landay, J. (2003). DENIM: An informal web site design tool inspired by observations of practice. *Human-computer interaction , 18*(3), 259-324.

Oney, S., & Myers, B. (2009). FireCrystal: Understanding interactive behaviors in dynamic web pages. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 105-108). Corvallis: IEEE.

Osborn, A. (1953). *Applied Imagination: Principles and Procedures of Creative Problem Solving.* Charles Scribner's Sons.

Paivio, A. (1983). The empirical case for dual coding. In J. Yuille, *Imagery, Memory, and Cognition: Essays in Honor of Allan Paivio.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Pane, J., & Myers, B. (2006). More natural programming languages and environments. In H. Lieberman, F. Paterno, & V. Wulf, *End user development* (pp. 31-50). Springer.

Pane, J., Ratanamahatana, C., & Myers, B. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies, 54*(2), 237-264.

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

Reeves, J. W. (1992). What is Software Design? *C++ Journal, 2*(2). Retrieved 2009 йил 14-12 from C++ Journal: http://www.developerdotstar.com/mag/articles/reeves_design.html

Repenning, A., & Ioannidou, A. (2008). Broadening participation through scalable game design. *International Conference on Software Engineering* (pp. 305–309). Leipzig: ACM.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., . . . Silver, J. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60-67.

Rist, R. S. (1991). Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Human-Computer Interaction, 6*(1), 1-46.

Rittel, H., & Webber, M. M. (1984). Dilemmas in a general general theory of planning. In N. Cross, *Developments in Design Methodology* (pp. 135-144). Chichester: J. Wiley & Sons.

Robertson, T., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J., Beckwith, L., & Phalgune, A. (2004). Impact of interruption style on end-user debugging. *ACM Conference on Human Factors in Computing Systems* (pp. 287-294). ACM Press.

Rode, J., & Rosson, M. B. (2003). Programming at runtime: Requirements and paradigms for nonprogrammers' web application development. *Symposium on Human-Centric Computing Languages and Environments* (pp. 23-30). New York: IEEE Press.

Rode, J., Rosson, M., & Pérez-Quiñones, M. (2002). *The challenges of web engineering and requirements for better tool support.* Blacksburg, Virginia: Virginia Tech Computer Science.

Rogers, I., Sharp, H., & Preece, J. (2011). *Interaction Design: Beyond Human - Computer Interaction.* John Wiley & Sons.

Rosson, M. B., & Carroll, J. M. (1993). Active programming strategies for reuse. *Proceedings of ECOOP'93: Object-Oriented Programming. 7th European Conference* (pp. 5-20). Kaiserslautem, Germany: Springer-Verlag.

Rowe, M. (1978). *Teaching Science as Continuous Inquiry.* New York, NY: McGraw-Hill.

Runco, M. (2007). *Creativity Theories and Themes: Research, Development, and Practice.* Burlington, MA: Elsevier Academic Press.

Sawyer, R. K. (2006). *Explaining Creativity: The Science of Human Innovation.* New York: Oxford University Press.

Scaffidi, C. (2010). Sharing, finding and reusing end-user code for reformatting and validating data. *Journal of Visual Languages and Computing, 21*(4), 230-245.

Scaffidi, C., Bogart, C., Burnett, M., Cypher, A., Myers, B., & Shaw, M. (2009). Predicting reuse of end-user web macro scripts. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 93–100). Corvallis: IEEE.

Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 207-214). Dallas, TX, USA: IEEE.

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

Seals, C., Rosson, M. B., Carroll, J. M., Lewis, T., & Colson, L. (2002). Fun learning stagecast creator: An exercise in minimalism and collaboration. *Symposia on Human Centric Computing Languages and Environments.* IEEE Press.

Seaman, C. (1999). Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering , 25*(4), 557–572.

Shneidermann, B. (2007). Creativity support tools: Accelerating Discovery and Innovation. *Communications of the ACM, 50*(12), 20-32.

Simon, H. A. (1980). Problem solving and education. In D. Tuma, & F. Reif, *Problem Solving and Education: Issues in Teaching and Research.* Lawrence Erlbaum.

Simpson, M., & Viller, S. (2004). Observing architectural design: Improving the development of collaborative design environments. *Lecture Notes in Computer Science , 3190*(1), 12–20.

Soloway, E. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations. *Communication of the ACM, 29*(9), 850-858.

Spool, J., Profetti, C., & Britain, D. (2004). *Designing for the scent of information.* User Interface Engineering.

Stolee, K., & Elbaum, S. (2011). Refactoring Pipe-like mashups for end-user programmers. *ACM International Conference on Software Engineering* (pp. 81-90). Honolulu, Hawaii, USA: ACM Press.

Strauss, A., & Corbin, J. (1998). *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Springer.

Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., . . . Fern, X. (2008). Testing vs. code inspection vs. what else?: male and female end users' debugging strategies. *ACM Conference on Human Factors in Computing Systems* (pp. 617-626). Florence, Italy: ACM Press.

Subrahmaniyan, N., Burnett, M., & Bogart, C. (2008). Software visualization for end-user programmers: trial period obstacles. *ACM Symposium on Software Visualization* (pp. 135-144). Herrsching am Ammersee, Germany: ACM Press.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257-285.

Terry, M., & Mynatt, E. (2002). Recognizing creative needs in user interface design. *Conference on Creativity & Cognition* (pp. 38-44). Loughborough, UK: ACM.

Terry, M., & Mynatt, E. (2004). Variation in element and action: Supporting simultaneous development of alternative solutions. *ACM Symposium on Human Factors in Computing Systems* (pp. 711-718). Vienna: ACM Press.

Valkenburg, R., & Dorst, K. (1998). The reflective practice of design teams. *Design Studies, 19*(3), 249-271.

Visser, W. (2006). Designing as construction of representations: A dynamic viewpoint in cognitive design research. *Human-Computer Interaction, 21*(1), 103-152.

Wickelgren, W. (1974). *How to Solve Problems: Elements of a Theory of Problems and Problem Solving.* W. H. Freeman & Company.

Wikipedia. (2012, 11 30). *Mashup (web_application_hybrid)*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)

Wikipedia. (2012, Oct 20). *Microsoft Popfly*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Microsoft_Popfly

Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., . . . Rothermel, G. (2003). Harnessing curiosity to increase correctness in end-user programming. *ACM Conference on Human Factors in Computing Systems* (pp. 305–312). Fort Lauderdale: ACM Press.

Wong, J., & Hong, J. (2007). Making mashups with Marmite: Towards end-user programming for the web. *ACM Conference on Human Factors in Computing Systems* (pp. 1435–1444). San Jose, U.S.A.: ACM Press.

Zang, N., & Rosson, M. (2008). What's in a mashup? And why? Studying the perceptions of web-active end users. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 31-38). Herrsching am Ammerssee, Germany: IEEE Press.

Zang, N., & Rosson, M. (2009). Playing with information: How end users think about and integrate dynamic data. *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 85–92). Corvallis, OR, USA: IEEE Press.

Appendices

Appendix A: Background Questionnaire

(Question 1-7 were shared by all studies in this thesis. Question 8 – 13 were shared by the two studies in Chapter 6 and the summative study in Chapter 7)

1. Gender (circle your selection): Male / Female
2. Age < 20 20 – 29 30 – 39 40 – 49 50 – 59 60+
3. Major: _____
4. Year or Degree Completed: Fresh. Soph. Jun. Sen. Post Bac. Grad.
5. Cumulative GPA: _____
6. Do you have previous programming experience?
    a. High school:
        - How many courses? _____
        - What programming languages?

          _____
    b. College:
        - How many courses? _____
        - What programming languages?

          _____
    c. Professional and/or recreational
        - How many years? _____

- What programming languages?

_____

7. Is English your primary language? Yes / No

If not, how long have you been speaking English? _____ years.

For the following questions, answer Yes or No. If you are not sure, please ask us.

8. Have you ever used CoScripter before? If yes, for what purpose?
   Yes No

9. Have you ever done mashup programming? If yes, in what language AND for what purpose?

   Yes No

10. Have you ever created a program that used information from one Web page as input for another Web page (e.g., use hotels' name from hotel.com as search queries on bbb.com)? If yes, in what language AND for what purpose?
    Yes No

11. Have you ever created a program that read information off of Web pages? If yes, in what language AND for what purpose?
    Yes No

12. Have you ever created a program that entered information into Web pages? If yes, in what language and for what purpose?
    Yes No

13. Have you ever programmed a loop (e.g., like "repeat")? If yes, in what language AND for what purpose?
    Yes No

Appendix B: Self-Efficacy Questionnaire

(This questionnaire was shared by all studies in this thesis. In the Popfly study where we replaced "CoScripter" with "Popfly". In each study, we administered the questionnaire twice, once before the task(s) and again after.)

The following questions ask you to indicate whether you could use a CoScripter-like software tool for creating scripts (that automate tasks you carry out 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 tool.

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

	Could you complete the job?	Not at all confident			Moderately confident						Totally confident
... if there was no one around to tell me what to do as I go.	YES NO	1	2	3	4	5	6	7	8	9	10
... if I had never used a similar CoScripter-like tool before.	YES NO	1	2	3	4	5	6	7	8	9	10
... if I had only the software manuals for references.	YES NO	1	2	3	4	5	6	7	8	9	10
... if I had seen someone else doing it before trying it myself.	YES NO	1	2	3	4	5	6	7	8	9	10
... if I could call someone for help if I got stuck.	YES NO	1	2	3	4	5	6	7	8	9	10
... if someone else had helped me get started.	YES NO	1	2	3	4	5	6	7	8	9	10
... if I had a lot of time to complete the task.	YES NO	1	2	3	4	5	6	7	8	9	10
... if I had just the built-in help facility for assistance.	YES NO	1	2	3	4	5	6	7	8	9	10
... if someone showed me how to do it first.	YES NO	1	2	3	4	5	6	7	8	9	10
... if I had used similar CoScripter-like tools before this one to do this same task.	YES NO	1	2	3	4	5	6	7	8	9	10

Appendix C: Tutorial

(This version of the tutorial was used in the summative study. Other studies had their own tutorials with think-aloud practice where appropriate)

May I have your attention, please.

My name is [TUTORIALIZER'S NAME]. I will be leading you through today's study.

I'll be reading from this script so that I'm consistent across all study sessions.

Before we start, I'd like to remind you to please **turn off your cell phone.**

Make sure both helpers and participants' phones are OFF, not vibrate

The **aim of our research** is to understand how people create scripts for automating tasks that they do on the web.

Scripts are just a type of computer program that **users like you can create**.

CoScripter

Today, we're using a tool called **CoScripter** to create scripts. Please look here.

[Point to the screen]

As you can see, this is just a regular web browser loaded with a webpage.

CoScripter has two parts, a **script panel** and a **table**.

Example script: Flight Status

The **script panel** is loaded with **a script** created by a user like you. It's called "Flight Status".

The **purpose** of the script is that at the click of a button, the script automatically checks the status of a flight.

Now I'm going to ask you to **step through the script** so you can get a feel for what it does. I will tell you what to do. **Do NOT move ahead** or do anything that I'm not asking you to do.

Now, the first line of the script reads: go to "AlaskaAir.com".

What do you think this line will do?

To check that out, click the **Step button** in the top left hand corner of the script panel.

Take a look at the web page right here

So the first line loaded Alaska Airlines' webpage for you.

The second line reads: click the "FLIGHT STATUS" link. What do you think this line will do?

Click the **Step button** to check that out.

Take a look at the webpage right here.

The second line clicked the "Flight Status" link for you so now you're on the Flight Status page.

The next line reads: enter "2154" into the "Flight Number" textbox

Click the **Step button** to see what it does.

Take a look at the webpage right here.

The script has entered a flight number for you.

The next line is: click the first "CONTINUE" button

Now, click the **Step button**.

Take a look at the webpage right here.

So the script clicked the "Continue" button for you. Now you have the status of a flight as a result of stepping through the script.

Experiment Procedure

So here's the plan for today. I'll lead you through **a tutorial**, and then you will have **2 experimental tasks** to work on.

- As we go through this tutorial, I want you to **perform** the steps I'm describing. Do **NOT** move ahead or do things that you are **NOT** asked to do.
- If you have any questions, please **ask**.

Flight Status Tutorial

Now let's **create the flight status script**.

To do that, we need to start a new script.

Click CoScripter's "**Home**" button right here, then click the "**New**" button.

Recording

In CoScripter, one way to create a script is to have CoScripter **record** our actions.

You see how the red **"Record" button** is flashing? This means that CoScripter is ready to record our actions in the browser.

Name your script "**My Flight Status**" in the box near the top of the script panel.

In your browser's address bar, type "**AlaskaAir.com**" and hit Enter.

Read the first line of your script silently.

Click the "Flight Status" link.

Read the second line of your script silently.

In the "Flight Number" box, enter "2154".

Click the "**Continue**" button.

You see how the script has been growing?

Now you have successfully created your first script.

Editing

Now I have a question for you. Suppose you want to make your script work for a different flight, what would you do? Please think silently.

One way to do this is to **edit** the script you have already created. We want to replace the flight number "2154" with "2552".

But before we edit the script, we need to stop the recorder. Do that now.

Click the line you want to edit which starts with the word "enter". You will see a cursor.

Replace the flight number with **"2552".**

[Driver hits enter to make a blank line]

Does anyone have a blank line? Whenever you have a blank line, you should remove it. Do it like this. Click the line to select it.

Click the **arrow** before the line so that the line turns blue.

Hit the “Delete” key on your keyboard.

Whenever you see an empty line like this, you should remove it.

To check out the result of the script, we want to **run** the script from the beginning.

So **select** the first line and click “**Run**”.

You see how this is showing you the **status of a different flight**?

Let’s make **one more change**. Add the word “**You**” before the **third** line so it becomes “**You** enter “2552” into the “Flight Number” textbox.”

Run your script from the beginning.

See how the script pauses at the line that starts with “**You**”?

“**You**” is a CoScripter **command**. Other commands that you have seen so far in the script include **click,** and **go to**.

CoScripter pauses whenever it sees the “You” command. That means that **you** need to do that step yourself.

Now, fill in the flight number yourself.

To resume, click the **Stop** button, then the **Run** button.

Take a look at your webpage. You see how it’s showing you the status of flight #2552?

The “You” command is **helpful** when there’s a step that you want to do on your own instead of having the script do it for you, like when you’re entering your password into a website.

Now, please **remove the word “You”** from the line in the script

In addition to adding and removing words, you can add or remove **a line**.

To add a line before the first line, click the first line in the script.

Move the cursor to the beginning of the line so it comes right before “**go**”.

Hit the **“Enter” key**.

You can also **remove a line**. Let’s remove the line we’ve just added. Click the **arrow** before the line so that the line turns blue.

Hit the “Delete” key on your keyboard.

Save your script by clicking the “Save” button near the top of the script panel.

You can tell that it’s saved because it says “Script saved” right above the title of your script.

To **see all your saved scripts**, click the Home button.

To view your script, click the white script icon next to “My flight status” to open it.

Now we have completed the first part of the tutorial. Do you have any questions?

[Tutorializer answers their questions]

The Table feature

Now let’s look at the “**Table**”.

The table allows you to store information that you find on the Web

Web →Table: import data from web page (bulk)

To create a new table, click the **“New Table…”** button right here.

You see how it already has a couple columns and rows in the table?

To **add a column,** click the **“+” button** here.

We are going to use this table to aid in the **creation of an itinerary**.

Suppose you are planning a trip to the Chicago area and you would like to find out about hotels there.

Let’s start a new script.

Name it “**Hotel Information**”.

In the address bar, type “**marriott.com**” and hit enter.

In the first textbox here, type “**Chicago**”. Do **NOT** pick from the auto complete choices because CoScripter doesn’t recognize those.

Click outside of the auto complete choices.

Look at your script. Your script should read “enter ‘Chicago’ into the first ‘City or Airport Code’ textbox”.

Depending on where you click, your script might read like this: “click the "bg-find.jpg\"" button”. That’s because you clicked an image that is part of the webpage’s background.

[Driver: if you don’t see the click command in your script, try clicking near the white line separator]

This “click” command is not useful to your script. If you have it, remove it now.

Whenever you see a click command like this, you should delete it.

Good. So CoScripter might record actions you don’t need. You have to remove those actions from your script yourself.

Let’s continue. For “Check in” date, **type** “**9 slash 15 slash 12**”. **Do NOT** pick from the calendar because CoScripter does not recognize it.

Look at your script. Your script should read “enter ‘9/15/12’ into the ‘Check-in date (mm/dd/yy)’ textbox”

For “Check out” date, **type** “**9 slash 20 slash 12**”.

Click the “**Find**” button.

If you happen to have closed the popup window, then you probably have a “click” in your script that you don’t need. Remove it now.

Now look at the web page. Do you see a list of results? Scroll down if you don’t see them.

We want to **store the data** in the table.

Before we do that, please name the table “**Hotels in Chicago**”.

We want hotel names in our table, so **name the first column “Hotel Names**”. **Double-click** the column header to edit it.

We also want prices. So name the second column “**Prices**”.

Save your table.

You see how it shows up in the table list?

Now we’re ready to store info in our table. **Click the “Import data from webpage” button.**

Hover your mouse over the webpage but **do NOT** click anything. You see how different parts of the webpage get highlighted as the mouse moves over them?

To tell CoScripter that you would like to put the hotel names into the table, click the **first hotel's name** where it says [FIRST HOTEL'S NAME].

We also want prices. So click the price next to [FIRST HOTEL'S NAME] which is [PRICE].

Now we have selected **EVERYTHING** we want, that is, hotel names and prices. We can go ahead and tell CoScripter to import the data into the table. So click the "**Done selecting. Import data…**" button.

You see how CoScripter has populated the table with hotel names and prices?

Read the last line of your script silently.

It should read: extract the "hotels in Chicago" scratchtable

Save your table.

Stop what you're doing and look here. For the next step, just watch. Note that you have to **pick everything you need at once.** Things you pick at **a different time will be appended to the columns you already have.**

Take a look at the screen here to see what I mean by that.

[ONLY the Driver does this] For example, we already have hotels and prices in this table. Watch what happens when we import the "addresses" this time.

You see how the addresses got appended to the "Hotel Names" column?

That was the wrong way to do it.

Another thing to note is that, depending on how you select items on a web page, CoScripter might or might not be able to import the data you want into your table.

Watch what happens if we select the hotel names differently from last time.

[ONLY the Driver does this: imports by clicking the big box slowly switch between highlighted big and little box] This time, we click the big box, instead of the small box.

You see how nothing got imported into the table?

That was the wrong way to do it.

In addition, **saving your table is critical**. This allows CoScripter to replay the table extraction step. **Please remember to always save your table after you have imported data.**

So save your table **now**.

Stop and look here.

For CoScripter to run a table extraction command, the table that the extract command is referring to must be open.

In our script, the extract command refers to the “Hotels in Chicago” table. If we have the wrong table open and we try to run the script the script will not run correctly.

[driver just circles the “Hotels in Chicago” table name in the extract command]

Web →Table: Copy a single item

What if you only need one item from a web page? For example, the number of hotels available.

[Hover! ☺ Driver circles number of hotels on the webpage.]

You can use the **browser’s copy and paste menus**.

Name your third column “Number of hotels”.

To copy, highlight the number of hotels near the top of the list of hotels.

[Driver: highlights everything starting with the number, e.g., 74 hotels … 8 brands]

Go to the browser’s Edit menu.

Select copy.

Read the last line of your script silently.

The last line reads: copy the "Choose from" text, which sounds a bit weird but let me explain.

When you selected “[NUMBER of] hotels”, CoScripter found “Choose from” next to the number of hotels. So CoScripter used it as the label for the number of hotels.

[Driver circles choose from text on webpage]

So what if you see a line like this in your script, and you are not sure what it means? For example, what is the “ ‘Choose from’ ” text?

In that case, you should stop the recorder. Then, click that line. Do that now.

See how CoScripter highlights number of hotels in green?

This is how you can tell.

Start the recorder.

Now, we want to paste the number of hotels into the table. So **double-click** the first cell of the column named “Number of hotels”.

Go to the browser’s Edit menu.

Select Paste.

Read the last line of your script silently.

Now you know how to move a single item from a webpage into your table. You must use the browser’s Edit menu.

Stop and look here.

For the next few steps, just watch and don’t do them.

Sometimes, CoScripter would record you copying from a webpage, but it would NOT run that step for you.

[ONLY the Driver does this] For example, we copy “Gallery” and “List”. CoScripter records “copy the text”.

But if we stop the recorder, and click that line, CoScripter shows a red box around the line, and says “error parsing the step”.

[Driver: if the line does not turn red, click on “paste”, then click it again]

This means that CoScripter cannot recognize what you have copied. In this case, you need to find a different webpage that CoScripter can work with.

Other times CoScripter would also say “error parsing the step” if you mistyped a command.

[ONLY the Driver does this] For example, in the first line, if we typed “go too”, CoScripter would say “error parsing this step” because it does not recognize “go too”. Let’s fix that.

You see how the line turns green once we fixed it?

Table → Web

So now we have a list of hotels in our table.

Suppose that in order to decide which hotel to stay at, you would like to look up each hotel on Better Business Bureau's website to find out more about each of them, for example if they are registered with the BBB, and if they have received any complaints.

Let's create a script to find that out. So first save your script and then start a new script.

Name your new script **"Hotels on BBB**".

In the address bar up here, type "**chicago.bbb.org**". And hit enter.

Click "Check out a business or charity"

To look up the first hotel, we need to copy the first hotel name and paste it into the search box.

To copy, **double-click** the cell of the first hotel. When it turns blue, go to the browser's Edit menu and select Copy.

Take a look at your script.

Your script should read: copy the cell in the "Hotel Names" column of row 1 of the scratchtable

To paste, click inside of the search box. Go to the Edit menu and click paste.

Take a look at your script.

Your script should read: paste into the "Search For: Business Name, Type [e.g., \"Plumbers\"], URL, Phone" textbox

Click the **Search** button.

Take a look at the result.

This is how you look up hotels stored in your table.

Please remember to **always use the Edit menu** in your browser when moving data from a table to a Web page.

You can also use CTRL C to copy, or CTRL V to paste. CoScripter works both ways.

Repeat

What If we want to look up each hotel without having to do that manually?

Well, we can use the **repeat** command which needs to be entered by hand.

We have to stop the recorder before we edit the script. Do that now.

Insert a line after the second line.

Type **repeat** there.

Having the repeat command alone isn't enough. We have to tell the repeat command **which lines to repeat.** We do that **by indenting** the lines following the repeat command.

We want everything after the repeat command to be repeated, **so we need to tab in every line** that comes after repeat.

Stop what you're doing and look at the screen here to see how it's done.

[Only the driver does this].

Now, try for yourself. Do it like this. Click the line after repeat. You should see a cursor.

Hit the Home key on your keyboard. That will bring the cursor to the start of the line.

Now hit the Tab key.

Do that for each line following repeat

To see what the script will do, click the first line, then click **the Step button several times to move through the script**.

What do you notice in this line that starts with "copy"?

[Wait for an answer!!!]

Yes, the **row** number is increasing.

So what will happen if we do **not indent** the last two lines (which are "Paste", and "Click the search button")

Stop and look here. I don't want you to make the changes. Just watch.

[Only the driver does this: de-indent the last two lines and steps through the script; participants get to watch]

So the repeat command **only repeats the lines** that are tabbed in underneath it, meaning that in this script it didn't repeat after "copy".

Exploration

For the next five minutes, you are going to work on a practice task.

During this time, **experiment** and figure out **as much as you can**. Our goal now is to help you become familiar with CoScripter. Before you begin, save your script and table, then start a new script and a new table. If you have questions, please ask.

[Helpers hand out Exploration Task: movies]

[Helpers go around and ask participants if they have any questions]

[Driver covers up projector]

[Start timer: 5 minutes for this exploration]

SE-questionnaire

This concludes the tutorial. We have a questionnaire for you to fill out.

[Helpers handout Pre-Self-efficacy questionnaires]

[Helpers wait until participants are done with the questionnaire, collect the questionnaires, hand them to the driver]

Before Task 1

We have **two tasks** for you to work on.

Before we start the first task, there are a few things that I would like to remind you of.

- **[TREATMENT]** The version of CoScripter that you will be working with is **an experimental version.** It has more features than what you have seen in the one from the tutorial.
- You might encounter websites that require your personal information, like your email address. We suggest that you stay away from those websites.
- If you want to, you can go back to the scripts from the tutorial.
- During the task, it is Ok to stop the recorder if you just want to explore for a bit.
- When recording, make sure you stay in the same tab. CoScripter does not record switching tabs.

- As a reminder, **run or step through your script frequently** to see if it does what you expect it to**.**
- Also, remember to **SAVE** both your **SCRIPT** and **TABLE FREQUENTLY**

[TREATMENT] [If Idea Garden Treatments, 1. Hand out Task 1 description, 2. Collect exploration task, 3. Save table, save script in tutorial profile, 4. minimize the tutorial Firefox window. Switch to the "Task" window]

[CONTROL] Helpers just collect exploration task.

Our helpers are handing out the first task.

Feel free to write down any ideas or comments that you might have.

You have **30 minutes** for this task.

- If you have any questions, let us know.
- If you finish early, let us know.

You may start now. I will let you know when your time is up.

[Tutorializer covers the projector lens]

[Driver starts entering questionnaire responses]

Task 1

[CONTROL] [Start timer 25 minutes]

[TREATMENT] [Start timer 10 minutes]

[TREATMENT] [Tutorializer notifies the driver when close to the 10-minute mark so Driver stops entering questions]

[CONTROL] [If control, driver continues entering questionnaire responses]

[TREATMENT] [Tutorializer uncovers the projector]

[TREATMENT] [If Idea Garden Treatments, driver maximize Task version. Starts a new script to make sure that the Help button is showing]

View Features

[TREATMENT] 10-minutes

Please stop for a minute and look here.

At the top of your browser tab, you should be able to see a **button.**

Now click it. There are **three choices**.

[Driver clicks first suggestion]

The first choice is “How do I get started?” Please click the first button and read along, where it says (Tutorializer reads first paragraph out loud.)

Please take a minute to read the rest of the suggestion silently.

(Tutorializer gives users about 30 seconds to read)

You can come back to this later if you’d like.

Now close out of the suggestion.

[Driver clicks second suggestion]

The second choice is “What if I can’t find the info I want?” Please click the second button and read along , where it says

(Tutorializer reads first paragraph out loud.)

Please take a minute to read the rest of the suggestion silently.

(Tutorializer gives users about 30 seconds to read)

You can come back to this later if you’d like.

Now close out of the suggestion.

[Driver clicks third suggestion]

The third choice is “Tutorializer reads from screen” Please click the third button and read along , where it says (Tutorializer reads first paragraph out loud.)

Please take a minute to read the rest of the suggestion silently.

(Tutorializer gives users about 30 seconds to read)

You can come back to this later if you’d like.

Now close out of the suggestion.

In addition to these, there are more "specific" versions of the choices in CoScripter that you might encounter at some point of your task.

They can be identified with a face icon next to them.

Now please continue your task.

[TREATMENT] [Start timer 15 minutes]

[TREATMENT/CONTROL] [Timer is up. Start timer 5 minutes]

You have 5 minutes left.

[TREATMENT/CONTROL] [Start timer 5 minutes]

Post-Task 1 Questionnaire

[TREATMENT/CONTROL] [Timer is up.]

Your time is up. Please stop.

Please fill out the questionnaire.

[Helpers hand out Post-Task 1 questionnaires. After participants are done, collect the questionnaires and hand them to the driver]

[Driver turns projector off, enters the rest of the questionnaires]

[TREATMENT] [Helpers: 1. Collect task 1 handout, 2. hand out Task 2, face down on their desks, 3. save scripts, tables, 4 minimizes the “Task” window, and maximize the “Tutorial” window 5 go home, start new table.]

[CONTROL] [Helpers: 1. Collect task 1 handout, 2. hand out Task 2, face down on their desks, 3. save scripts, tables, 4. minimize “tutorial” window, maximize “Task 2” window 5 go home, start new table]

Task 2

The second task has been handed to you.

Feel free to write down any ideas or comments that you might have.

You have 25 minutes to work on it.

Let us know if you have any questions or if you think that you have completed the task.

If you finish early, let us know.

[Start timer: 20 minutes]

You have five minutes left.

[Start timer: 5 minutes]

Ok time's up.

Please stop and fill out the questionnaire. Raise your hand when you're done.

[Helpers: 1. Collect task 2 handouts, 2. hands out questionnaires, 3 saves scripts and tables, 4. Exit out of ALL Firefox windows]

Exit

[Helpers: when a person is done, bring them two receipts, and let them know they can get their money on their way out]

Appendix D: Tasks

(There were one or two tasks in the studies in this thesis. The Popfly study presented in Chapter 3 used the Popfly Movie Task. The CoScripter studies presented in Chapter 4, Chapter 6, and Chapter 7 used the CoScripter Apartment Task. The summative study from Chapter 7 also included the CoScripter Pet Task.)

Popfly Movie Task

Movies at theaters near you?

For this task, create a mashup to find out what movies are on at theaters around Corvallis. Your mashup should display the following information:

1. Movies that are shown at each theater.
2. Information for each movie such as length and show times. You may include other information if you like.
3. News items for each movie.
4. A picture for each movie.

CoScripter Apartment Task

An Apartment near Ohio State University

You are looking for a two-bedroom apartment for you and your friend to stay while going to school at Ohio State University in Columbus, Ohio. Your friend has suggested using **MyCheapApartments.com** to look for these apartments. You and your friend prefer apartments:

- with two bedrooms
- priced under $1300 per month
- within 10 minutes' driving time of the Ohio State University campus

Your task is to ***create a script*** that automates the searching process. Your script should result in ***a record of time*** from each apartment to campus in your table. You may have other information in your table as well.

Why bother with a script? You are going to allow yourself a couple weeks to find an ideal place. Apartments come and go. During this period, you will check the listing once or twice a day to keep up on the latest posts. Checking the listing is laborious, so you would like to write a script to automate the process.

Ideas? Thoughts? Feel free to write them down below!

CoScripter Pet Task

Looking for a Cat to Adopt

You have just moved into a new apartment that allows pets in Corvallis, OR. You have been looking for a cat to adopt on **PetFinder.com**. Specifically, you are looking for a cat from animal shelters within a hundred miles of where you live. In addition, you prefer a cat:

- of any shorthair breeds
- from a reputable shelter; the more reviews the shelter has, the better!

Your task is to ***create a script*** that automates the searching process. Your script should result in ***a record of the number of reviews*** for each shelter in your table. You may have other information in your table as well.

Why bother with a script? You are going to allow yourself a month or so to find the cat you would like to adopt. The adoption listing changes all the time. During this period, you will check the listing once or twice a day to keep up on the latest posts. Checking the listing is laborious, so you would like to write a script to automate the process.

Ideas? Thoughts? Feel free to write them down below!

Appendix E: Post-Task Interview

(This was used for Study 2 in Chapter 6 to measuring learning.)

Post- Interview PART I

Question 2. For the following questions, imagine this scenario. Suppose you would like to buy a novel online at the best price available from multiple bookstores, and you have just begun to record a new script to search for the best price. What would you do? List a few steps.

Question 2.1 If you did not know what websites sell the book, what would you try instead? List a few steps and explain.

Question 3. Suppose the novel is only released in France so the prices of the novel at different online bookstores are in Euros. But you need to know how much the book is worth in US dollars in order to purchase it. What would you do? List a few steps. Keep in mind that you would like to use your script to automate the steps.

Question 4: Where would you place the prices (in Euros) in order to let your script to make use of the conversions? Pick one.

a) in the script you are creating
b) in the CoScripter table
c) in a spreadsheet on your computer
d) in the "Euro" textbox of a currency conversion website

Explain why.

Question 5: What would you do to the resulting price in US dollar so that you have a record of it by the time your script finishes running? Explain your decision.

[If user had trouble getting started and has viewed the how-to-start hint, as Question 6 and its sub-questions. Otherwise, skip them.]

Question 6: If you did not know how to create the script, what would you try to create the script instead?

Question 6.1: Based on your answer to the last question, how would you proceed from there? List a few steps.

Post-Interview Part II

[If their answer to Question 3 does not involve using a webpage, ask this question] Question 3.1 Prompt 1: For Question 3, your answer was…, could you think of a different way to do this? Keep in mind that you would like to have your script do the conversion for you automatically.

[If user has seen the getting started suggestion, ask Question 6 and its sub-questions. Otherwise, skip them]

Question 6-1 [If their answer to Question 6 doesn't involve working backward] For Question 6, your response is … Can you think of another way to start?

Question 6-2 [If their answer to Question 6-1 doesn't involve work backward] Can you think of another way to start based on how you got started with the apartment searching task?

Question 6-3 [If they do NOT know to start with the table] What if you started with the table? What would you do? How would you proceed from there? [Goal is to get them to link output to code]

[If they have seen the starter script suggestion, ask Qa1 and sub questions]

Qa1-2: Suppose you are just beginning to create the book buying script, and you named the first column of your table "Bookstore", then this shows up, what would you do based on what you see in the figure below?

Qa1-3: For the book buying task, suppose the beginning of your script looks like Script A. In what ways is the suggested script like Script A? Why?
[If they ask for clarifications, say in what ways are the two scripts similar to each other? What do they share in common?]

Qa1-4: Fill the blanks in the following sentence with the options below in a way that makes sense to you:

The suggested script is to _____ as Script A is to _____

a. google.com
b. apartments.com (an apartment look-up site)
c. Include
d. the apartment searching task
e. Hide
f. pierrebooks.com (a bookstore)

g. restaurants.com (a restaurant look-up site)
h. the book buying task

Explain your answer.

[If they have seen a webpage suggestion, ask Qa2 and sub questions]
Qa2-2: Continuing with the book buying task, suppose the novel is only released in France so the prices of the novel at different online bookstores are in Euros. But you need to know how much the book is worth in US dollars in order to purchase it. What would you do based on what you see in the figure below?

Qa2-3:

[If their response to Question 3 is correct] Your response to Question 3 was to… In what ways is the suggestion in the figure related to your solution from Question 3? Why?

[If their response to Question 3 is incorrect, provide them with the correct solution] Suppose that one solution is to use a currency conversion website called finance.yahoo.com/currency to convert Euros into Dollars. In what ways is this solution related to the suggestion in the figure? Why?

[If they ask for clarifications, in what ways are they similar to each other? What do they share in common?]

Qa2-4: Fill the blanks in the following sentences with the options below in a way that makes sense to you:

Qa2-4-1: Zipcode in the suggestion (as shown in the figure above) is to _____ as Euros are to _____ in the solution from the previous question.

a. The apartment searching task

b. USPS.com (United States Postal Service)

c. Dollars

d. The book buying task

e. finance.yahoo.com/currency (a currency conversion site)

f. pierrebooks.com (an online bookstore)

Explain your answer.

[If they have seen the second page suggestion, ask Qa3 and sub questions.]

[Note: Qa2 and Qa3 give away each other]

Qa3-1: Suppose you are unable to find the info you need, what would you try? For example, you are unable to find Dollar prices for the novels from the online bookstores' websites, what would you try based on what you see in this figure?

Qa3-3: [If their answer to Qa3-2 is correct] Your response to Qa3-2 is… In what ways is the suggestion in the figure related to your solution related to your solution? Why?

[If their response to Qa3-2 is incorrect] Suppose that one solution is to use a currency conversion website called finance.yahoo.com/currency to convert Euros into Dollars. In what ways is this solution related to the suggestion in the figure?

[If they ask for clarifications, in what ways are they similar to each other? What do they share in common?]

Qa3-4: Fill in the blanks in the following sentences with the options below in a way that makes sense to you:

Qa3-4-1 Ingredients are to _____ is as Euros are to _____

a. ACalorieCounter.com (a site that looks up calories in foods)

b. Calories

c. Euros

d. Recipe.com (a site that looks up recipes)

e. Book prices

f. The book buying task

g. The chicken sandwich recipe

h. finance.yahoo.com/currency (a currency conversion site)

Explain your answer.

Qa3-4-2 Calories are to _____ is as Dollars are to _____

a. ACalorieCounter.com (a site that looks up calories in foods)

b. Calories

c. Euros

d. Recipe.com (a site that looks up recipes)

e. Book prices

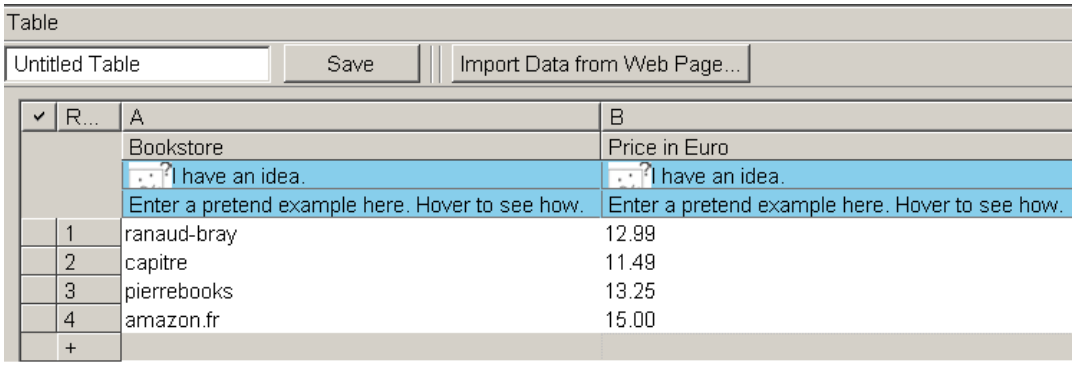
f. The book buying task

g. The chicken sandwich recipe

h. finance.yahoo.com/currency (a currency conversion site)

Explain your answer.

[If user has seen a repeat suggestion, ask Qu1 and Qu2. Otherwise, skip them.]
Qu1 Suppose that you have a list of prices in Euro from different book stores in your table, and you need to look up the Dollar amount for each price on a currency conversion website. What you would do to create a script that would do that for you? List all the steps necessary.



Qu2: Suppose you know how to make your script convert the amount in the first cell. What would you do next? List a few steps.

Post-Interview PART III

Qe: Which of the following paragraphs shows the correct way to use the "repeat" command?

a.
* repeat
* go to "www.rottentomatoes.com"
* copy the cell in the "Movie name"
column of row 1 of the "Movies" scratchtable
* paste into the "Search movies, actors,
critics" textbox
* click the "Search" button

b.
* repeat
* go to "www.rottentomatoes.com"
* copy the cell in the "Movie name"
column of row 1 of the "Movies" scratchtable
* paste into the "Search movies, actors,
critics" textbox
* click the "Search" button

c.
* go to "www.rottentomatoes.com"
* repeat
* copy the cell in the "Movie name"
column of row 1 of the "Movies" scratchtable
* paste into the "Search movies, actors,
critics" textbox
* click the "Search" button

d.
* go to "www.rottentomatoes.com"
* repeat
* copy the cell in the "Movie name"
column of row 1 of the "Movies" scratchtable
* paste into the "Search movies, actors,
critics" textbox
* click the "Search" button

Explain why.

Question 1: For your choice of script from Qe, answer the following questions:

(1) Does any part of the script change each time the computer runs through "repeat"?

(2) If yes, circle the part that changes AND explain how it changes.

(3) If no, explain why not.

Qz. Imagine your choice of script from Qe being placed in the box below to form a combined script. Suppose the combined script is currently on the last line of the inserted script, what would happen if you let the script keep running?

* go to “www.google.com”
* enter “movies in Corvallis” into the “Search” textbox
* click the “Search” button
* click the “www.fandango.com/corvallis_or_movietimes” link
* extract the “Movies” table

Ask Jill to place your choice from Qe here

* go to “www.MrMovieTimes.com”
* enter “97330” into the “Zip” textbox
* click the “Search” button

Qh. Given the table below and the combined script from Qz, when will the “repeat” command finish going through the “Movies” column?

Table

Movies | Save | Import Data from Web Page...

	R...	A	B
		Movies	Rating
		I have an idea.	I have an idea.
		Enter a pretend example here. Hover to see how.	Enter a pretend example here. Ho
	1	The Muppet	98%
	2	Twilight Saga: Breaking Dawn Part 1	26%
	3	Hugo	94%
	4	Arthur Christmas	92%
	5	Happy Feet 2	42%
	+		

Qi. What will happen when the “repeat” command finishes going through the “Movies” column?

Appendix F: Post-Task 1 Questionnaire

(This questionnaire was used in the summative study presented in Chapter 7. This particular version was used by the Combined treatment. The Strategy and Programming treatments' questionnaires had the same questions as in this questionnaire with screenshots of their respective Idea Garden prototype.)

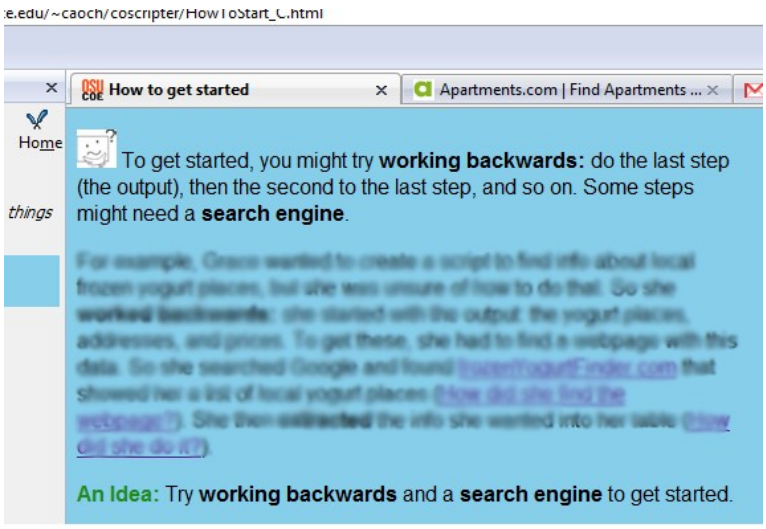
Questionnaire 3 (C)

Please rate how much agree with the following statements.
If you never saw the feature a statement is referring to, you should choose "**Never Encountered**".
You are welcome to write down any comments you might have.

Feature 1:

This feature **helped** me to accomplish my task:

Never Encountered	Strongly Disagree	Disagree	Neither Agree Nor Disagree	Agree	Strongly Agree
●	●	●	●	●	●

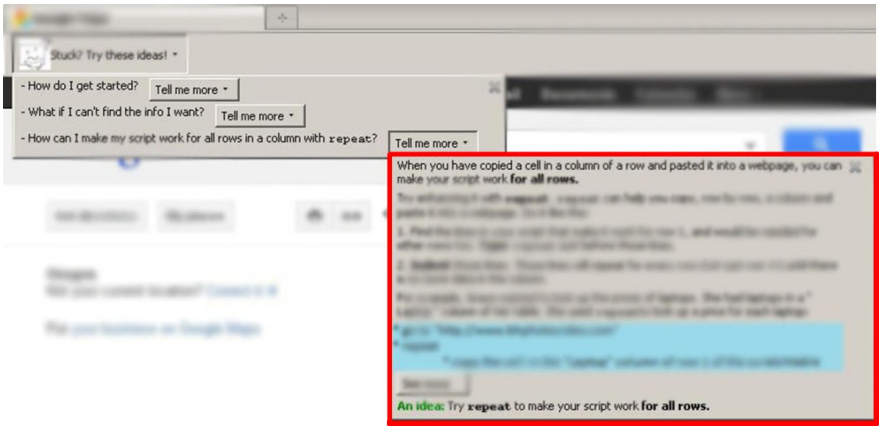


Comments:

Feature 2:

This feature **helped** me to accomplish my task:

Never Encountered	Strongly Disagree	Disagree	Neither Agree Nor Disagree	Agree	Strongly Agree
●	●	●	●	●	●

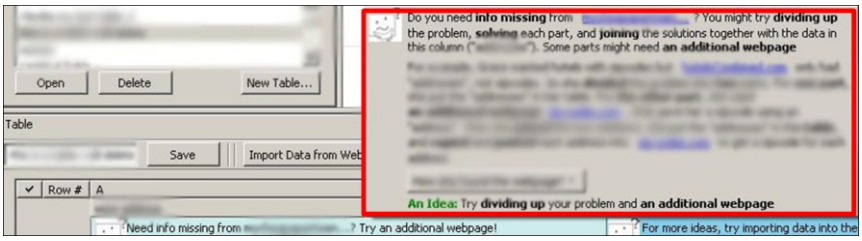


Comments:

Feature 3:

This feature **helped** me to accomplish my task:

Never Encountered	Strongly Disagree	Disagree	Neither Agree Nor Disagree	Agree	Strongly Agree
●	●	●	●	●	●

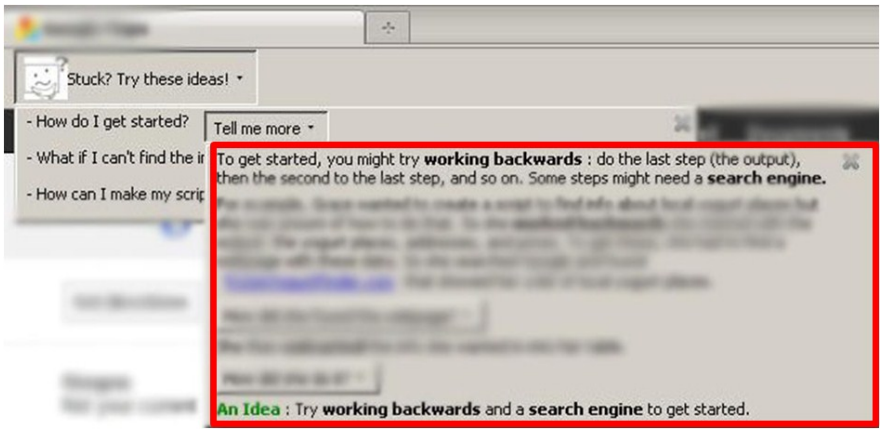


Comments:

Feature 4:

This feature **helped** me to accomplish my task:

Never Encountered	Strongly Disagree	Disagree	Neither Agree Nor Disagree	Agree	Strongly Agree
●	●	●	●	●	●

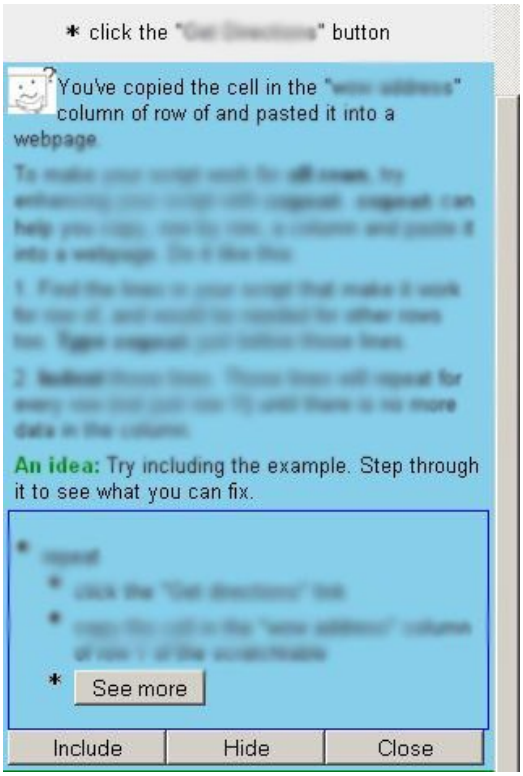


Comments:

Feature 5:

This feature **helped** me to accomplish my task:

Never Encountered	Strongly Disagree	Disagree	Neither Agree Nor Disagree	Agree	Strongly Agree
●	●	●	●	●	●

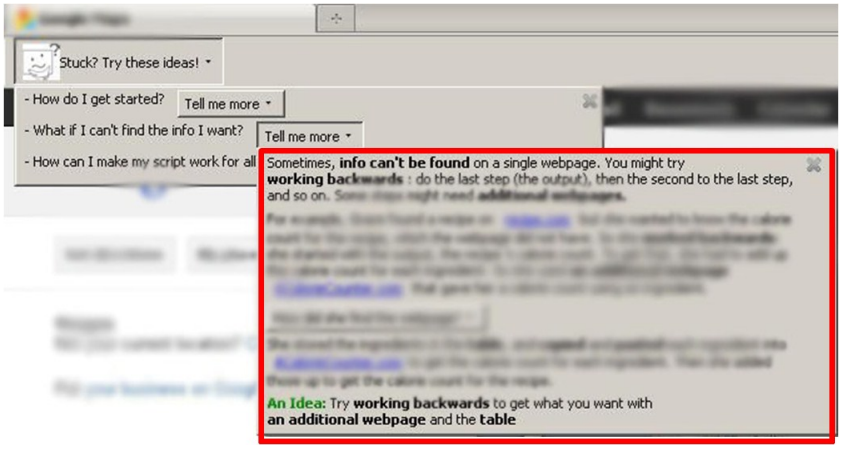


Comments:

Feature 6:

This feature **helped** me to accomplish my task:

Never Encountered	Strongly Disagree	Disagree	Neither Agree Nor Disagree	Agree	Strongly Agree
●	●	●	●	●	●



Comments: